Bachelorarbeit an der Technischen Fakultät der
Albert-Ludwigs-Universität Freiburg im Breisgau

Solving the Flatland Challenge with Multi-Agent Pathfinding (MAPF)

vorgelegt von
Marco Bernhard Kaiser

am 15.07.2021

am Lehrstuhl von
Prof. Dr. Bernhard Nebel

Tag der Abgabe:
15.07.2021

Gutachter:
Prof. Dr. Bernhard Nebel

Zusammenfassung

Bei der ständig steigenden Zahl von Fahrgästen, die mit Zügen transportiert werden, sollte deren individuelle Planung robust, möglichst schnell und effizient sein. Um interessierte Menschen auf der ganzen Welt zu motivieren und ihre Ideen zu diesem Problem einzureichen, wurde die FLATLAND competition ins Leben gerufen. Diese Herausforderung beschäftigt sich mit virtuellen Umgebung, in denen Züge ihr Ziel finden müssen. Diese Umgebungen bilden damit Multi-Agent Path Finding (MAPF) Instanzen. Der folgende Ansatz basiert auf einem der derzeit führenden MAPF Algorithmen, genannt Conflict-Based Search (CBS) [Sha+15]. Die in der folgenden Arbeit erbrachte Erweiterung berücksichtigt bestimmte Eigenschaften der FLATLAND competition Umgebungen, was dann zu einem Pruning von unnötiger Exploration bestimmter Zweige führt. Mit dieser Erweiterung kann sie zu geringeren Rechenzeiten in den vorliegenden Instanzen führen. Nach der empirischen Auswertung und dem entsprechenden Vergleich mit einem anderen, bereits eingereichten Solver, kann man zu dem Schluss kommen, dass der vorgestellte Ansatz keine signifikante Ergänzung zu den derzeit entwickelten Algorithmen darstellt und gegen andere nicht einmal eine Chance hat. Ein Solver, der auf bereits entwickelten Algorithmen basiert, kann möglicherweise eine bessere und schnellere Lösung liefern, auch wenn er nicht für diese speziellen Arten von Umgebungen entwickelt wurde.

Abstract

With the ever-increasing number of passengers transported by trains, their individual planning should be robust, as fast as possible and efficient. To motivate interested people around the world to submit their ideas on this problem, the FLATLAND competition was launched. This competition deals with virtual environments in which trains have to find their destination. These environments thus form Multi-Agent Path Finding (MAPF) instances. The following approach is based on one of the current leading MAPF algorithm called Conflict-Based Search (CBS) [Sha+15]. The extension yielded in the following work takes into account certain properties of FLATLAND competition environments, which then leads to pruning of unnecessary exploration of certain branches. With this extension, it can lead to lower computation times in the present instances. After the empirical evaluation and the corresponding comparison with another already submitted solver, it can be concluded that the presented approach is not a significant addition to the currently developed algorithms and does not even stand a chance against others. A solver based on already developed algorithms may be able to provide a better and faster solution, even if it was not developed for these specific types of environments.

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Freiburg, den DATUM (Marco Bernhard Kaiser)

Contents

Chapter 1

Introduction

In the history of mankind, all kinds of means of transportation have been invented, using the air, water or the ground. Among them, large railroad networks have been created, which are today maintained by railroad companies such as the SWISS FEDERAL RAILWAYS (SBB) or the DEUTSCHE BAHN AG (DB). Day after day, these companies face the logistical challenge of scheduling their trains correctly to transport passengers and goods to their destinations. As it is in their best interest to solve this dispatching task as effectively as possible, they have outsourced this problem by founding the FLATLAND competition. This competion is hosted by `aicrowd.com` and has already taken place in the previous two years (2019, 2020). It is based on virtual train network environments in which a varying number of trains must reach their destination station. In order to be able to adapt parts of the submitted solutions to the actual planning, this is kept as close to reality as possible. Therefore, these trains may have different speed values and may have malfunctions during the journey.

Such a problem instance, with multiple agents that must reach its goal in a given environment, is called **Multi-Agent Path Finding** (MAPF). A valid solution of such an instance is a path for each agent that starts at its initial location and eventually leads to its destination. This path is described by the different locations at the corresponding timestep where the agent is located. In addition, the NP-hard problem must provide collision free paths of the agents, which means that no two agents can be at the same location at the same timestep. There are several methods to solve this type of instance, but in the following approach, a modification of the optimal **Conflict-Based Search** (CBS) [Sha+15] is used. The main idea behind this type of search is to first schedule each of the agents individually without considering the others. Subsequently, the resulting conflicts can be resolved by constraining the agents at specific locations and timesteps, which then eventually leads to a valid solution of the given MAPF instance. In the following, the approach is based on minimizing the number of conflicts that can occur between two agents and thus reducing computation time. Moreover, the idea behind conflict minimization compared to the standard CBS algorithm is to eliminate certain parts of the environment where the agent can not choose a different path. In the related work of *Pairwise Symmetry Reasoning for Multi-Agent Path Finding Search* [Li+21b] they deal with a similar technique found independently of this work. The differences will be presented later when the needed terms are defined.

Chapter 2

Background

2.1 The Flatland environment

This section is intended to convey essential information about the given FLATLAND competition environment. The reader is given a brief overview of how to operate the given MAPF instance. First of all, the **basic challenges** every train has to face until the destination target is reached are presented. In order to anticipate directly, the trains in these MAPF instances are also called agents and are later defined in more detail. Further on, the given observation on the environment, the possible heuristic values and how a train can eventually reach its goal are discussed. Last but not least, the **visual representation** of an environment will be considered.

2.1.1 Basic challenges

The given environment consists of a **two-dimensional grid** with a given height and width. In these borders, there are well defined locations, comparable to a coordinate system. At each **location** in the grid there is a specific **type of rail** which indicates the possible paths to drive through. The locations and rails are defined as follows:

Definition 1 (Location)**.** *Let h be the height and w the width of an environment, where $h, w \in \mathbb{N}$. A locating l is then defined as $l = (y, x)$ where $y \in [0, height], x \in [0, width]$ and $y, x \in \mathbb{N}$.*

Definition 2 (Rail)**.** *Let l be a location in the environment. Every location l in the environment contains a certain type of rail. The basic rail types can be found in Figure 2.1. They can be rotated in steps of $45°$ and mirrored along the North-South or East-West axis.*

Those rails are the foundations of every **agent** (train), which needs to navigate through the specific parts of the given environment to eventually reach its **target** location (railway station). To indicate the respective state of an FLATLAND competition environment, **timesteps** defined are used as follows:

Definition 3 (Step)**.** *In a step, every train get assigned an action (Def. 6), which are then executed.*

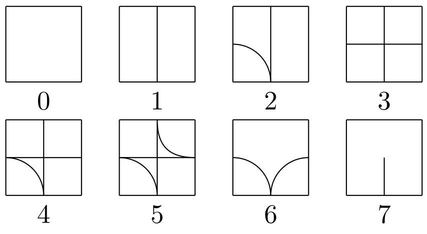


Figure 2.1: All basic rail types in the Flatland environment, adopted from [Moh+20]

Definition 4 (Timestep)**.** *Let t be a timestep where $t \in \mathbb{N}$. t indicates how many steps already have been performed. Every environment t gets initialized with $t = 0$ and t will get increase by one with each passing step.*

During an argent's journey, it always has a well-defined **direction**, which is important to be able to navigate on the different types of rails. Just if an agent can show a specific direction, it is able to move into particular locations. Those directions are defined as:

Definition 5 (Direction)**.** *A direction d is either North, East, South or West.*

Thus an agent can eventually reach its target, it needs to provide an **action** at each timestep, which will be executed in the next timestep. Depending on the rail type the agent is placed on and the direction it is currently facing, it can apply an action and move further on. The different possible types of actions an agent can somehow apply, are defined as:

Definition 6 (Actions)**.** *An action act is either Left, Forward, Right or Wait.*

To check if an agent **is able to apply** an action on a specific rail type, the direction it is currently facing needs to be given. Then we can simply draw an **imaginary arrow** following the direction that the agent is currently facing on the desired rail type from Figure 2.1. Afterwards, you can simply analyse if there is an edge to the indicated place of the action.

Example 1 (Apply an action, 1)**.** *If an agent is placed on a rail of type Nr. 1,3 (without mirroring or rotating) and is facing the North direction, it is able to apply the Forward action.*

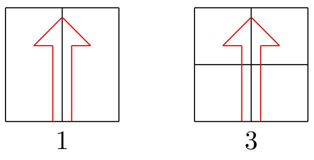


Figure 2.2: Arrow overlay of rail types Nr. 1, 3

Example 2 (Apply an action, 2)**.** *If an agent is placed on a rail of type Nr. 4,5 (without mirroring or rotating) and is facing the East direction, it is able to apply the Forward and Right action.*

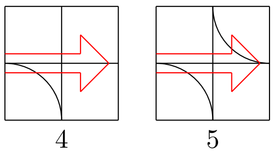


Figure 2.3: Arrow overlay of rail types Nr. 4,5

Obviously, there is no possible action to get into its previous location, once an action has been executed. However, there are two possible methods of returning to a previous location. The first trivial method is to find a curve in the environment and thus navigate back. The other method is to use a rail from type Nr. 7 out of Figure 2.1. This rail is the **dead end**, which can be used to reach the previous location and change the direction by 180°.

Example 3 (Apply an action, 3)**.** *If an agent is located on a rail of the type Nr. 7 (without mirroring or rotating) and is facing the North direction, it is able to apply the Forward action. The resulting direction will then be South.*

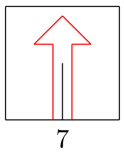


Figure 2.4: Arrow overlay of rail type Nr. 7

At this point, it is known how an agent can change its location by applying various actions from the given action set. In order to apply these actions correctly, the initial location and the particular direction in which the agent is facing must first be known. Even after an agent has begun its path, the location and direction it possesses must be **constantly observed**. These problems are solved by defining **states** in which all this information is recorded. With the definition of a state, the concept of a **agent** can be fully defined:

Definition 7 (State)**.** *Let t be a timestep, l a location and d a direction. A state s is then defined as $s = (t.\ l, d)$.*

Definition 8 (Agent)**.** *Let i be a state and $trgt$ a location. $i = (t.\ l, d)$ represents the initial state and has a value with $t \geq 0$, since the agent does not need to be placed directly on the grid. $trgt$ represents the target the agent have to reach, so $agt = (i, trgt)$.*

As described, the main goal of an agent is to find a **path** from its initial state to its target using the intended actions that leads to further states. This **journey** must be documented for further investigation after the agent reaches its destination. Such a journey or path is called a **solution** and is defined as:

Definition 9 (Solution)**.** *Let $agt = (i, trgt)$ be an agent. A solution soln gets assigned to agent agt and is a linear list of [state, action] pairs. soln starts with the state i and ends with a state $(t^{end}.\ l^{end}, d^{end})$ where $l^{end} = trgt$. The action next to a state $(t.\ l, d)$ must lead into to the next state $(t + 1.\ l', d')$. It follows that the ending pair is just a state and provides no further action, so:*

$$soln(agt) = [[\underbrace{(0.\ l^0, d^0)}_{i}, act^0], [(1.\ l^1, d^1), act^1], ..., (t^{end}.\ l^{end}, d^{end})]$$

Example 4 (Solution)**.** *Let* $agt = ((11.\ (1,2), West), (0,2))$ *be an agent. A valid solution soln for the agent agt for the example environment can be examined in Figure 2.5.*

Solution *agt*:

0. (1, 2), *West* → *Forward*
1. (1, 1), *West* → *Forward*
2. (1, 0), *West* → *Right*
3. (0, 0), *North* → *Right*
4. (0, 1), *East* → *Forward*
5. (0, 2), *East*

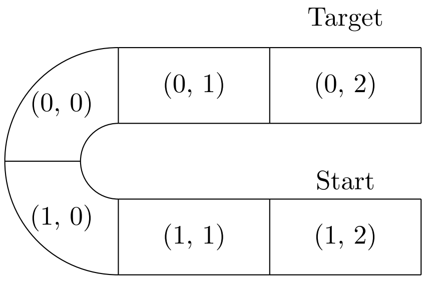


Figure 2.5: Example solution and environment

Definition 10 (Access state in solution)**.** *Let soln be a solution and* $k \in \mathbb{N}$ *a given index. To access a state* $s = (t.\ l, d)$ *with* $t = k$*, use:* $soln(k) = s$*.*

As mentioned, an agent can locate its way around in the environment by providing an action at each timestep, which will be executed in the next step. But that is not the case, if we introduce **different speed values** of agents. This is supposed to mean that certain agents need to wait longer than the one timestep to finally execute the applied action. In comparison to the reality, there are fast passenger trains, fast freight trains, slow commuter trains, etc.

Definition 11 (Speed)**.** *Let* $n \in \mathbb{N} \setminus 0$ *denote the required timesteps an agent must wait at a location to perform an action and thus change its location. So the speed spd of an agents is defined as* $spd = \frac{1}{n}$*.*

To make the FLATLAND competition even harder and stay as close to reality as possible, the agents are not perfect and may have **malfunctions**. When initializing an environment, the user has the option to set whether these additions should be taken into account. To do this, an interval of natural numbers must be passed to a provided function that generates these random malfunctions. When an agent is malfunctioning, a uniformly random number is selected from this interval. This random number represents the timesteps the agent must wait before it can continue. When such a malfunction occurs, there is no other option but to reschedule the current solutions, in this approach this is done by performing a **new search**. In this search, the initial states are set to the current states the agents were in when the malfunction occurred. The exact definition can be seen below:

Definition 12 (Malfunction)**.** *Let a and b, numbers defined by the user, for which holds* $a, b \in \mathbb{N}$ *and* $a \leq b$*. The malfunction duration* $m \in \mathbb{N}$ *an agent agt can suffer from is a uniformly random picked number* $m \in [a, b]$*. Therefore agt needs to wait for m timesteps to continue with actions.*

2.1.2 Search procedure / observation / heuristic

Having learned the basic challenges of the Flatland competition, we will now outline how an agent needs to maneuver towards its target location. At the **initialisation** of an environment, every agent will be assigned an initial state, where no agent shares the same location. After this process the agents are not placed on the grid right away, rather they are waiting for the **departing**. To depart an agent, we need to provide an action, except the *Wait* action. Then it will be placed on the grid and will start its journey. Before placing the agent on the grid, it is not vulnerable for other agents, because it can not interfere with them.

If an agent has reached its target, there are **two possible options** to choose from. Either the agent will be removed at its target location, or the solver needs to compute a solution for every agent thus every agent will reach its target at a specific time. The usual way to use the environment is to remove the agents, which is the only option considered in this approach. The real world analogy for removing the agents at their targets is putting them on siding at their target railway stations.

The Flatland competition provides a MAPF instance with a lot of challenges to deal with, nevertheless there is **full observation** at any time about the current status of every agent. Another easement is that we can use the **perfect heuristic**. This heuristic get created at the initialisation of the environment and thus the computation time who not count into the computation time of the algorithm per se.

2.1.3 Visual representation

In Figure 2.6 you can see an example of a **rendered environment** where the trains and their stations are shown. You can even see the black crosses above the agents, which are supposed to be the **visual indicator** indicating that this particular agent is currently malfunctioning. Therefore, it will be out of service for a randomly specified period of time.

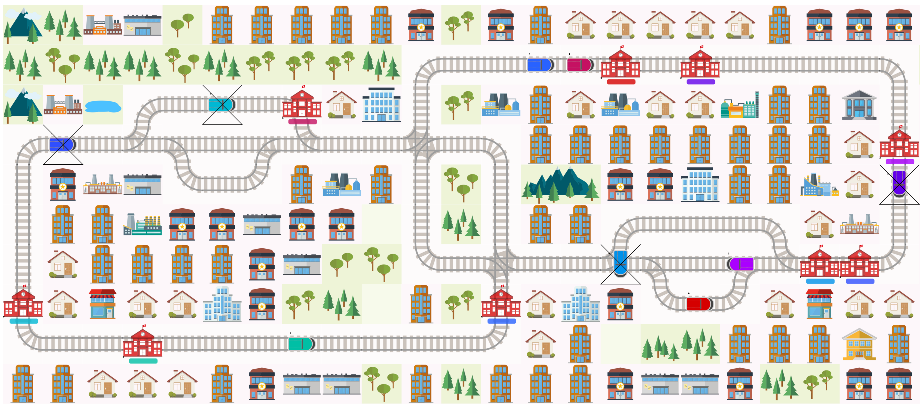


Figure 2.6: Rendered Flatland environment (Test 6)

Since the rails and the agents are not very well visible, **my own cutout** of the environment will be illustrated. This is necessary in order to portray the functionality of the following approach, which will also be used in other examples. Here an example of a very small environment and two agents is illustrated:

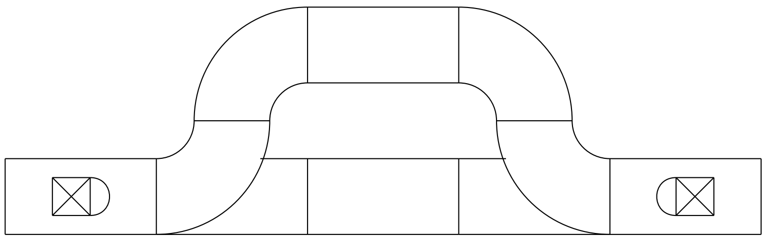


Figure 2.7: Own drawn cutout of an very small environment

2.1.4 Main task of the competition

To resolve a given Flatland environment, the placed agents must reach their destination in a given timestep that is smaller than the given maximum step. This maximum step varies for some environments with other dimension sizes and is defined as follows:

$$\text{max steps} = 4 \cdot 2 \cdot (\text{dimension y} + \text{dimension x} + 20) \tag{2.1}$$

Thus, either the maximum number of steps in the respective environment has been reached or all agents have successfully reached their goal. The goal of Flatland competition can therefore be formulated as follows:

With the above challenges, how do you get as **many agents** as possible to their destination with as **few timesteps** and as little computation time as possible?

2.2 Using the Flatland Python package

To provide uniformity for each participant of the FLATLAND competition there is a given PYTHON environment. The package provides its own basic functions to extract all the needed information of the given environment, how to go one step further by providing actions to specific agents and how to extract the current state of all agents in the environment. In the following sections, the main features are presented with the information on how to use them properly. More information about the package can be found here at the PYPI-PACKAGE [ML21]. Also, for more references about the API, you can look up the documentation at the respective Website [Bau+21].

2.2.1 Downloading the Flatland environment

To download and use the provided PYTHON-PIP-PACKAGE, just install it with the package management program **pip** by inserting following command into the terminal:

```
pip install flatland-rl
```

Code Listing 2.1: Install flatland-rl package

2.2.2 Setting up an environment

At first you obviously need to create and initialize an environment. The initialisation is based on the parameters the user provides. So he must provide the **dimension** of the grid, the connected **railways** in the grid and the **number** of the agents. As mentioned in chapter Background 2, the **malfunction** and/or the **speed** of the agents can be provided as additional parameters. To provide the dimension the user just need to provide two natural numbers, which will represent the height and the width of the grid. The same applies to the overall number of agents, where the user needs to provide a natural number. Nevertheless, keep in mind that the creation process **can take really long** based on this parameters.

Creating the rail generator

Here the user must create the network of the rails and how they are ordered in the two dimensional grid. So the user must provide several needed parameters. Those parameters are the number of the railway stations $\in \mathbb{N}$, the maximal number of rails between cities $\in \mathbb{N}$ and the maximal number of rails in cities $\in \mathbb{N}$. Optionally, the user also can provide the seed $\in \mathbb{N}$ and several more optional parameters, which are not elaborated any further in this section. In the provided package the user can make use of multiple rail generator objects. Every rail generator provides its own style how to properly arrange the rails in the environment. For this example we will use the **sparse_rail_generator** to define the rail generator:

```
from flatland.envs.rail_generators import sparse_rail_generator
```

```
rail_generator = sparse_rail_generator(
                    max_num_cities=3,
                    seed=100,
                    max_rails_between_cities=3,
                    max_rails_in_city=4)
```

Code Listing 2.2: Rail generator

Creating the malfunction generator

To specify the malfunctions of the agents the user needs to provide a rate $\in [0, 1)$, the minimal duration $\in \mathbb{N}$ and maximal duration $\in \mathbb{N}$ to the **MalfunctionParameters** object. This object then acts as the stochastic data and gets then passed further into the actual malfunction generator. Example how to create the malfunction generator:

```
from flatland.envs.malfunction_generators import \
                                  MalfunctionParameters, \
                                  ParamMalfunctionGen

stochastic_data = MalfunctionParameters(
                    malfunction_rate=1/200,
                    min_duration=3,
                    max_duration=10)

malfunction_generator = ParamMalfunctionGen(stochastic_data)
```

Code Listing 2.3: Malfunction generator

Creating the speed generator

If the user wants an agent without a standardised velocity, the user must specify a dictionary with the speed values as keys and the respective possibilities as items. To successfully define this dictionary, the user must ensure that the sum of all elements, i.e. all probabilities of the respective speed, must add up to exactly one. Such a dictionary can then get passed to the rail generator corresponding schedule generator. This means if the user used the sparse_rail_generator earlier the user also must utilize the **spars_schedule_generator** now. Example how to initialize the speed object with the speed data dict:

```
from flatland.envs.schedule_generators import \
                                  sparse_schedule_generator

speed_data = {1 : 1 / 4, 2 : 1 / 2,
              3 :   0.0, 4 : 1 / 4}

speed_generator = sparse_schedule_generator(speed_data)
```

Code Listing 2.4: Speed generator

Creating the environment

Finally the user can now set up the actual environment. Also keep in mind that it is not necessary to provide the speed or malfunction data. So the

malfunction_generator and the **schedule_generator** are totally optional and can be omitted for any environment. In order to successfully create an environment the user must provide the following necessary parameters. At first the dimensions of the grid, then the overall number of agents and finally the **rail_generator**. There are more optional parameters like an observation but those are not discussed further and can be looked up in the official documentation of the package. Example how to create the environment:

```
from flatland.envs.rail_env import RailEnv

env = RailEnv(
      width=25,
      height=25,
      number_of_agents=4,
      rail_generator=rail_generator,
      schedule_generator=speed_generator,
      malfunction_generator_and_process_data=malfunction_generator)
```

Code Listing 2.5: Flatland environment

2.2.3 Resetting the environment

To fully initialize the environment and finally use the environment in the code the user also need to reset it. By calling the **reset function** the agents get placed at their initial locations and are now waiting to depart. The return value of this function is a iterable tuple with two elements. The first of them is the neglectable observation which is only used in the Reinforced Learning (RL) branch. The second item is the important **information dictionary** which provides the needed status about every agent. This dictionary contains the current information about the malfunction and **how many steps** the agents need to wait until they can move further and proceed with an action. Example how to reset the environment:

```
obs, info = env.reset()
```

Code Listing 2.6: Reset Flatland environment

2.2.4 Extracting needed informations

At this point the user have successfully initialized the environment and the user can theoretically execute any action for every agent, but without the computation of the CBS solver it is just arbitrarily and would not makes sense at all. To properly use the solver the user need to extract some parameters. These can be extracted by some built in functions. At first the user needs to extract all the **agents informations** that means their initial state, target and speed. This can be made by looping through all agents in the environment and accessing certain variables. Example to extract these informations:

```
for handle, agent in enumerate(env.agents):
    initial_position = agent.initial_position
    initial_direction = agent.initial_direction
    target = agent.target

    # Informations about the speed is saved in this dict
    speed_dict = agent.speed_data

    speed = speed_dict["speed"]  # Actual speed
```

Code Listing 2.7: Extract agent informations

These informations are necessary for the initial search. If the user introduces malfunctions to the environment, the solver must replan the agents schedules based on how long such a malfunction lasts. It is to be noted that even directly after resetting the environment a malfunction of an agent can occur. To extract these information and pass into the solver we need to investigate the previous mentioned **information dict**. Here an example how to print the duration of the malfunction from every agent is given:

```
_, info = env.reset()  # observation can be ignored

for handle, time_remaining in info["malfunction"].items():
    print(f"Agent {handle}: {time_remaining} timesteps!")
```

Code Listing 2.8: Extract malfunction informations

The solver also needs to know where and how an agent can move if it can point a certain direction. To save time in the computation, a map with all valid locations and the possible next locations is computed. This can be done with the function **get_transitions**, where the user must specify a location on the grid and a specific direction. This function is a member function of **rail**, which is also part of the environment. To use this function, the user needs to call it with **env.rail.get_transitions**. This then ultimately leads to the fact that during the initialisation of the CBS solver you loop over every location and every direction. These values get passed into the mentioned function and you will recieve a tuple with the size of four. In this tuple are **either zeros or ones** which indicates the direction the agent can move next. Here are some made up examples for better understanding (return values are also made up and do not corresponds to the *env* variable which have been defined earlier on):

```
>>> env.rail.get_transitions(5, 5, 0)  # Next at 5, 5, North
(0, 1, 0, 0)  # east --> 5, 6

>>> env.rail.get_transitions(5, 7, 2)  # Next at 5, 7, South
(0, 1, 1, 0)  # east / south --> 5, 8 / 6, 7

>>> env.rail.get_transitions(7, 7, 3)  # Next at 7, 7, West
(1, 0, 0, 1)  # north / west --> 6, 7 / 7, 6

>>> env.rail.get_transitions(0, 0, 1)  # Next at 0, 0, East
(0, 0, 0, 0)  # invalid and no successor location
```

Code Listing 2.9: Examples for get_transitions

At this point all the needed information about the agents and the possible next locations have been extracted. Now there is only one important step left:

Extract the given perfect heuristics. As explained earlier, the perfect heuristic for every agent get created at the initialisation of the environment and we can simply extract this and pack it into an four dimensional array. This array is located in the member **distance_map** of the environment and can be extracted with the member method **get**. The result is an array with the following four dimensions:

$$\text{Agent number } \times \text{ Env height } \times \text{Env width } \times \text{ } 4$$

This array can be used to access the **exact steps** that a single agent needs to get to the destination when it is in a certain place and facing a given direction. Note that the assumption about the perfectness is only true when no other agent is interfering this path. Example how to access the value:

```
env_map = env.distance_map.get()

env_map[1][20][5][2]   # agent=1 | y=20 | x=5  | dir=South

env_map[5][1][12][3]   # agent=5 | y=1  | x=12 | dir=West
```

Code Listing 2.10: Examples get heuristic

2.2.5 Proceed one step

At this point we have initialized the environment, extracted the needed variables and the CBS solver have calculated valid routes for the given agents. Now the next challenge is to **pass those actions** into the environment so the agents perform its given actions. We also need to check for malfunction which can occur while using the step function. To proceed one step the user must make use of the **env.step** function, which takes an actions dictionary. A corresponding dictionary is formed in such a way that the keys are the handles of the agents we want to perform an action. The items are the actual actions of the corresponding handles of each agent, which should be applied. Such an action dict could look like:

```
# Agent 0 --> Left, Agent 1 --> Wait, Agent 2 --> Right
# Agent 3 --> Wait, Agent 4 --> Forward

{ 0 : 1, 1 : 4, 2 : 3, 3 : 4, 4 : 2}
```

Code Listing 2.11: Examples action dict

It should be noted that the CBS solver is calculating **individual actions** at each location. This means when we are using speed less than one we could pass in actions in the step function without the agent needing them at the moment and we would use up all this action way before the end. The same applies when using malfunctions. For this reason, we should always determine in this step whether the current agent actually requires an action, and otherwise we simply ignore it and do not enter an action. This is checked by using the **action_required** function, which takes an agent as parameter and returns true if an agent really needs another action. Example of how to go one step further, paying attention to whether the respective agent also needs an action.

```
action_dict = {}
for handle, agent in enumerate(env.agents):
    if (env.action_required(agent)):
        action_dict[handle] = CBS.agent.action()  # Black Box

obs, rewards, done, info = env.step(action_dict)
```

Code Listing 2.12: Examples using step function

Obviously, many return values are available for further investigation. However, **observation** and **rewards** are not needed for this approach and are only used in the Reinforcement Larning (RL) branch. Therefore they are ignored and the focus is set to the values **info** and **done**. The value info represents the mentioned information dictionary. Among other things, it contains the information about the malfunctions of the agents in the current timestep, so that it can be seen which agent is now malfunctioning and must be **replanned**. It is important to remember that multiple agents can have a malfunction at the same timestep and even if other agents are currently malfunctioning, other malfunctions can occur. So it only needs to be rescheduled when new malfunctions have occurred.

2.2.6 Detect the ending

Currently there is only one question left to answer to the final evaluation of the respective environment. This question is:

When should you **stop evaluating** your current environment and move on to another?

The previously addressed parameter **done** is now added here to this answer. This return value done is actually a dictionary with the handle of the agents and a truth value if the agent is in the goal or not. In this dictionary is also the key "__all__"which specify if **all agent** have reached the goal. Obviously this is the key to test for in every step and if this is true the solver have successfully brought every agent into its target location. There may also be the case that a solver returns an invalid solution, which can lead to a complete deadlock of the agents and no agent can reach its target location. Testing if there is a deadlock is not very efficient and therefore it may be better to let only the following and final case occur. The last case occurs when the **maximum number of steps** in the environment has been reached before all agents have reached their destination, for some reason.

2.3 Standard CBS algorithm

The CBS algorithm [Sha+15], or Constrained-Based Search, was mentioned several times in the previous section, but there was still no explanation and why there should be a **modification** to use it for the FLATLAND competition. Therefore, in the following section, starting with the initialization and the further procedure, we explain how this **optimal** algorithm works.

2.3.1 Initialisation

At the beginning of the CBS algorithm there are just the given agents with their starting locations and targets. The algorithm now starts by computing the path from each individual agent without considering other agents at all, using the low level search. If there is an agent which can not reach its target the algorithm will abort and **return false**, because an unsolvable MAPF instance was provided to the solver. But if each agent has a valid path from the initial state to its target the high level search will now start and will analyse **pairwise confilicts** of the agents. The high level search solve them by constraining each other.

2.3.2 Low level search

CBS consist of two different types of search. One of those is the low level search where the **A* (A-Star)** pathfinding algorithm is used. This algorithm is applied to each agent in the environment and starts with the initial state of that agent. This initial state is converted to a **lower level node** and these are expanded until one of these nodes contains a state with the location of the agent's target. Such low level nodes get defined as following:

Definition 13 (Low level node)**.** *A low level node consists of the tuple* (s, h, g, prt)*, where* s *is a state,* h *is a heuristic value,* g *is the tentative score, and* prt *is a pointer to the parent node.*

The difference with "usual"application of a pathfinding algorithm is that this type of search takes into account the constraints set during the high level search. This means that the algorithm does not expand low level nodes whose location may not be entered in a given timestep. To avoid constrained locations, the solver can apply the action *Wait* or choose another solution, depending on the cost of the current open states. Since it is an **optimal algorithm**, there must also be an optimal solution in the low level search. This can be achieved by using an admissible heuristic such as the Manhattan distance [Cra17], since the A* algorithm provides an optimal solution when a heuristic is admissible. In the case of the FLATLAND competition environment, the supplied **perfect heuristic** can be used, which is of course an admissible heuristic. This heuristic specifies the exact steps required to reach the target at each location and in each direction. It should be noted that the constraints do not change these heuristic values and the low level search becomes even more difficult when multiple constraints are involved.

2.3.3 High level search

The second type of nodes that make up the CBS algorithm are the high level nodes. These are even larger and contain the information about each agent's solution and its current constraints. Those get defined as:

Definition 14 (High level node)**.** *A high level node consists of the tuple* $(id, soln - lst, con - lst, costs)$*, where id is a unique natural number,* $soln - lst$ *is a list of the solutions of all agents,* $con - lst$ *is a list of the constraints of all agent, and costs is the sum of the costs of all solutions in* $soln - lst$*.*

Example 5 (High level node)**.** *Here you can see the basic structure of an example high level node with id* X*:*

<u>High level node, id: X</u>

Solution agent 0:

...

Constrains agent 0:

...

Solution agent n:

...

Constrains agent n:

...

Overall cost: c

After the algorithm has found the different solution for each agent those will be passed into the initial high level node. By passing the solutions into such an node, the costs of all those solutions will be summed up and this node will be stored in a **fitting data structure**, for example a Fibonacci Heap [Koz92], where we can easily extract the node with the lowest overall solution cost and expand this one further. This heap represents an Conflict Tree [Sha+15] (CT) which keeps track of every solution based on the set conflicts.

First, the respective high level node with the lowest total cost is picked from the conflict tree or rather the heap. Then a **pairwise iteration** is performed over all solutions of this node and checked for vertex or edge conflicts. To define the terms vertex and edge conflict, it is necessary to introduce some operations between states and how they can be compared. These operations compare states in terms of their properties (timestep t, location l, direction d), with some operations ignoring certain properties when comparing. One of these operators will be the normal comparison that compares each of the properties. The other will be an operator that ignores the location. The last operator will then compare only the location. Formally, they are defined as follows:

Definition 15 (Comparing states)**.** *To compare two states* $s_1 = (t_1. \, l_1, d_1)$ *and* $s_2 = (t_2. \, l_2, d_2)$ *with each other, three equivalence relation will be introduced* $=$, $\overset{!d}{=}$ *and* $\overset{l}{=}$*, which are defined as follows:*

$$\begin{aligned} s_1 = s_2 &\quad \textit{true only if} \quad t_1 = t_2;\ l_1 = l_2;\ d_1 = d_2 \\ s_1 \overset{!d}{=} s_2 &\quad \textit{true only if} \quad t_1 = t_2;\ l_1 = l_2 \\ s_1 \overset{l}{=} s_2 &\quad \textit{true only if} \quad l_1 = l_2 \end{aligned}$$

Definition 16 (Vertex conflict)**.** *Let t be a timestep, $soln_1, soln_2$ solutions of two agents and $soln_1(t) = (t_1.\ l_1, d_1), soln_2(t) = (t_2.\ l_2, d_2)$ the states at the timestep t at each solution. A vertex conflict $vtc = (t, l)$ occurs when holds:*

$$t = t_1 = t_2 \quad \textit{and} \quad l = l_1 = l_2 \qquad \textbf{\textit{or}} \qquad soln_1(t) \overset{!d}{=} soln_2(t) \tag{2.2}$$

Definition 17 (Edge conflict)**.** *Let t be a timestep for which $t \geq 1$ hold and $soln_1, soln_2$ solutions of two agents and $soln_1(t) = (t_{1b}.\ l_{1b}, d_{1b}), soln_1(t-1) = (t_{1a}.\ l_{1a}, d_{1a}), soln_2(t) = (t_{2b}.\ l_{2b}, d_{2b}), soln_2(t-1) = (t_{2a}.\ l_{2a}, d_{2a})$ the states at the timestep t or $t-1$ at each solution. A edge conflict $edc = (t, l_1, l_2)$ occurs when:*

$$l_1 = l_{1b} = l_{2a} \quad \textit{and} \quad l_2 = l_{1a} = l_{2b} \quad \textit{and} \quad t = t_{1b} = t_{2b}$$
$$\textbf{\textit{or}}$$
$$s_1(t) \overset{l}{=} s_2(t-1) \quad \textit{and} \quad s_1(t-1) \overset{l}{=} s_2(t)$$

If there is an conflict, both agents will be constrained with the **respective constraint**, so they will not enter this specific location at this specific time again. Those constraints are, as might expected, the vertex- and edge constraints and get defined as:

Definition 18 (Vertex constraint)**.** *Let t be a timestep, l a location, and agn an agent. The agent agt can be assigned to a vertex constraint $v = (t, l)$ such that it can not enter the location l at time t. Such a constraint can be associated with a vertex conflict $vtc = (t', l')$, where $t = t'$ and $l = l'$.*

Example 6 (Vertex conflict/constraint)**.** *Here we show an example solution of agents $agnt_1, agnt_2$ where a vertex conflict $vtc = (2, (2, 0))$ has occurred. The resulting vertex constraint is $v = (2, (2, 0))$.*

Solution agent a:	Solution agent b:
0. (0, 0), *South* → *Forward*	0. (4, 0), *North* → *Forward*
1. (1, 0), *South* → *Forward*	1. (3, 0), *North* → *Forward*
2. 🟥 (2, 0), *South* → *Forward*	2. 🟥 (2, 0), *North* → *Forward*
3. (3, 0), *South* → *Forward*	3. (1, 0), *North* → *Forward*

Definition 19 (Edge constraint)**.** *Let t be a timestep, l a location, and agt an agent. The agent agt can be assigned to a edge constraint $e = (t, l_1, l_2)$ such that it can not enter the location l_1 and l_2 at time t. Such a constraint can be associated with a edge conflict $edc = (t', l'_1, l'_2)$, where $t = t'$, $l_1 = l'_1$ and $l_2 = l'_2$.*

Example 7 (Edge conflict/constraint)**.** *Here you can see example solutions of agents a, b where a edge conflict $edc = (2, (2, 0), (3, 0))$ has occurred. The resulting edge constraint is $e = (2, (2, 0), (3, 0))$.*

Solution agent a:	Solution agent b:
0. (0, 0), *South* → *Forward*	0. (5, 0), *North* → *Forward*
1. (1, 0), *South* → *Forward*	1. (4, 0), *North* → *Forward*
2. 🟦 (2, 0), *South* → *Forward*	2. 🟥 (3, 0), *North* → *Forward*
3. 🟥 (3, 0), *South* → *Forward*	3. 🟦 (2, 0), *North* → *Forward*
4. (4, 0), *South* → *Forward*	4. (1, 0), *North* → *Forward*
5. (5, 0), *South* → *Forward*	5. (0, 0), *North* → *Forward*

Firstly one agent and then the other is constrained by always creating a new high level node and then inserting them into the conflict tree. Before a new high level node can be created, we first need to calculate the new solution of the constrained agent by paying attention to the currently **updated constrains**. The resulting solution will get then replaced with the old one and the newly generated node gets inserted into to heap to eventually get expanded later on. It can also happen that there is no solution for this set constraint. If so these high level node will not get entered back into the heap. Summarized: In the most cases one high level node with an conflict will result in **two new high level nodes**.

2.3.4 Termination

CBS will terminate when a high level node gets popped out of the conflict tree and this node **does not exhibit any conflicts** in its agents solutions. Since the conflict tree is organised such that the node with the lowest cost is always poped we also got the **optimal solution**. This was the case because every other possible constraining of the agents was more expensive or at most the same cost.

2.3.5 Overall algorithm

Here the **pseudocode of parts** of the high level search is given. Two functions in this pseudocode are unique and never mentioned before therefore it will be briefly explained which function takes which argument and their return values.

<u>FindSolution</u>: This function can take two different parameters. Either an agent *a* can be passed, or an agent *a* and the a list of computed constraints *c*. This is the low level search using the A* algorithm, and the function either returns **false** if there is no possible path from the initial state to the agent's target, or it returns the **actual solution**. If constraints is passed to the function, the low level search computes a solution under **observance of constraints** and returns that solution, or it returns **false**.

<u>GetConfilct</u>: This function takes a high level node and a mutable list of pairs with an agent handle and the computed constraint. When the function is executed, it determines whether two agents of the given node **has a conflict**. If there is a conflict, the calculated conflict of the two agents is then written to the given list as a pair of the respective agent handle and the computed conflict, and finally **false** is returned. Otherwise, if none of the agents had a conflict, then the function returns **true**.

Algorithm 1: Conflict-Based Search (CBS) algorithm

```
input  : Agents informations and obstacles to avoid
output: Agents solutions or false
1  HighLevelNode Start;
2  foreach agent a in Agents do
3      Solution = FindSolution(a);
4      if not Solution then                    /* no Solution -> abort */
5          return false
6      Start += Solution;
7  Heap Open(Start);
8  while Open not empty do                      /* still nodes in Heap */
9      Node = Open.pop();
10     List [agent, constraint] Constraints;
11     if GetConflict (Node, Constraints) then
12         return Node.solution ;          /* no Conflicts -> success */
13     foreach agent a, constraint c in Constraints do
14         Solution = FindSolution(a, Node.a.constrains + c);
15         if Solution then                   /* no Solution -> ignore */
16             HighLevelNode New = Node.copy();
17             New.a.constrains += c;
18             New.a.solution = Solution;
19             Open.push(New);
20 return false ;                          /* unsolvable MAPF instance */
```

2.3.6 Example application of the algorithm

The following section shows an example application of the CBS algorithm in a **very small section** of the FLATLAND competition environment. In Example 2.8 you can see the labeled locations, the labeled agents and even their targets. To keep this simple, some paths of the high level search are **skipped** and not further expanded.

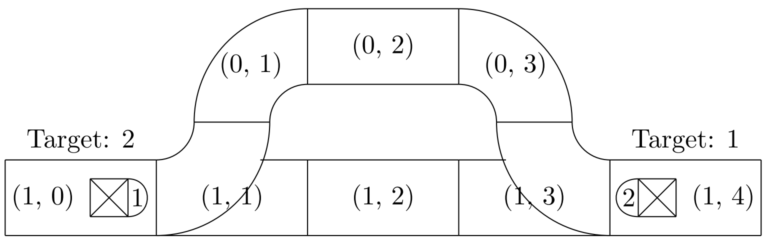


Figure 2.8: Labeled cutout of an environment

Here, the entire conflict tree is shown without the skipped high level nodes. The solution of each iteration marked with (It: timestep) is explained below. The dashed lines should represent extensions that will be skipped:

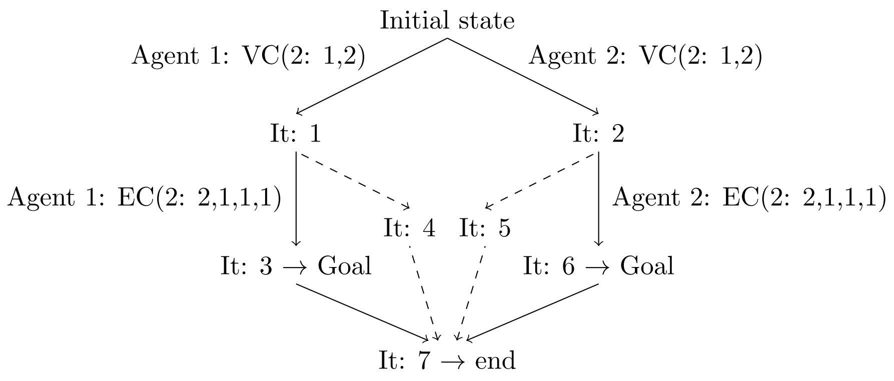


Figure 2.9: Built conflict tree

Initial iteration

Solution Agent 1:

0. (1, 0), *East* → *Forward*
1. (1, 1), *East* → *Forward*
2. (1, 2), *East* → *Forward*
3. (1, 3), *East* → *Forward*
4. (1, 4), *East*

Solution Agent 2:

0. (1, 4), *West* → *Forward*
1. (1, 3), *West* → *Forward*
2. (1, 2), *West* → *Forward*
3. (1, 1), *West* → *Forward*
4. (1, 0), *West*

After this iteration the high level search will detect the vertex conflict at timestep 2 and will investigate by constraining agent 1 and agent 2 with the vertex constraint $(t = 2, y = 1, x = 2)$ and starts the new search on both new high level nodes.

Iteration 1, agent 1

Solution Agent 1:

0. (1, 0), *East* → *Forward*
1. (1, 1), *East* → *Forward*
2. (1, 1), *East* → *Forward*
3. (1, 2), *East* → *Forward*
4. (1, 3), *East* → *Forward*
5. (1, 4), *East*

Solution Agent 2:

0. (1, 4), *West* → *Forward*
1. (1, 3), *West* → *Forward*
2. (1, 2), *West* → *Forward*
3. (1, 1), *West* → *Forward*
4. (1, 0), *West*

Edge conflict at timestep 2 and 3 → both agent will get constrained with the edge constraint $(t = 2,\ y1 = 2,\ x2 = 1,\ y1 = 1,\ x1 = 1)$.

Iteration 2, agent 2

Same as in *iteration* 1 just the solution get exchanged.

Iteration 3, agent 1

Solution Agent 1:

0. (1, 0), *East → Forward*
1. (1, 1), *East → Left*
2. (0, 1), *North → Right*
3. (0, 2), *East → Forward*
4. (0, 3), *East → Right*
5. (1, 3), *South → Left*
6. (1, 4), *East*

Solution Agent 2:

0. (1, 4), *West → Forward*
1. (1, 3), *West → Forward*
2. (1, 2), *West → Forward*
3. (1, 1), *West → Forward*
4. (1, 0), *West*

Both agents has now reached its goal, but the other high-level nodes with lower costs still need to be considered.

Iteration 4, agent 2 / Iteration 5, agent 1

Skipped.

Iteration 6, agent 2

Same as in *iteration* 3 just the solution get exchanged.

Iteration 7, agent 1

Now one of the previous high level nodes can be picked. All of them has the **same value** and the conflicting nodes from *iteration* 4 and *iteration* 5 can also be selected. If so we must expand this two further before **eventually picking** a high level node from *iteration* 3 or *iteration* 6 and terminating the algorithm. Those two nodes have already reached the goal with a solution which does not interfere each other.

2.3.7 Conclusion in relation to the Flatland competition

It was shown that by applying the algorithm to this MAPF instance, a valid solution is obtained. So **this algorithm** can also be used for the FLATLAND competition and it always get the best possible solution?

Answer: You will get **someday** the best possible solution for a given FLATLAND competition environment!

One **big disadvantage** of the standard CBS algorithm is that it is not capable with the structure of the given environment, because it can not really deal with the fact that there is no action *Back*. In the environment there are a lot of rails which an agent can only apply the *Forward* action. So the agent can only decide at **specific locations** and **directions** where it wants to head next. However, the algorithm considers all possible locations although there is no possible solution for this state, which leads to **unnecessary expansions** of high level nodes. This can already been seen at the previous Example 2.3.6, because we have expanded at least $7 = 2^3 - 1$ high level nodes for such a small environment. When you consider Figure 2.6 you will notice that there are a lot of rails which have no other possible action than applying *Forward* which leads, as described, to a lot of **avoidable node expansions**.

Considering the similar Environment 2.10, where we just **added two rails**, we need to set at least three constraints to reach the solution of this environment. This means that the Conflict Tree 2.9 must be extended by one level, which then leads to an exponential increase of the expanded high level nodes. At least $15 = 2^4 - 1$ nodes must now be expanded just to achieve this simple environment.

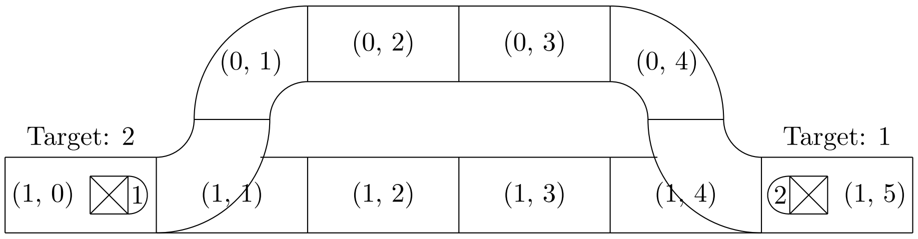


Figure 2.10: Bigger cutout of a environment

Conclusion: The standard CBS algorithm is **not able** to handle environments from the FLATLAND competitionin an allowable computation time.

Chapter 3

Solving the Flatland Challenge

3.1 Motivation

As seen in Environment 2.8 and Environment 2.10, the standard CBS algorithm will expand a lot of unnecessary nodes. For example, in Environment 2.10 those nodes will be expanded because of the rails types at the locations $(1, 2)$ and $(1, 3)$. At these locations, the only applicable actions are *Forward* and *Wait*, regardless of which direction an agent is facing. It follows that an agent **cannot choose** another solution when it is in a location with these properties. When a particular state has such a locations in an agent's solution, it is considered a **non-critical state**. Which can be defined as follows:

Definition 20 (Non-critical state)**.** *Let $s = (t.\ l, d)$ be a state and $l(d)$ be the number of applicable actions in addition to the Wait action at location l while facing direction d. A applicable critical state crit exists if $l(d) < 2$.*

Example 8. *Consider state $s = (t.\ l, d)$, where $l = (1, 4)$ and $d =$ East, in the example Environment 2.10. It holds $l(d) < 2$, i.e., the state s is no-critical state.*

Therefore, the **opposite term** also exists, where an agent can apply more actions the trivial actions *Wait* and *Forward*, which is called a applicable critical state. Note that only the state is affected by the term and at a location l, of a applicable critical state $crit = (t.\,l, d)$ a non-critical state can still exist if the appropriate direction d is chosen.

Definition 21 (Applicable critical state)**.** *Let $s = (t.\ l, d)$ be a state and $l(d)$ be the number of applicable actions in addition to the Wait action at location l while facing direction d. A applicable critical state crit exists if $l(d) = 2$.*

Example 9. *Consider state $s = (t.\ l, d)$, where $l = (1, 1)$ and $d =$ East, in the example Environment 2.10. It holds $l(d) = 2$, i.e., the state s is a applicable critical state.*

Thus, when an agent passes such a critical state, it is **forced** to take a certain solution path without deciding where to go next until it reaches another critical state. Such a set of **non-critical states** in a solution must be eliminated somehow, otherwise the algorithm will expand to many nodes in which it cannot solve the given conflict. Such a set is also called passage and is defined as:

Definition 22 (Passage)**.** *Let agt be an agent, soln its corresponding solution, and $soln(t)$ a applicable critical state. t is incremented to t' until the first occurrence of a state $soln(t') = (t'.l', d')$, where $soln(t') = 2$ and $t' > t$ for any possible direction d. The unique locations of the states between these critical states are referred to as a passage.*

If a conflict between two agents occurs in a passage, this can lead to an **exponential blowup** in the CBS algorithm in terms of passage size, even if this conflict can be resolved by expanding two high level nodes. For this reason, these passages should be eliminated in order to reduce the number of active branches in the conflict tree.

3.2 Idea

Because of this possible exponential blowup, the following approach must **block the entry** of such passages when a conflict has occurred in one, and not expand high level nodes that cannot be resolved at all. To block such a passage, the location after a critical state can be constrained for a certain interval of timesteps. The low level search can then consider another action in the location of the critical state, which then leads to a **different solution**, and the conflict between the agents is resolved. Such an constrain with an interval of timesteps can be defined as follows:

Definition 23 (Multi time constraint)**.** *Let $start, end$ be timesteps where $start \leq end$ and l a location. A multi time constraint mtc can be attached to an agent agt, such that it cannot enter the given location l in the interval $[start, end]$, so:*

$$mtc = [s, e]\ l$$

This approach uses different **rules** based on the type of critical state and the type of conflict. In each case, there are different ways to constrain the particular agent. So, if an agent is constrained in an environment at a location for a certain timestep interval, it will be indicated with a yellow box and might look like the following. Here, the agent is constrained at the respective location for the interval $[3 - 10]$:

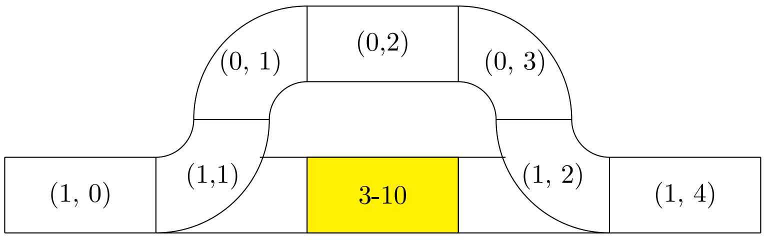


Figure 3.1: Illustration of constraint in an environment

To illustrate the application of the rules, a general representation of the solution of two conflicting agents is considered. This representation uses timesteps at specific locations to indicate that an agent is there at that timestep. **Note**: These illustrations shares the **same locations** and only for better readability of the timesteps two different representations are used one below the other.

Agent 1:	8	7	6	5	2-4	1	0
Agent 2:	0	1-3	4	5	6		

Figure 3.2: Illustration of solutions from two agents

In this figure, you can see the solution of two agents that initially starts at timestep 0. Agent 1 stays in one location for the timestep interval $[2-4]$ and then moves on, finally reaching its **target** at timestep 8. Agent 2 stays in one location for the timestep interval $[1-3]$ and then reaches its target at timestep 6. To bring additional information into this representation, **critical states** are also introduced. They are translated in the following way:

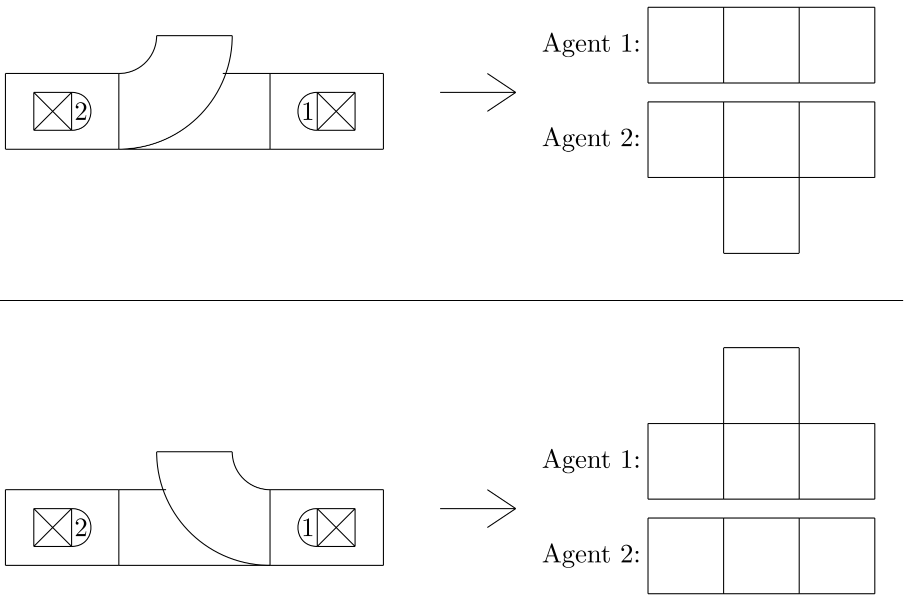


Figure 3.3: Translation of critical state into simple notation

A visual indicator for **edge** and **vertex conflicts** in the solutions is also needed. This could look like the following:

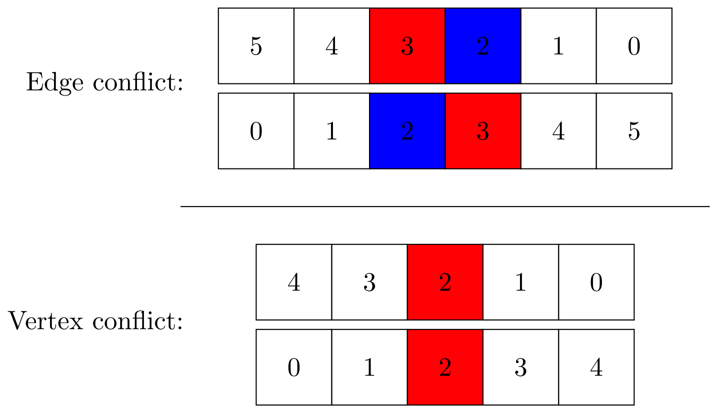


Figure 3.4: Illustration of simple edge and vertex conflict

In advance: there are six different types of critical states that can be detected. For each of these types, a specific rule must be applied. Since there are always two affected agents in a conflict, only one half of the conflict resolution is considered in this paper, although the other half may apply the same rules defined for one half. The edge conflict is only useful to determine if a conflict has occurred between two agents, then the conflict is also **converted** to a vertex conflict, observing the state at the previous timestep. The transformation based on the Figure 3.4 to the final illustration then looks like the following. In the illustration the solutions is halved and the edge conflict is converted to a vertex conflict, keeping the timestep the same. Later, those timesteps will remain the same because the rules are robust enough to handle it this way.

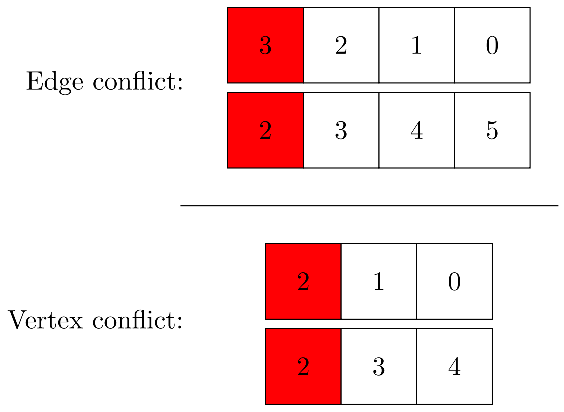


Figure 3.5: Transformation of simple edge and vertex conflict

3.3 Types of critical states

As previously stated, there are **six different** types of critical states that affect how an agent is constrained. These critical states are defined by the number of actions an agent can perform in addition to the *Wait* action at a given location and in a given direction. In addition to the applicable critical state defined in *Def.* 21 and the non-critical state defined in *Def.* 20, there is also an **inapplicable critical state**. Such a state can potentially be found when two agents collide at timestep t, which is then incremented until a corresponding state is found in agent 2's solution, i.e., $s_2 = (t_2.\, l_2, d_2)$, where $l_2(d_2) = 2$ and $t_2 \geq t$, making it an applicable state for agent 2 and an inapplicable state for agent 1. This means that agent 2 can use this branch if agent 1 would have no conflict with it. Simply put, this state cannot be used for another solution for an agent, but the conflicting agent can use it for its **own purposes**. Here is a small example of what these types of critical states might look like, where at $(0, 2)$ you can see an applicable critical state for agent 2 and an inapplicable critical state for agent 1:

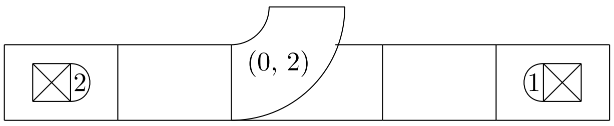


Figure 3.6: Example of an applicable or inapplicable critical state

These states also **differ** with respect to the location, i.e. , such a state can be further **distinguished** by dividing them into initial, regular, and target applicable / inapplicable states. The properties of these states in an agent solution are defined as follows:

1. Initial: The state with the initial location of the agent, with the highest time value
2. Target: The state with the target location of the agent
3. Regular: Any matching state without such locations

3.4 Related work

Finally, after all the needed terms have been defined, a comparison can be made to the related work mentioned in the introduction of the paper *Pairwise Symmetry Reasoning for Multi-Agent Path Finding Search* [Li+21b]. As stated, the idea and the following approach were found independently of the cited work, but there are some similarities and modifications that will now be pointed out. The main idea on which the algorithm is based is to find the redundant **passages** *Def.* 22, or **corridors** *Def.* 9, as it is called in the comparison work, and then constrain the respective agent to prevent it from entering such a segment of the environment. This is also done by placing a **multi time constraint** *Def.* 23, or **range constraint** [Atz+18], as it is called in the comparison work, at a specific location. The difference between the approaches is that the compared work only supports states where there is no fixed direction the agent must face.

This means that the agent can apply any action in any cardinal direction, so the action *Back* is added to the set of applicable actions. Moreover, the additions from the competition are not taken into account and there is no named solution that can solve an instance when there are agents with different **speed values** or there is a **malfunction** while executing the computed solution. Finally, there is no way that agents cannot be placed directly on the grid and held back as long as they want.

3.5 Basic algorithm

3.5.1 Initialisation

The following algorithm is an extension of the CBS algorithm mentioned in Chapter 2.3. Thus, the defined search strategies and the corresponding nodes are used in this approach and only the differences from the original algorithm are discussed. After the solver has extracted all the necessary information, such as initial states, targets, or heuristics, a **low level search** is started on each agent. In this search, the corresponding solution is returned for each agent. In the FLATLAND competition it is guaranteed that each agent has a valid solution from its initial state to its destination. These solutions are then passed into the initial **high level node** and the high level search starts by detecting the conflicts of the agents in their just computed solutions.

3.5.2 Detecting conflicts

To detect agent conflicts, they must be **compared pairwise** at each timestep and check whether there is a vertex or edge conflict. This can be done by extracting the maximum timestep of each solution and then using the conflict definition (vertex conflict *Def.* 16, edge conflict *Def.* 17). It should be noted that this pairwise search starts with timestep 1 and not 0. The reason for this is that the environment guarantees that no agents are at the same initial location. The evaluation of an agent's solution can be skipped if the current timestep is larger than the size of the corresponding solution, since the agent has already been removed from the grid in this timestep. If there is no conflict between every agent, a **valid solution** has been found for the given FLATLAND competition environment. Otherwise, the conflict found is returned. Such an algorithm may look like the following:

Algorithm 2: Detect a conflict of agents in a high level node

```
input  : High level node hln
output: Conflict or true
/* Get maximal timestep of all solutions                    */
1 maxTime = 0;
2 foreach solution soln in hln do
3     if maxTime < soln.size then maxTime = soln.size;
```

```
4  for $t \leftarrow 1$ to $maxTime$ do
5      for $handle1 \leftarrow 0$ to $hln.agentCount$ do
6          if $t >= hln.solution[handle1].size$ then continue;
7          for $handle2 \leftarrow handle1 + 1$ to $hln.agentCount$ do
8              if $t >= hln.solution[handle2].size$ then continue;
9              state1a = hln.solution[handle1].at(t-1);
10             state1b = hln.solution[handle1].at(t);
11             state2a = hln.solution[handle2].at(t-1);
12             state2b = hln.solution[handle2].at(t);
               /* Check if vertex conflict */
13             if $state1b \overset{!d}{=} state2b$ then
14                 return $vtc = (t, state1b.location)$;
               /* Check if edge conflict */
15             if $state1b \overset{l}{=} state2a$ and $state1a \overset{l}{=} state2b$ then
16                 return $edc = (t, state1b.location, state2b.location)$;
17 return true ;                      /* no conflict has been found */
```

3.5.3 Detecting critical states

The idea is to constrain the critical states of the conflicting agents to prevent them from entering a particular passage. Thus, these critical states must be identified only after a conflict between two agents has occurred. These agents each have a solution *soln* where the **particular states** are examined according to the current time value. This time value starts with the timestep of the conflict and is decreased until the respective state $s = soln(\text{time value})$ is a critical state. A critical state may not be found and the time value may eventually fall to -1. If so, there is no critical state in the solution and a **false** is returned, otherwise the critical state found. Such an (now incomplete) algorithm may look like the following:

Algorithm 3: Extracting critical state from an agent (incomplete)

```
input  : Solution agent $soln$, Conflict
output: Critical state or false
1 timeValue = Conflict.timestep;
2 while $timeValue \geq 0$ do
3     state = $soln$(timeValue);
4     if $state.location(state.direction)$ is 2 then
5         return $state$ ;                /* return critical state */
6     timeValue = timeValue − 1;
7 return false ;                      /* no critical state was found */
```

1. Finding the applicable critical state in a agent's solution:

In the following cutout from an incomplete Environment 3.7, one agent can be seen starting at location $(1, 3)$. The other agent is not visible in this cutout, but a vertex conflict has occurred between these agents at location $(1, 0)$. In this particular environment with this given solution, the applicable critical state at location $(1, 2)$ is found by applying the previous algorithm with the solution of the visible agent and the vertex conflict. It should be noted that the first occurrence of a critical state is always represented with a green box. Here an solution and its corresponding environment is illustrated:

Solution agent:

0. (1, 3), *West* → *Forward*
1. (1, 2), *West* → *Forward*
2. (1, 1), *West* → *Forward*
3. (1, 0), *West* → *Forward*
4. ...

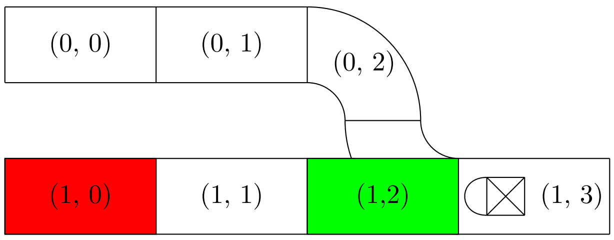


Figure 3.7: Finding critical state after conflict

Here you can see the **simpler version** of the solution in Figure 3.7:

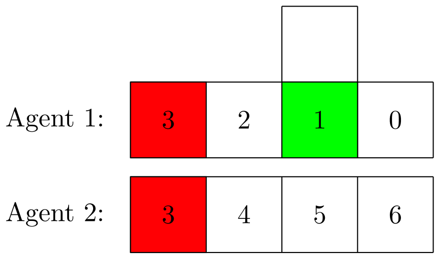


Figure 3.8: Simpler illustration, finding critical state

2. No critical state in agent's solution:

In the following cutout from an incomplete Environment 3.9, the same vertex conflict has occurred between the visible and non-visible agents. The difference between the environments is that there is no critical state in the solution of the visible agent. The algorithm will then return **false**.

Solution agent:

0. (1, 3), *West* → *Forward*
1. (1, 2), *West* → *Forward*
2. (1, 1), *West* → *Forward*
3. (1, 0), *West* → *Forward*
4. ...

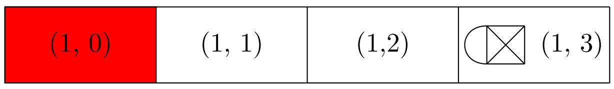


Figure 3.9: No critical state after conflict

Here you can see again the matching **simpler environment** with the made up solution from agent 2:

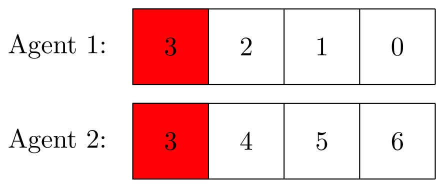


Figure 3.10: Simpler illustration, no critical state

3. Finding the inapplicable critical state in a agent's solution:

At this point, there is only **one type** of environment which has not been discussed yet. This type cannot be handled by the current implementation of the algorithm because it **does not recognize** the first occurrence of the inapplicable critical state in the particular solution and will return false. In the following cutout, you see the same vertex conflict as before, but the inapplicable critical state at location $(1, 2)$ cannot be identified. Here you can see the failing problem instance:

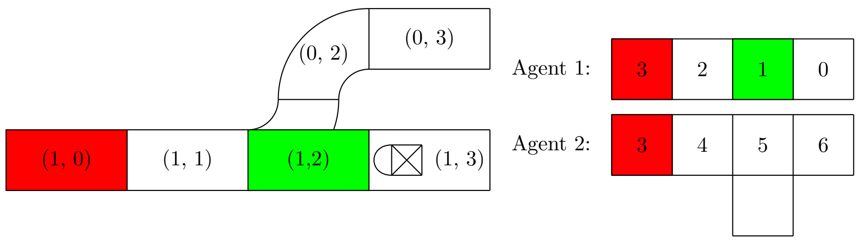


Figure 3.11: Inapplicable critical state after conflict & simpler illustration

To identify such an inapplicable critical state, the solution of the other conflicting agent must be examined by the algorithm. This algorithm then takes the timestep of the conflict and its corresponding state s_1 in the solution of agent 1. It also takes the corresponding state s_2 in the solution of agent 2, but if it is an edge conflict, the previous timestep is taken into account so that both states have the same location. Then the function GetPrevState is applied to s_1 and the function GetNextState is applied to s_2. The returned states are determined if any of the critical state **types match**. If so, this state is returned, otherwise they are stored as new s_1 and s_2 and the function is executed again. This algorithm **terminates** because one of the critical states must occur, in the worst case a type of the initial critical states.

Function, GetNextState:

This function takes a state $s = (t.\ l, d)$ and a solution *soln* and **increments** the value t until a state $soln(t') = s' = (t'.\ l', d')$ is reached where $l \neq l'$ and $t' > t$. If so, this state s' is returned, otherwise **false**.

Algorithm 4: Function GetNextState

```
input  : Solution agent soln, State $s = (t.\ l, d)$
output: Next state or false
1 for $t' \leftarrow t + 1$ to soln.size do
2     if $soln(t') = (t'.\ l', d')$ applies $l \neq l'$ then
3         return soln(t') ;            /* next state has been found */
4 return false ;                   /* was already the last state */
```

Function, GetPrevState:

This function takes a state $s = (t.\ l, d)$ and a solution *soln* and **decrements** the value t until a state $soln(t') = s' = (t'.\ l', d')$ is reached where $l \neq l'$ and $t' < t$. If so, this state s' is returned, otherwise **false**.

Algorithm 5: Function GetPrevState

```
input  : Solution agent soln, State $s = (t.\ l, d)$
output: Previous state or false
1 for $t' \leftarrow soln.size - 1$ to 0 do
2     if $soln(t') = (t'.\ l', d')$ applies $l \neq l'$ then
3         return soln(t') ;        /* previous state has been found */
4 return false ;                  /* was already the first state */
```

Final algorithm to extract all types of critical states:

Algorithm 6: Extracting critical state from an agent (complete)

```
input  : Solution agent 1 $soln_1$, Solution agent 2 $soln_2$, Conflict
output: Tuple (bool, critical state)
1 stateOne = $soln_1$(Conflict.timestep);
2 if Conflict is Edge conflict then
3     stateTwo = $soln_2$(Conflict.timestep $- 1$);
4 else
5     stateTwo = $soln_2$(Conflict.timestep);
```

```
while true do
    /* If true the initial state has been reached */
    if $stateOne = false$ then
        if $stateOne.location(stateOne.direction)$ is 2 then
            return (true, stateOne) ;  /* applicable initial */
        return (false, stateOne) ;  /* inapplicable initial */
    /* If true a target has been reached */
    if $stateTwo = soln_2.back()$ then
        if $stateOne.location(stateOne.direction)$ is 2 then
            return (true, stateOne) ;  /* applicable target */
        return (false, stateOne) ;  /* inapplicable target */
    if $stateOne.location(stateOne.direction)$ is 2 then
        return (true, stateOne) ;  /* applicable */
    if $stateTwo.location(stateTwo.direction)$ is 2 then
        return (false, stateOne) ;  /* inapplicable */
    stateOne = GetPrevState (stateOne, $soln_1$);
    stateTwo = GetNextState (stateTwo, $soln_2$);
```

3.5.4 Standard collision

The idea of how to constrain the conflicting agents is to examine their solutions and set the lower and upper bounds on the resulting multi time constraint based on the timesteps of certain states in their **current solutions**. To do this, first determine the critical state and its type from both solutions. Based on the returned state and type, the remaining solution of agent 2 is considered and the function GetNextState is called until a certain scenario occurs. The solver may not call this function at all and may set the boundary with the current information. It should be noted that agent 1 and agent 2 are only **placeholder** for the particular agents where a conflict has occurred. For some types of returned states, there are some **special cases** that need to be considered. In addition to the special cases explained previously, there are also some terms that need to be defined first. One of these terms is the timestep at which an agent **leaves a location** corresponding to a state with the same location:

Definition 24 (Timestep agent leaves a location)**.** *Let agt be an agent, soln its associated solution, and $s = (t. l, d)$ a state in soln. The timestep at which agt leaves a location l is defined by applying* GetNextState *and subtracting* 1 *from the timestep of the resulting state.*

There is also the reverse notion, where the timestep of an agent **entering** a particular location is required. This term will get defined as follows:

Definition 25 (Timestep agent enters a location)**.** *Let agt be an agent, soln its associated solution, and $s = (t. l, d)$ a state in soln. The timestep at which agt enters a location l is defined by applying* GetPrevState *and adding* 1 *to the timestep of the resulting state.*

The states whose locations are the same, but which are in each other's solution, are also of need. Thus, the location of one state may correspond to the location of the other state. Such a **corresponding state** is the first occurring state, after incrementing the timestep of the conflict time, until the locations of these states are equal. Such a corresponding state is defined as:

Definition 26 (Corresponding state)**.** *Let t be the timestep of a conflict, $s_1 = (t_1.\ l_1, d_1)$ a state lying in $soln_1$ and $s_2 = (t_2.\ l_2, d_2)$ a state lying in $soln_2$ for which $t \geq t_1$, $t \geq t_2$ and $l_1 = l_2$ hold. s_2 corresponds to s_1 if the timestep t is incremented to t_2 and during this increment $soln_2(t)$ does not yield any other state with l_2.*

1. Applicable critical state

If the algorithm of the previous chapter returns an applicable critical state $crit_1 = (t.\ l, d)$, then the corresponding state s_2 of the solution of agent 2 is considered. Then the mentioned function GetNextState can be applied at least once to s_2, where the solver applies this function and also counts the applied times until this function returns a state with one of the following scenario:

1. Applicable critical state
2. Same location as the initial state of agent 1
3. Target location of agent 2

For further illustration, some sample environments are used and how they will get constrained based on the following rules. Starting with the **first scenario** and a matching environment:

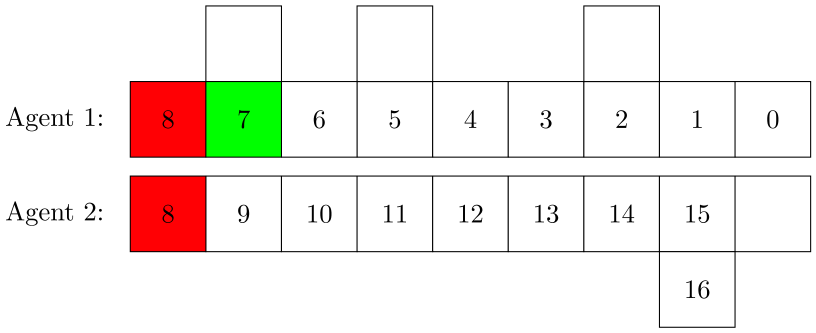


Figure 3.12: Environment: applicable critical state (1)

To determine the **lower bound** of the multi time constraint, the timestep at which agent 1 would enter the critical state location is taken and the value 1 is added. In this example, this would be the value $7 + 1 = 8$. To compute the **upper bound**, the function GetNextState is applied to the corresponding state of $crit_1$, which is s_2. In this example, the state corresponds to the state in solution $soln_2$ with timestep 9. The application of this function continues until an applicable critical state is returned, i.e. , a path by which agent 2 could escape if agent 1 does not block its path. Then, the timestep in which agent 2 leaves the location of the returned state is added to the number of times the function needs to be applied and set as an upper bound. In this example, the timestep in which agent 2 leaves the location of state s_2 is 15. To reach the returned state, the function must be applied 6 times, starting from s_2. Therefore, the upper bound is $15 + 6 = 21$ and agent 1 is constrained at the critical state location and the location thereafter with these calculated bounds. Here the visualization of the constrained environment is shown:

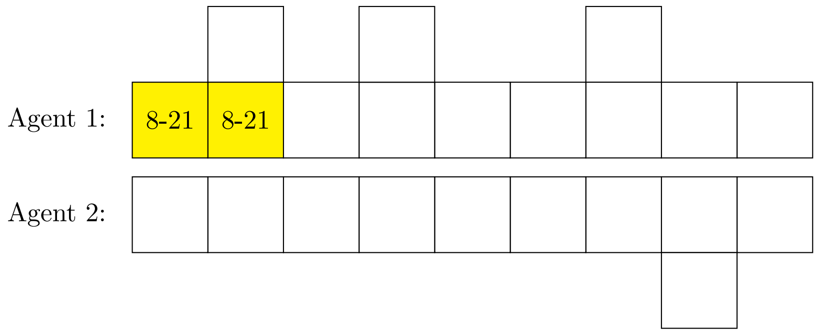


Figure 3.13: Constrained: applicable critical state (1)

note: The reason for constraining the location after the critical state and the location of the critical state itself is to prevent the expansion of unnecessary nodes. This requires agent 1 to leave the location of the critical state as soon as possible unless it is more efficient to wait before the location of $crit_1$. If it is more efficient to wait, agent 1 would later cause a vertex conflict. This resulting high level node would then contain a constraint in which agent 1 is forced to wait before the location where agent 2 could exit the passage, which is the last state returned by the GetNextState function. This is then an inapplicable critical state, the handling of which is explained later.

The **second scenario** occurs when an applicable critical state is **not** found in the solution form agent 2 and therefore a state is detected that has the same location as the initial state from agent 1. Such an scenario can be seen in the following cutout environment:

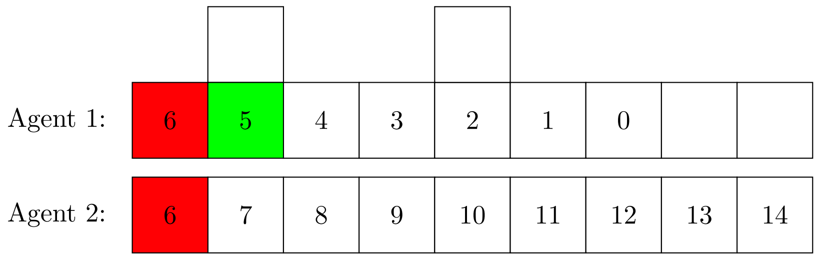


Figure 3.14: Environment: applicable critical state (2)

To resolve this scenario, agent 1 must take a different route than the one through the current passage where the conflict occurred. The first idea that comes to mind is to constrain the critical state for an **infinite amount** of time, forcing agent 1 to take a different route. The problem with this idea is that there are some environments that have a valid solution, but the conflicting agent has to go through the passage, which is then blocked for an infinite amount of time. The solver will thus create an high level node that **cannot lead** to a valid result, even if there is one. This means that the idea must be discarded and a more expensive approach is considered. Below you can see an environment that shows such a problem:

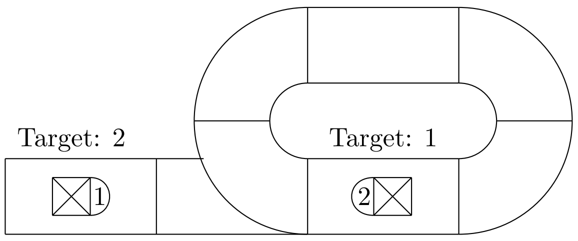


Figure 3.15: Counterexample: using constraint with infinite amount of time

In the shown Environment 3.15 there **exists** a solution. This is for agent 2 to take the outer loop and allow agent 1 to pass into its target, and then be removed. After removal, it is possible for agent 2 to get into its target by taking the original path, which is now **unblocked**. To solve such types of environments, a local search can be applied where the initial state is the first state of the other possible route and the target location remains the same. The resulting number of states, including the local initial and target states, is then taken and added to the lower bound. Speaking of which, the **lower bound** is still the same as in the first scenario. The value of this addition is then set as the **upper bound**. After calculating the limits, the location at the critical state and the location thereafter is constrained. Note: It can be seen that with this method it is possible to solve the environment shown in Figure 3.15. Here is an illustration of how you can constrain such an environment by declaring V to be the value that was just calculated by the local search:

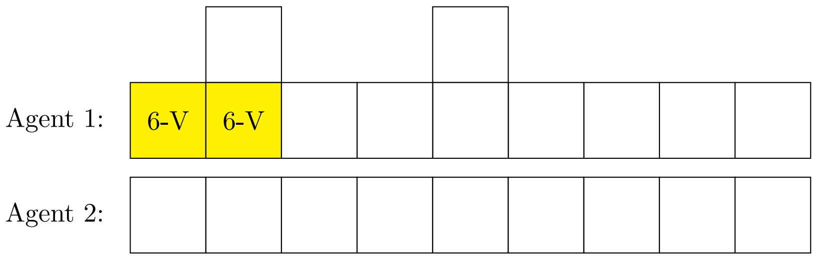


Figure 3.16: Constrained: applicable critical state (2)

Finally, the **last scenario** is discussed. This can occur when agent 2's solution ends in its **target** without first discovering other applicable critical states or a state with the same location as agent 1's initial location. Here you can see such an example environment:

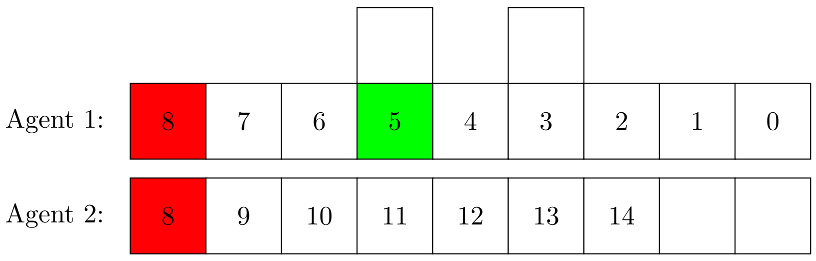


Figure 3.17: Environment: applicable critical state (3)

To solve this scenario, the **lower bound** is set in the same way as in the previous two scenarios. In this example the lower bound will be the value 6. To obtain the **upper bound**, the same method as in the first scenario is used, except that instead of an applicable critical state, the target is found in the solution of agent 2. Likewise, we use the timestep of this target state and add it to the number of times the function GETNEXTSTATE was applied. To apply this rule to the **present environment**: The timestep of the target state is 14 and the function has been called 3 times. So the final value of the upper bound is $14 + 3 = 17$. The locations of the constraints are also **the same** as in the other scenarios, so they are not shown further.

2. Inapplicable critical state

After an inapplicable critical state $crit_1 = (t.\ l, d)$ is returned, the corresponding state s_2 is also considered. The idea is that agent 1 waits at the previous location of $crit_1$ for as long as it takes agent 2 to move from the conflict to the next location of s_2. So the solver must set the **lower bound** to the timestep in which agent 1 would enter the location of $crit_1$. The **upper bound** will be the timestep in which agent 2 would leave the location of s_2. The constrained location will be the location of the inapplicable critical state.

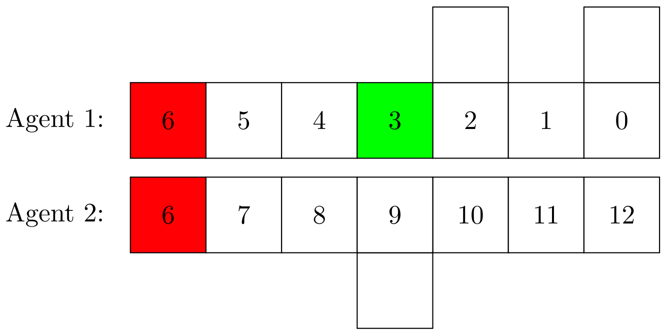


Figure 3.18: Environment: inapplicable critical state

In this environment, agent 1 would enter the location of the inapplicable critical state at timestep 3 and agent 2 would leave this location at timestep 9. Thus, the resulting multi time constraint has the value 3 as **lower bound** and the value 9 as **upper bound**. Here you can see how the environment is constrained based on these values:

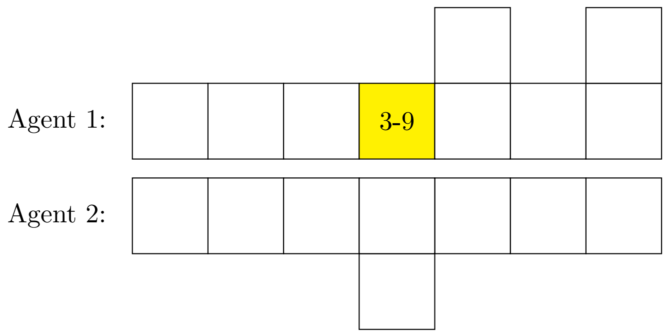


Figure 3.19: Constrained: inapplicable critical state

3. Target inapplicable critical state

If the algorithm returns this type of critical state, the best way to resolve this conflict is to have agent 1 wait until agent 2 **reaches its target** and is removed from the grid. Then agent 1 is able to continue without coming into conflict with agent 2 again. This can be achieved by constraining the location of the target for as long as it takes agent 2 to get to the target. Therefore the **upper bound** will be the timestep of the **last state** in the solution of agent 2. The **lower bound** will be the timestep at which agent 1 would enter the location of the target. Here you can see a cutout of an environment corresponding to this case and how it is constrained:

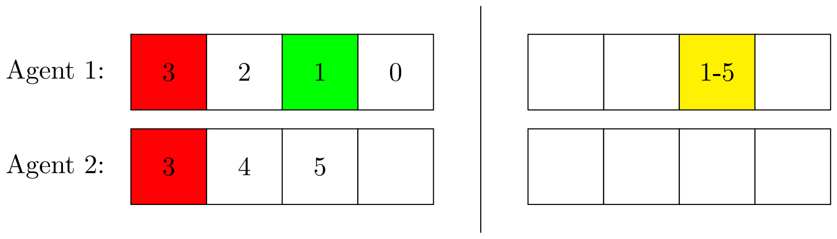


Figure 3.20: Environment & constrained: target inapplicable critical state

4. Target applicable critical state

When this type of critical state is returned, there are even more options for agent 1 to take so that the conflict can be resolved and it does not have to wait at the location in front of the target until the conflicting agent 2 reaches it. To constrain the agent, the solver applies the same strategy as in the applicable case before. First, the **upper bound** is set to the timestep of the last state of the solution of agent 2. Then, the **lower bound** is set to the timestep at which agent 1 would enter the target location of agent 2. Here you can see a cutout of an environment corresponding to this case and how it is constrained:

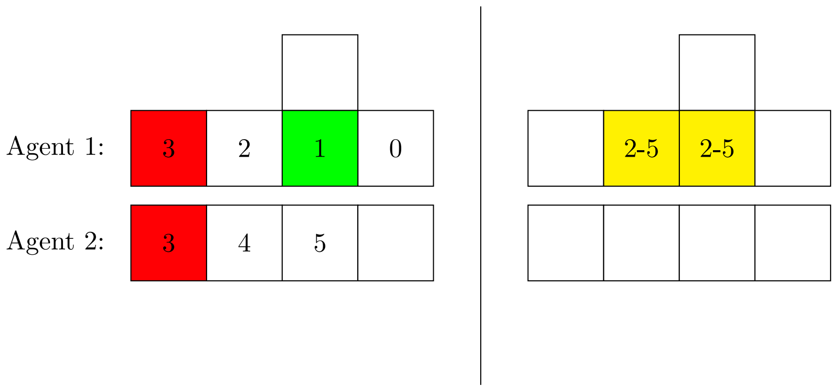


Figure 3.21: Environment & constrained: target applicable critical state

5. Initial applicable critical state

If this case occurs, there may still be a possible solution for agent 1 by taking the other branch provided by the applicable critical state. For the agent to take this path, the solver must constrain the agent so that the low level search takes this other path. As in the case *1. Applicable critical state*, the solver must apply a **local search** starting with the first state after the found critical state that is not in the passage where the conflict occurred and ending at the same target location. Here you can see a cutout of an environment corresponding to this case and how it is constrained when V is the computed upper bound:

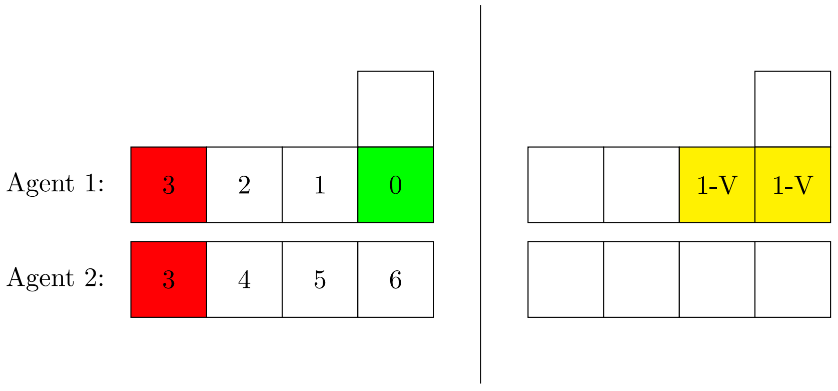


Figure 3.22: Environment & constrained: initial applicable critical state

6. Initial inapplicable critical state

This is the case when the algorithm returns a state s_2 with the same location as the initial state of agent 1. When this happens, there is no other path for agent 1 or agent 2 to take in this half of the passage. Thus, to solve this conflict, agent 1 is not placed directly on the grid, but must wait until agent 2 has passed the initial location. The **upper limit** is then set to the timestep with which agent 2 would leave the returned state. The **lower bound** is 0 to prevent agent 1 to get placed to early. The resulting multi time constrain is then placed at the initial location. **Note** that the low level search must be **capable** when the initial state is constrained and the agent is not placed directly on the grid. Here you can see a sample environment that corresponds to this scenario:

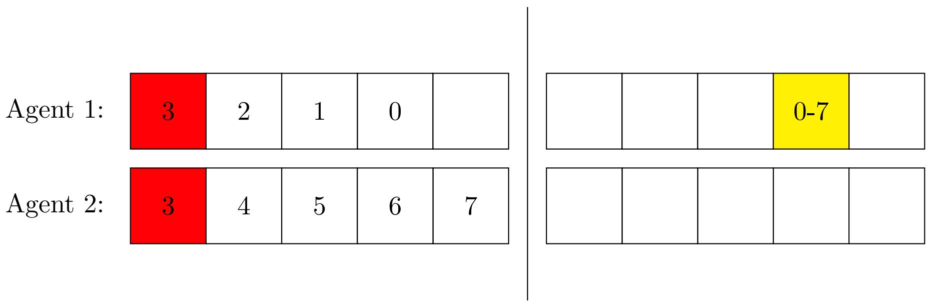


Figure 3.23: Environment & constrained: initial inapplicable critical state

Note: The corresponding state s_2 can also include the target location of agent 2, because the algorithm will first return the initial inapplicable critical state of agent 1. Therefore, the function GETNEXTSTATE returns **false**. If this is the case, the timestep of the last state of the solution of agent 2 is taken as **upper bound**.

3.5.5 Special rule in applicable critical state

This special case occurs when the found applicable critical state and the conflict have the same location. As always, the goal of the solver is to reschedule agent 1 to the other possible route of the critical state. However, if the conflict and the critical state are at the same location, agent 2 will still get to that location at the timestep of the conflict and agent 1 will not be able to use the other path. So agent 2 has to wait for an **additional timestep** before reaching the location of the critical state. This can be done by constraining agent 2 at the location of the conflict for one timestep. This is the **only addition** to the particular case in which there is an applicable critical state. Here you can see a corresponding environment and how the solver constrains the particular agents at the multiple locations:

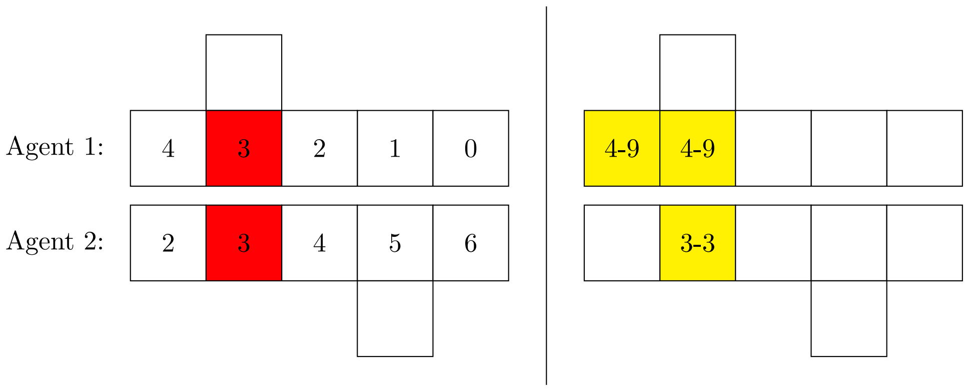


Figure 3.24: Special environment and how to constrain the agents

There is just one **problem** that affects the optimality of the solver. This problem occurs when agent 1 is unable to take the other route because, for example, there is another conflict with an agent. Thus, in any successor of a high level node where the solver has constrained an agent like agent 2, this **delay** remains the same even if it is may not needed.

3.5.6 Rail type corner rule

At a particular location l, a special vertex conflict $vtc = (t, l)$ may occur if that location l has a particular rail type of type 6 in Figure 2.1. If the states of the two agents leading to this conflict can have parallel directions, the algorithm above may **fail** in detecting the critical state. That is, if there are solutions $soln_1, soln_2$ of the two conflicting agents, the corresponding states $s_1 = (t.\ l_1, d_1) = soln_1(t)$ and $s_2 = (t.\ l_2, d_2) = soln_2(t)$ can show that their direction is either the *North-South* or *East-West* parity, and the order does not matter. Here is a small corresponding environment to indicate this case:

Solution agent 1:

0. (2, 1), *North* $\rightarrow$ *Forward*
1. (1, 1), *North* $\rightarrow$ *Left*
2. (1, 0), *West*

Solution agent 2:

0. (0, 1), *South* $\rightarrow$ *Forward*
1. (1, 1), *South* $\rightarrow$ *Right*
2. (1, 0), *West*

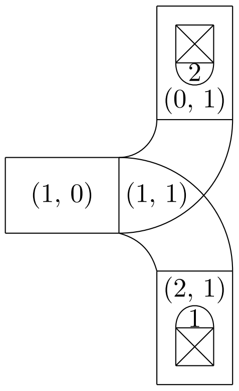


Figure 3.25: Sample solution & environment of rail corner case

It can be seen that at timestep $t = 1$ a vertex conflict has occurred and the resulting directions of the states leading to the conflict are *North* and *South*. Therefore, the **lower bound** is set to the timestep that agent 1 would use to enter the critical state location, and **upper bound** is set to the timestep it takes agent 2 to leave that location. The location of the conflict will then be constrained.

3.5.7 Rear-end collision case

The solver should be able to determine if an agent has bumped up another agent that has been waiting at a particular location, and then constrain the conflicting agent for as long as the other agent has to wait. This can be done by examining the directions of the state that led to the **vertex conflict**. If the directions of those states match, one agent has bumped into the other. Then the solver has to find the agent who was waiting and how long the agent still has to wait at that location. Here you can see an matching environment:

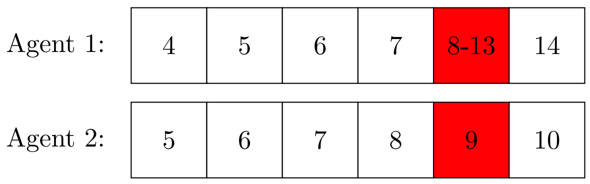


Figure 3.26: Environment where agent 2 bumped into agent 1

First of all, it is necessary to identify the agent who was **already waiting** at the conflict location. To do this, take the timestep of the conflict t and compare the location of the state with timestep t and the location of the state from the previous timestep $t-1$. If in a solution this equation is true, the agent belonging to the solution has already been at that location, so it is waiting. It is to be noted that **only one** of the two equations can be true:

$$soln_1(t-1) \overset{l}{=} soln_1(t) \implies \text{if } \textbf{true} \text{ agent 1 is waiting}$$
$$soln_2(t-1) \overset{l}{=} soln_2(t) \implies \text{if } \textbf{true} \text{ agent 2 is waiting}$$

In the case of the example cutout environment, these equations would look like this, given that the timestep of the conflict is $t = 9$:

$$soln_1(9-1) \overset{l}{=} soln_1(9) \implies \text{Agent 1 } \textbf{is} \text{ waiting}$$
$$soln_2(9-1) \overset{l}{\neq} soln_2(9) \implies \text{Agent 2 } \textbf{is not} \text{ waiting}$$

So it must be prevented that agent 2 enters the location of conflict. This can be done by setting the **lower bound** to the timestep with which agent 2 would enter this location, in this case to the value 9. The **upper limit** is set to the timestep at which agent 1 would leave this location, i.e. , the value 13. This then ultimately results in the following constraints for agent 2:

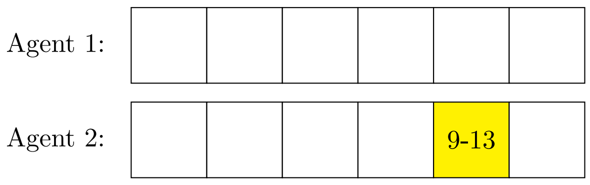


Figure 3.27: Constrained environment where agent 2 bumped into agent 1

3.5.8 Side collision rule

This type of collision occurs when two agents have a vertex conflict at one location, but the further solution does not lead into the previous solution of the other agent. To illustrate this conflict, here is a sample cutout environment:

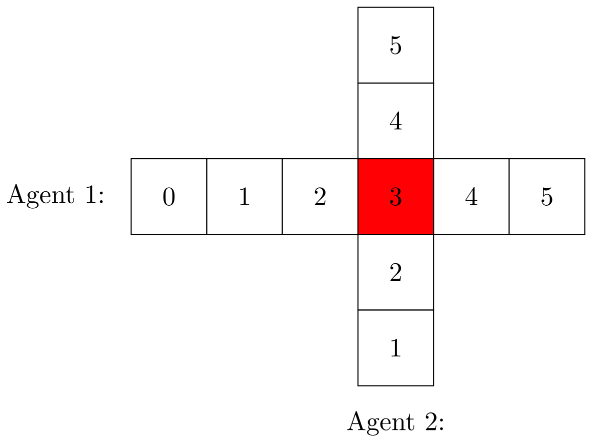


Figure 3.28: Illustration of an side collision

To resolve this type of conflict, one of the agents must wait until the other leaves this location. So the first step is to **recognize** such a kind of conflict. As you can see in the environment, the directions of the states that triggered the conflict are **orthogonal** to each other. However, this is only a necessary condition and **not a sufficient** one, because there may be certain sections of environments where these states have orthogonal directions but there is no side collision. Here you can see a cutout of an environment and the two agents representing such a case of invalid side collision:

Solution Agent 1:

0. (3, 0), *East* → *Forward*
1. (3, 1), *East* → *Forward*
2. (3, 2), *East* → *Forward*
3. (3, 3), *East* → *Left*
4. (2, 3), *North* → *Forward*
5. (1, 3), *North*

Solution Agent 2:

0. (0, 3), *South* → *Forward*
1. (1, 3), *South* → *Forward*
2. (2, 3), *South* → *Forward*
3. (3, 3), *South* → *Right*
4. (3, 2), *West* → *Forward*
5. (3, 1), *West*

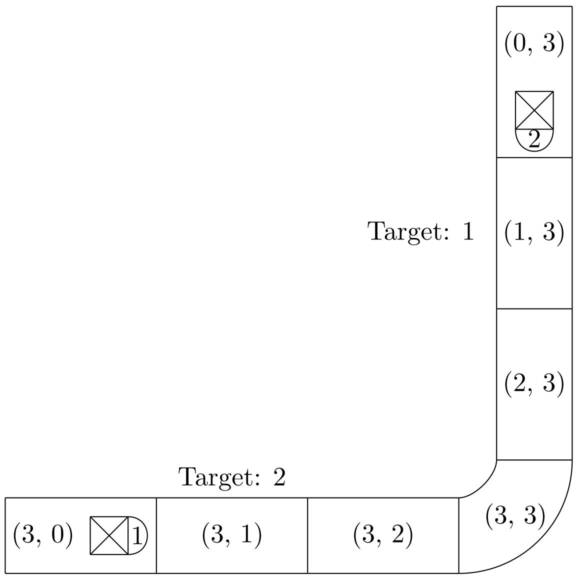


Figure 3.29: Example: Invalid side collision

In this example, the states leading to the vertex conflict are $((3,3), \mathit{East})$ and $((3,3), \mathit{South})$, which means that their directions are orthogonal to each other, but there is **no side collision**. So, after the directions of the states have met the necessary condition, there must be another investigation comparing the previous location of one conflict state and the next location of the other conflict. If they are equal, there is no side collision, otherwise there is on. In this case, further investigation leads to:

$$\begin{array}{l} \textsc{GetNextState}(soln_1(3)) = (2,3), \mathit{North} \\ \qquad\qquad\qquad \overset{?}{=} \\ \textsc{GetPrevState}(soln_2(3)) = (2,3), \mathit{South} \end{array} \implies \text{No side collision} \tag{3.1}$$

After the solver successfully identifies a valid side collision, **one** of the two agents is constrained at the location of the conflict until the timestep when the other agent leaves the location. The reason the solver constrains only one agent is to minimize the number of extended higher level nodes. Therefore, **optimality** is again **not maintained**, but much unnecessary higher level nodes are prevented from being expanded. In Environment 3.28 the constrains would look like:

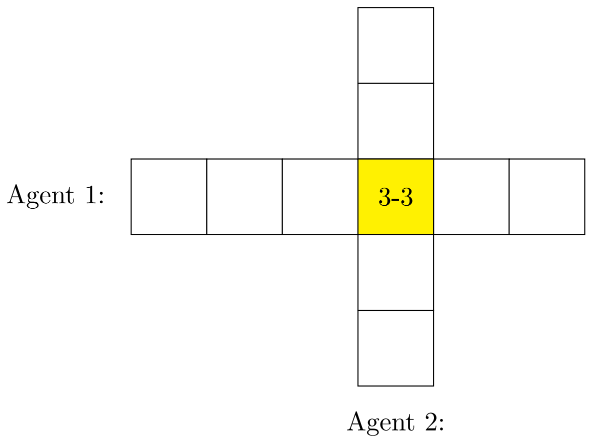


Figure 3.30: Constrained: environment side collision

3.6 Adding speed

The above rules are so far already quite robust to different speeds of the agents, since the terms like *timestep agent leaves / enters a location*. Those will ensure that even an agent needs longer to proceed through a location, the timesteps are right. There are only **small changes** in the following rules:

3.6.1 Applicable critical state

Adjust lower bound

Previously, the lower bound of the multi time constraint at the critical state was set to the timestep at which the agent can leave that location at the earliest. If this agent now has a different speed value, this must also be taken into account. In the constrained Environment 3.16, you can see that the lower bound is set to the value 8 because the agent was already at this location in timestep 7 . Therefore this is the fastest possible route to get there. The agent then has exactly one timestep to move on and can take the other path if it yields a better solution. Since in the addition of the speed the lower bound cannot simply be increased by one value. So the solver must add the timesteps that the respective agent needs until it can proceed in the minimum timesteps. Thus, if the agent in Environment 3.14 has, for example, a speed value $spd = \frac{1}{3}$, it takes $1/\frac{1}{3} = 3$ timesteps to apply an action and can move to another location. The **lower bound** would then be the value $7 + 3 = 10$.

Adjust upper bound

The basic principle how to set the upper bound on the critical states that are applicable remains the same. The only difference is that the number of times the GetNextState function is called until one of the mentioned scenarios occurs is multiplied by the number of timesteps it takes the agent to continue. This value is then added to the respective other timestep, if necessary. So in Environment 3.12, agent 2 will leave the critical state, found by the function, at timestep 15 which had to be applied 6 times. If agent 1 had a sped value $spd = \frac{1}{4}$, the final **upper bound** value would be $15 + 6 \cdot 4 = 39$.

3.6.2 Side collision & rail type corner rule

In these two types of collisions, it is mentioned that only one of the two agents is constrained to keep the high level nodes as low as possible. Since it is more likely that constraining only one agent will lead to a faster solution, because an entire branch in the conflict tree can be removed. If the agents have different speed values, **different strategies** can be used to constrain only one of the two agents. For example, the particular agent that needs more states in the current solution to reach its target location can be constrained. Another strategy would be to constrain the agent with a lower velocity value because it is more likely that the faster one will reach its goal faster if it is ahead of the other. There are many conceivable strategies that are more appropriate for **some instances** than others, but these will not be considered further since these types of collisions will rarely occur.

3.7 Forming the algorithm

Since there are always two conflicting agents and there is always a rule for the initial search to resolve the conflict, there are also two resulting high level nodes. To keep the number of expanded nodes as small as possible, the solver can examine the type of the critical state and expand only one of these two high level nodes. This can lead to a solution that can be computed faster than expanding all possible nodes, but again, optimality suffers. Also, the **completeness** of the algorithm suffers in which the algorithm indicates that there is no valid solution in the current environment, but every environment has a valid solution without considering the maximum number of steps. Such a solution may look as follows:

Proof. Let be given an environment with an arbitrary number of agents. To solve the given MAPF problem, place an agent on the grid and guide it to its target using a search algorithm. At the target, remove this agent. Select the next agent and follow the same strategy until all agents have reached their goal, resulting in a valid solution. □

Returning to the idea of the solver expanding only one higher level node, there are some situations where an agent may be where it is not necessary to expand both. Such a situation could be when there are two conflicting agents and the algorithm that detects the critical states returns an initial inapplicable critical state. This means that both agents are initially facing each other without the possibility to evade by another path, and one of the agents has to be held back from the grid until the other one passes. Such an environment can be illustrated as follows:

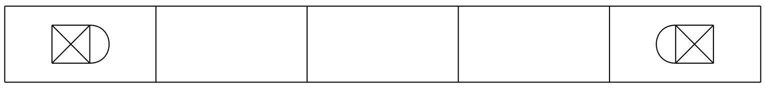


Figure 3.31: Two agent initially facing each other

When such a situation occurs, only one of the agents needs to be held back from the grid. Otherwise, the conflict tree could contain another path with almost the same conflicts to be resolved, which could result in twice the computation time. In order to determine which of the agents must be held back, it is possible to proceed in several ways. For example, the **slower** agent may be withheld, or the agent with the **greater number** of states in its solution. There are several ways that can, but do not have to, lead to a good solution.

The next idea where the number of extended high level nodes can be reduced is when only one of the agents provides an inapplicable initial state. If this is the case, this node can be pruned since it is more likely to lead to a more expensive solution, which still needs to be expanded. The idea behind this is that the agent is held back for too long and cannot start its journey to the target. Combining this idea with the collision described earlier results in the following algorithm after a conflict between two agents has occurred. The full algorithm can be seen at Algorithm 7.

3.8 Adding malfunctions

Adding the malfunction has not been focused on in this approach and only naive rescheduling is considered. There could also be more effective methods to solve this challenge, but these will not be discussed further here. Compared to the previous algorithm, there is no real change as the malfunction is detected in the PYTHON environment. The detection is just an if statement, which then leads to a reschedule that looks like the following. There are **various malfunction** scenarios when an agent sets out to reach the target. Depending on those, the solver must constrain the agent differently to add as many *Wait* actions until the agent is able to continue. To do this, a **new search** is performed where the initial state of the agents is the current location where they are at the timestep of the malfunction. The timestep of the initial search is also set to 0. Then, to ensure that the malfunctioning agent does not leave that location until the malfunction is over, the **next possible locations** are constrained to a certain value depending on the type of malfunction. It should also be noted that once an agent has been placed on the grid, it cannot be removed until it reaches its target location. Thus, it is not possible to apply rule number 6, where an agent can be held back from the environment.

3.8.1 Initial malfunction

This type of malfunction can occur when an agent has a malfunction without having been previously placed on the grid. When this scenario occurs, all of the above rules, including *Rule 6*, can be applied to resolve the current environment. It should be noted that the environment does not check if the agent is **actually on** the grid, but only checks for the elapsed timesteps. To resolve the malfunction, the agent is held back from the grid because it cannot move and there may be another agent that can thus pass the initial location of the malfunctioning agent without waiting the same amount of timesteps. To resolve this scenario, a multi time constraint is then set on the initial location with the **lower bound** of 0 and the **upper bound** with the value $m - 1$.

3.8.2 Standard malfunction

This scenario occurs when an agent wants to perform an action at a specific timestep, but it is directly blocked by a malfunction with the value m. This means that the agent is forced to stay at the current location and can only apply another action after $m-1$ timesteps. To prevent the local search from leaving this location too early, the locations of the next reachable states are constrained. Thus, the **lower bound** is set to the value 0 and the **upper bound** is set to the value $m-1$. In the following figure, you can see how an agent can be constrained when a malfunction occurs at an applicable critical state. Note that the value of the upper bound is represented as the value V:

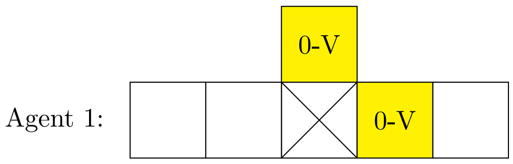


Figure 3.32: Standard malfunction: resolved by constraining next states

3.8.3 Transition malfunction

This scenario can only occur if the respective agent has a speed below 1, i.e. takes longer than one timestep to finally execute an action. If an agent a requires sp timesteps to finally execute an action and applies an action at timestep t, such a type of malfunction may occur at time t' if $t' > t$, $t' - t < sp$, and $sp > 1$. This means that the agent has already applied an action, but still has to wait for the corresponding timestep to finally execute it. Agent a must wait the $m-1$ timesteps at the location of the malfunction. In addition, agent a must also wait $sp - (t' - t)$ timesteps in order to execute the forced action. Thus, the final **upper bound** will be:

$$\text{upper bound} = (m-1) + (sp - (t' - t)) \tag{3.2}$$

To keep the low level search at the location of the malfunction, the solver constrains the next reachable location with the computed upper bound. Note that if the malfunction occurred in an applicable state, there is a possibility that the agent will prefer a different solution, when constraining like in Initial malfunction 3.8.2. Therefore, only the state with the location provided by the forced action are valid. To resolve this scenario, the initial location is constrained by setting the lower **and** upper bound to the value upper bound$+1$. This addition forces the agent to also take the applied action like in the real environment. The location of the next valid state is constrained as in the previous example, with the upper bound of $m-1$ timesteps.

Algorithm 7: Full algorithm with all combined functions

```
input  : Conflict between two agents
output: Constraints how to solve the conflict
critOne = GetCriticaltState (Conflict, agent1);
critTwo = GetCriticaltState (Conflict, agent2);
/* If rear-end collision → constrain new agent                */
if CheckRearEndCollision (Conflict) then
    agt = GetNotWaitingAgent (Conflict, agent1, agent2);
    ConstrainAgent (agt) ;
    return;
/* If side collision → constrain slower agent                 */
if CheckSideCollision (Conflict) then
    if agent1.speed < agent2.speed then
        ConstrainAgent (agent1) ;
    else ConstrainAgent (agent2) ;
    return;
/* If rail corner case → constrain slower agent               */
if CheckRailTypeCornerCase (Conflict) then
    if agent1.speed < agent2.speed then
        ConstrainAgent (agent1) ;
    else ConstrainAgent (agent2) ;
    return;
/* If both initial inapplicable → constrain slower agent */
if IsInitInapplicable (critOne) and IsInitInapplicable (critTwo)
 then
    if agent1.speed < agent2.speed then
        ConstrainAgent (agent1) ;
    else ConstrainAgent (agent2) ;
/* If one initial inapplicable → constrain other agent        */
else if IsInitInapplicable (critOne) then
    ConstrainAgent (agent2);
else if IsInitInapplicable (critTwo) then
    ConstrainAgent (agent1);
/* If no initial inapplicable → constrain both agent          */
else
    ConstrainAgent (agent1);
    ConstrainAgent (agent2);
```

Chapter 4

Evaluation

In order to properly evaluate the results of the algorithm, only **standardized speed** values and **no malfunctions** are used to avoid having to deal with random probabilities of these additions, which can lead to a bias in the results. The environments used, in which an increasing number of agents must navigate to their destinations, are provided by the FLATLAND competition. In these, agents are randomly assigned an initial location defined by a user specified seed. The dimensions of these environments increases with the number of test instances. The first test starts with an environment that has dimensions 25×25. The tests are then completed with an environment that can have dimensions as large as 314×314. In total, there are **41 different** environments where the solver must first compute a valid solution and then apply the STEP function mentioned in section 2.2.5 to validate the computed solution. Only when the function STEP returns the info that all agents have reached their destination, the computation time is stopped. For this process, each solver has the **time limit** of $3600sec = 1h$ in which to compute and validate its solution. Additional information about the settings used in the evaluation can be seen in Table 4.1. If a solver returns an invalid solution and applies it to the environment, this can lead to an **deadlock** where agents cannot apply any action because they block each other. Such a deadlock is not checked and the function STEP is applied as many times as the maximum defined timesteps in the given environment, which may result in a high number of applied steps in the environment.

4.1 Evaluating own algorithm

This section evaluates some diagrams of sample environments that stand out especially. To discuss how well each environment was solved by the algorithm, the number of agents is increased after the previous environment evaluation is completed. This number initially starts with 1 and is increased to 20 as the solver is applied to each environment and number of agents individually, taking into account the mentioned time limit of $1h$. In the following graphs, there is a "reward bar"(purple) that shows the number of agents that reached their target locations in relation to the actual time it took the algorithm to solve the particular instance. The formula for calculating the height of the reward bar is as follows, where *time* is the value of the time in seconds it took the solver to calculate the solution, *agt* is the number of agents in the environment, and *actagt* is the number of agents that actually reached their target:

$$\text{bar height} = \frac{time}{agt} \cdot actagt \tag{4.1}$$

Note: The addition with the "reward bar"is only important for the later comparison, because if the presented algorithm provides a solution, it will also lead every agent to its target. The solver, which is compared later, can lead to a deadlock and thus not all agents will reach their target. In addition, a "no solution bar"(green) is displayed, which reaches to the required computation time in which the solver returned that there is no valid solution for the given environment and the corresponding number of agents. To show the correlations between the required time (in *sec*) (pink) and the number of expanded nodes, the number of low level nodes (cyan), initial search node (orange) and high level nodes (yellow) are also shown. The final comment on the graphs would be that if there is no data at the particular agent number, the solver **failed** to find a valid solution in the given time limit.

Evaluation environment 0, seed 1:

Below you can see the statistics of the environment provided by Test 0. Agents are placed randomly with the default seed of value 1. In the upper left corner you can see a small picture of this particular environment without the placement of agents and their targets.

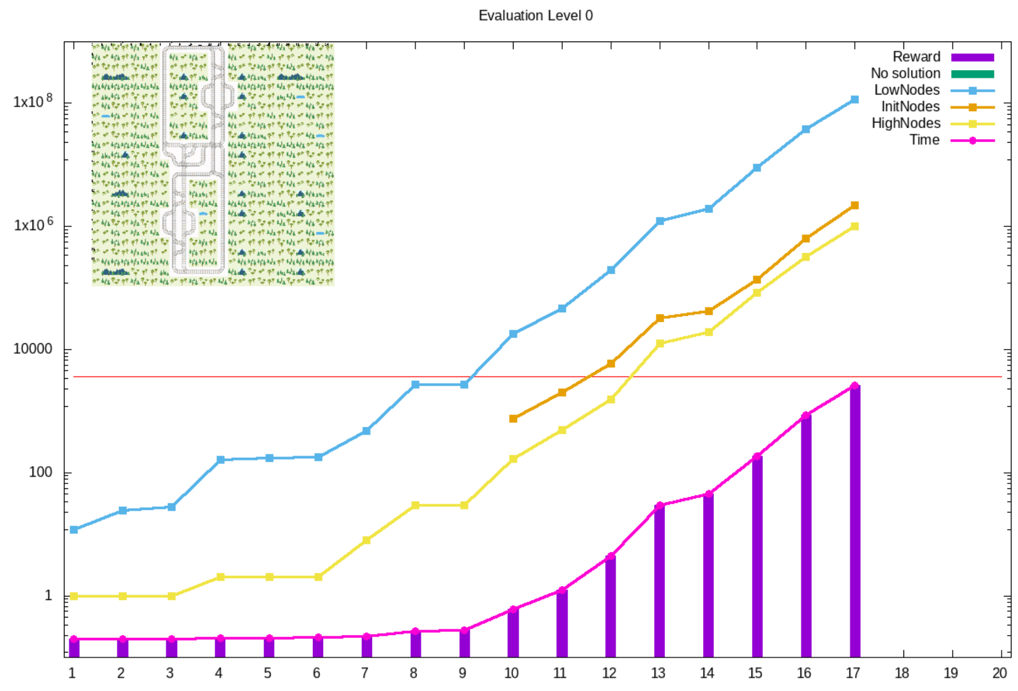


Figure 4.1: Diagram Test 0, seed 1

In Diagram 4.1 you can see a basic type where the computation time is small for a few agents and increases exponentially as more and more agents are added to the environment. It can also be seen that the number of extended high level nodes for the first 9 agent placed almost does not change. This number of extended nodes correlates with the number of conflicts encountered. Thus, it can be seen that for this particular environment and random seed, the initial states for the first 9 agents were quite well placed with respect to the occurrence of conflicts. These conflicts are resolved quite quickly by the presented algorithm, resulting in a very short computation time for this number of agents. As the number of agents increases, the random placement in the environment tends to cause more conflicts, which then leads to more computation time. This is evident in the instances where the number of agents is greater than 14. Thus, for each additional agent, the algorithm requires exponentially more computation time to obtain a valid solution, due to the logarithmic scale in the diagram.

Evaluation environment 3, seed 1:

In this diagram, it can be seen that the computation time increases dramatically at a number of 6 agents, rather than at a number of 12 as in the previous environment. The total number of agents guided to their target locations is also lower than in the previous example.

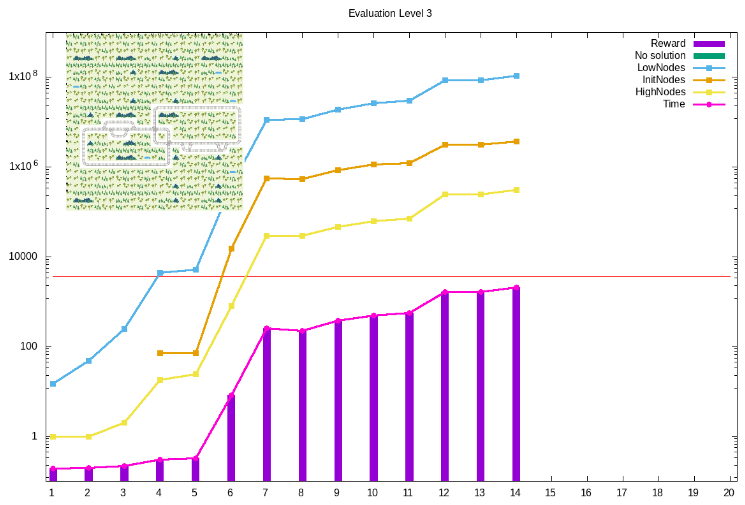


Figure 4.2: Diagram Test 3, seed 1

Evaluation environment 5, seed 1:

As stated in the section Forming the algorithm 3.7, there is always a valid solution when the default search is considered, but the algorithm is not complete in every case. So, in the following statistics you can see that after the specified time shown by the respective bar, the solver returns that there is no valid solution.

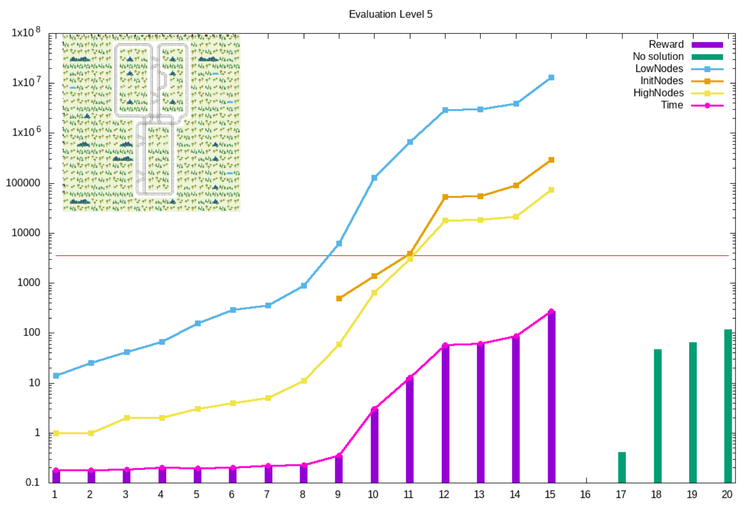


Figure 4.3: Diagram Test 5, seed 1

Evaluation environment 5, seed 7:

In this diagram, the same environment was taken as in the previous Diagram 4.3, but the initial states have changed due to the change of the default seed 1 to the seed with the value 7. In this particular case, some interesting anomalies are seen, all due to the changing seed. For example, the solver could not find a solution in its time limit with the number of 15, 19 and 20 agents, but could with the number of agents in between. Furthermore, even after adding this many agents, the computation time is very low. This confirms that the number of agents placed is not the point, but the initial states of the agents and whether they cause conflicts.

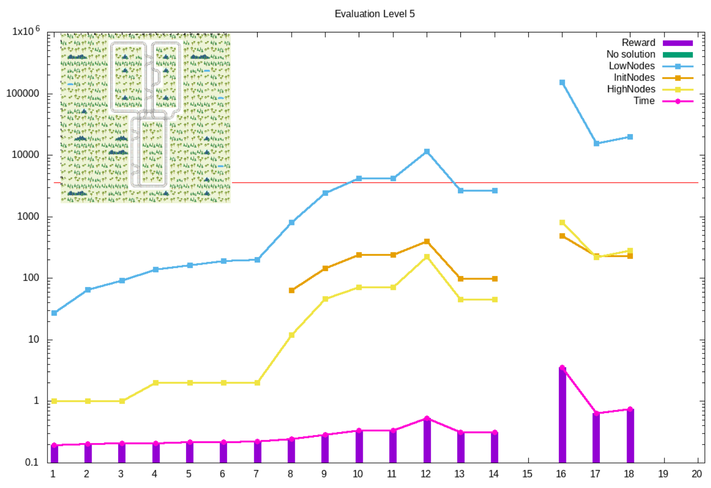


Figure 4.4: Diagram Test 5, seed 7

Evaluation environment 40, seed 1:

The following diagram shows the collected statistics of the solver when applied to the last test environment. As mentioned earlier, the dimensions are 314×314, which is quite large. This test shows that not only the number of agents, but also the size of the environment is crucial to get a good computation time. The reason for this is that it is simply more likely that an agent will not have a conflict after the low level search is applied, because the agents are far enough apart from each other. This can also be seen in the number of expanded high level nodes and thus the number of collisions. Even when 11 agents are present in the environment, the solver only expanded 2 high level nodes, indicating that only one conflict occurred that could be resolved quickly.

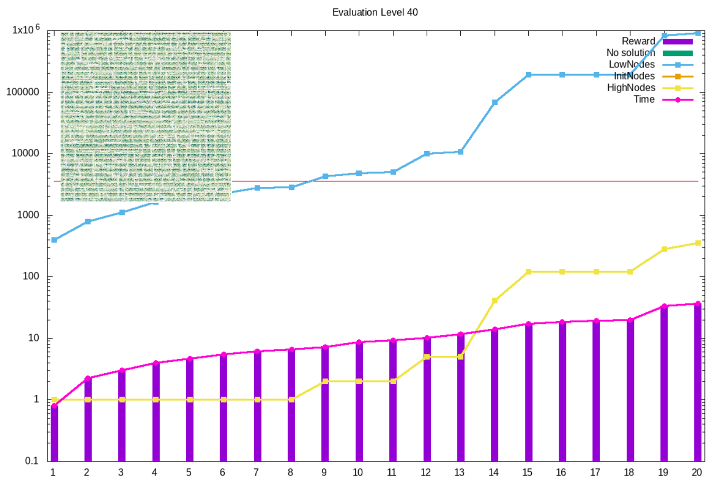


Figure 4.5: Evaluation: Test 40, seed 1

Multiplot of some environments, seed 1:

After all the special cases presented before, the following graph shows the trend of several randomly selected statistics for the given environment. It can be seen that regardless of how the agent was initially set, as the number increases, the computation time also increases exponentially.

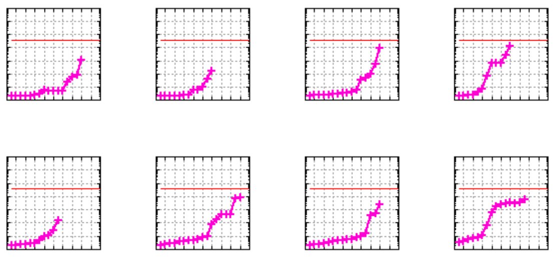


Figure 4.6: Multiple plot of some statistics, seed 1

4.2 Comparing approaches

In order to properly assess the strengths and weaknesses of the presented algorithm, the solution of the previous year's winner (2020) is compared in different values. This approach is also based on classical planning, where the given MAPF instance is solved by a **hand crafted** algorithm, rather than by a solver that emerges from the Reinforced Learning (RL) branch. The contributors to this solution are the same as the authors of the previously cited paper in the chapter Related work 3.4. These authors have even published their own paper under the team name **An_old_driver** describing how they solved this challenge: *Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge* [Li+21a]. The code used and the basic example of how to run it can be found on the corresponding GITHUB repository [Li+20].

4.2.1 Comparable values

To ensure a fair comparison, the same method is used as in the tests shown previously. Thus, a particular environment E: X is considered, where X stands for the respective test number. This environment is then divided into 20 different instances, where an increasing number of agents are placed and then the solver is applied to them. As before, this number starts at 1 and increases up to 20. To refer to the particular instance of each environment, the notation E: X-A is used, where $A \in [1, 20]$ and represents the number of agents in that instance and X is defined as above. From these instances, the following values can be evaluated, which the algorithm has calculated in its solution process. The resulting values are the computation time ct, the number of steps sc, and the completed agents ca. The **computation time** is defined as in the tests shown previously. The **step count** is the number of iterations required to bring all agents to their targets, which is also referred to as the "makespan"or simply the maximum number of states of all agent solutions. The value **completed agents** is the number of agents that have reached their destination after either all agents have reached their destination or the maximum timestep has been reached because the solver has given an invalid solution.
Note: When testing certain environments, the compared algorithms often result in a different number of solved instances. Therefore, to keep the comparison fair, only the statistics of instances solved by both approaches are considered for some selected value types. These types of values are the following:

Overall %:

The percentage of the number of all agents in all instances of an environment that reached their goal and the number of all agents that were given an initial state, which will be 210. For a given environment corresponding to test number X, this formula would look like this:

$$\frac{\sum_{A=1}^{20} \text{completed agents}_X(A)}{210} \cdot 100 \tag{4.2}$$

Steps:

This value represents the sum of the steps count in all instances of the environment solved by **both** solvers, so:

$$\sum_{A=1}^{20} \text{steps count}_X(A) \cdot \text{both}_X(A) \tag{4.3}$$

Where the function $\text{both}_X(A)$ returns 1 if both instances in the respective environment were solved by both solvers, else 0. It should be noted that this value can **hardly vary** in the compared solver. The reason for this is that there may be environments where the solver does not compute a valid solution and the agents are completely deadlocked, or the solver simply does not return a better solution even if there is still an agent that can move. Therefore, the function STEP is called until the maximum number of timesteps is reached, which can lead to a very high number of steps. To further illustrate, consider the following FLATLAND competition environment where an agent can still escape from the situation, but the solver only returns the action *Wait* until the maximum number of timesteps is reached.

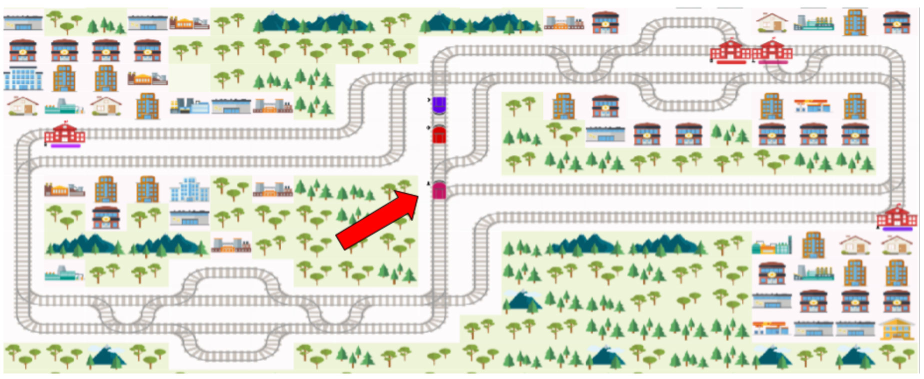


Figure 4.7: Example environment An_old_driver: solver fails to return other action than *Wait*

Solved %:

This value shows a similar value as the value of *Overall %*. This time, however, not all instances of an environment are considered, but the instances that both solvers solved in the time limit, so:

$$\frac{\sum_{A=1}^{20} \text{completed agents}_X(A) \cdot \text{both}_X(A)}{\sum_{A=1}^{20} \text{all agents}_X(A) \cdot \text{both}_X(A)} \cdot 100 \tag{4.4}$$

Note: There is no column for this value in the evaluation of the presented algorithm, since all agents will reach their targets once the solver has returned a solution. So there would be only one column with the value 100, but it is omitted for clarity.

Time:

To calculate this value, the required computation time of the respective instance is summed up when both agents have solved it. This will then lead to:

$$\sum_{A=1}^{20} \text{computation time}_X(A) \cdot \text{both}_X(A) \tag{4.5}$$

In addition to the defined values, there are some values under the tables that represent the result of a certain calculation. Thus, one can find a row where all data points represent the average **AVG** or the sum **SUM** of all the above tests at the certain value. For example, it can be the average of all values, which represents the *Overall %* of the particular algorithm. There will be one more column that is not the calculation of the above values. This is the row **ALL** and its data points represents the calculations of the actual total of the above values, that is, when the function $\text{both}_X(A)$ is not applied. It is added because most of the values are calculated with respect to the fact that the two algorithms must have solved the instance to be considered.

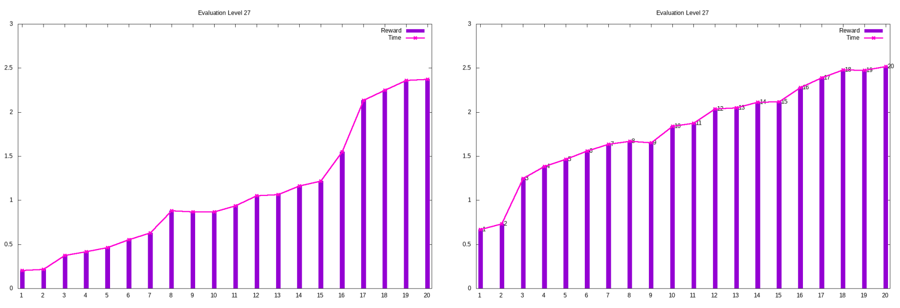
4.2.2 Final data discussion

Finally, after declaring all the required terms, the two solvers can be compared to find out which performs better. So you can see two tables consisting of all the collected and computed statistics that the solver used during its calculation. These tables are split into two halves, with the left half representing the data from the algorithm explained in this paper, and the right half representing the competing algorithm from the An_old_driver team. In these tables, various statistics of the respective test environment can be seen, showing certain properties of the algorithm. These are discussed below:

Discussing Table 4.2:

The data in Table 4.2 were collected by randomly assigning the initial states of the agents with respect to the default seed with value 1. After a quick check, it can be seen that the average values of the *Overall %* column are not that far apart, which are 62.39% and 74.4%. However, this value should not be considered without taking into account the **total time** it took the respective solver to achieve this result. To extract this information, you need to look at the **ALL** row and the corresponding *Time* column. So these values would be 81739.74s for the solver relying on the presented algorithm and 5149.68s for the compared solver of the An_old_driver team. This means that the presented algorithm gives a worse overall result and is almost **16 times slower**. This time interval can also be confirmed if the individual time values of the tests are examined. In most cases, the time values are significantly higher for the presented algorithm than for the compared one. This means that the compared algorithm is even significantly faster in the environments solved by both algorithm. But after further examination of the various tests, it can be seen that the presented algorithm works better for some particular environments that are displayed with a green row color. In these tests, the presented algorithm found faster solutions. Moreover, these results are also quite comparable, as both agents brought 100%

of agents to their target locations, so the disadvantage illustrated in Figure 4.7 is not present. To illustrate the performance of the algorithms, sample graphs showing the results of both algorithms in the TEST 27 environment are shown below:



Presented algorithm An_old_driver

Figure 4.8: Comparing both results of Test 27, seed 1

In these linear plots, with the upper y-bound set to $3s$, one can see that the overall performance of the presented algorithm is slightly better. It does not take as long as the compared solver to obtain a sum of the makespans that is equal, i.e. 15696. In the other green colored row results, one could see similar graphs, which is not shown further. In the remaining test environments, where the respective row is not colored green, the results of the compared algorithm are significantly better overall.

Since the presented algorithm always brings all agents to its target as soon as there is a solution in the time limit, it has the upper hand in terms of computing a **valid solution** of the given instance. The reason for this is that compared solver does not always compute a valid solution. Thus, only 87.56% of all agents in the instances solved by both algorithms reached the target. This value can be taken from the line **AVG** in the column of the value *Solved %*.

Discussing Table 4.3:

To further analyze the results of the respective test environments, the same test as before is performed, but the seed defining the random initial states of the agents is changed to the value 7. At the end of the table in the row **AVG** it can be seen that the compared solver from the An_old_driver team has again a better *Overall %* value than the solver based on the presented algorithm. The results of the own solver are again achieved in a much slower time, to be exact almost **4.5 times slower** than the comparison algorithm. It can be seen immediately that this multiplier has dropped to a lower value compared to the test results in Table 4.2. This is largely due to the 38 and 40 test environments, where the comparison algorithm took quite a while to compute a result. If these test environments were excluded from the total time summation, the own solver would be **14.5 times slower** than the comparison solver. In addition, the previously green colored test environments in Table 4.2 now provide worse results than before or are not comparable at all. For example, in the 38 and 40 test environments, the presented algorithm gives significantly better results, which is due to the problem presented in Figure 4.7. Therefore, the function STEP must be applied in the PYTHON environment as many times as maximum step values are defined in the respective test. Such an application of a STEP function can take up to $0.02s$ of computation time if the number of agents and the dimensions of the environment are high enough. For example, if this STEP function has to be called 20.000 times in an instance, this would result in an $400s$ increase for a single instance. So these are **not comparable**, even if the own solver calculates much better results. There is only one comparable better test environment, namely the TEST 39. The presented algorithm even calculates a better solution than the compared one.

In these test results, it can again be seen that the comparison solver brought only 86.73% of the agents to the target in the test instances solved by both algorithms. Since the presented algorithm again lead all agents to its target, this is the **only advantage** over the compared one.

Processor settings	QEMU Virtual CPU v. 2.5+, 1 Core, 512KB Cache, 2.35GHz
Memory limit	4GB
Time limit	3600s = 1h

Table 4.1: Additional technical informations about the settings used for the evaluation

	Own solver			An_old_driver			
Level	Overall %	Steps	Time (s)	Overall %	Solved %	Steps	Time (s)
0	72.86	506	3721.56	49.05	59.48	5860	17.53
1	26.19	468	2.28	69.05	100	458	4.38
2	81.43	831	3425.68	40.00	39.18	16848	24.41
3	50.00	962	7305.38	66.19	100	718	5.62
4	64.76	613	3.13	26.19	100	567	4.05
5	57.14	550	496.49	47.62	70.00	4042	11.66
6	31.43	427	5.29	100	100	429	4.50
7	50.00	547	28.87	26.19	100	550	4.00
8	37.14	461	25.34	3.81	7.69	8530	15.86
9	31.43	650	3.96	79.05	100	597	3.73
10	50.00	305	1.08	7.14	100	305	1.88
11	31.43	918	1295.44	68.10	54.55	13902	14.12
12	64.76	1459	937.31	25.71	33.82	15364	36.47
13	31.43	835	3.68	100	100	833	4.23
14	37.14	1068	189.88	26.19	100	1070	5.54
15	81.43	1584	779.65	33.33	35.67	36113	59.38
16	37.14	1068	1822.92	100	100	1140	6.12
17	50.00	1469	3724.36	100	100	1456	8.62
18	81.43	1091	4.17	26.19	100	1090	5.16
19	21.43	894	21.63	58.57	62.22	27714	18.88
20	31.43	1070	20.56	100	100	1055	5.86
21	31.43	1264	7.91	100	100	1121	5.81
22	21.43	1983	2640.73	100	100	1946	12.42
23	83.81	2893	8807.05	90.48	91.48	25027	45.75
24	81.43	3136	1615.12	100	100	3139	21.26
25	100	8880	1230.94	52.38	52.38	204495	446.80
26	43.33	3642	2404.79	100	100	3644	23.49
27	100	7848	21.58	100	100	7848	36.20
28	64.76	2604	346.68	100	100	2604	21.65
29	37.14	2889	7658.94	100	100	2887	30.31
30	64.76	6282	9061.55	72.86	91.18	46450	139.07
31	90.48	5600	1882.81	100	100	5590	57.89
32	57.14	5361	2302.80	100	100	5286	67.56
33	81.43	5511	3843.16	94.29	100	5499	69.22
34	90.48	6474	207.80	100	100	6470	83.87
35	100	8550	3587.44	100	100	8570	132.09
36	100	7580	82.08	100	100	7580	141.68
37	90.48	9921	8768.13	100	100	9936	186.75
38	100	9119	135.39	100	100	9073	240.83
39	100	10061	149.81	88.10	88.10	139566	1688.74
40	100	11144	238.38	100	100	11144	377.65

AVG	62.39			74.4	87.56		
SUM		138518	78811.74			646516	4091.02

ALL		141272	81739.74			873374	5149.68

Table 4.2: Test results 1: own solver / an_old_driver, seed: 1

	Own solver			An_old_driver			
Level	Overall %	Steps	Time (s)	Overall %	Solved %	Steps	Time (s)
0	43.33	537	6.51	100	100	620	4.12
1	26.19	470	8.13	37.14	100	482	3.32
2	31.43	565	36.13	100	100	555	3.83
3	26.19	354	2.62	100	100	343	2.81
4	31.43	559	416.74	39.52	69.70	3155	6.79
5	74.29	502	3.52	43.33	100	473	3.79
6	72.86	1061	207.05	100	100	1178	5.75
7	64.76	682	1385.35	100	100	672	4.75
8	80.00	770	2620.18	75.24	89.92	1599	7.00
9	50.00	1045	5540.08	57.62	89.52	2235	8.15
10	72.86	1231	337.93	31.90	34.64	19003	28.81
11	50.00	886	4359.34	100	100	874	6.05
12	43.33	1256	1550.52	100	100	1142	5.60
13	21.43	935	2984.83	91.90	100	964	5.06
14	100	2338	384.77	100	100	2330	11.13
15	90.48	1676	372.63	48.10	48.95	33962	54.42
16	50.48	1198	5713.79	61.90	84.91	6262	15.08
17	100	2327	3525.62	100	100	1916	11.55
18	72.86	2385	109.16	80.95	96.08	5430	14.18
19	50.00	2169	6525.00	59.05	77.14	11934	29.41
20	21.43	903	4705.61	100	100	987	5.56
21	37.14	1504	5007.40	42.86	69.23	26857	27.55
22	64.76	2838	2445.69	60.00	60.29	75189	80.77
23	90.48	4887	126.27	100	100	4887	19.34
24	57.14	3460	41.72	100	100	3433	16.62
25	64.76	6662	2096.32	100	100	6646	31.02
26	100	5064	561.87	80.95	80.95	43916	246.23
27	100	2670	88.47	57.14	100	2668	26.21
28	90.48	5107	238.56	62.38	62.11	320207	460.16
29	100	5014	401.86	100	100	5044	39.23
30	50.00	5223	209.91	100	100	5217	35.58
31	100	5588	4537.37	57.14	57.14	402260	1103.04
32	100	12949	742.16	100	100	12949	71.32
33	100	5703	353.84	66.19	66.19	208780	874.54
34	100	7082	2929.44	100	100	7070	97.53
35	100	6583	324.91	100	100	6536	111.83
36	100	7525	986.81	100	100	7537	153.75
37	100	7596	1280.94	100	100	7613	205.25
38	100	9701	149.65	65.24	65.24	460316	5245.53
39	100	9041	190.70	100	100	9043	305.52
40	100	11448	327.91	3.81	3.81	1035238	5557.01
AVG	71.42			78.59	86.73		
SUM		149494	63837.32			2747522	14945.19
ALL		150715	66775.76			2897198	15373.82

Table 4.3: Test results 2: own solver / an_old_driver, seed: 7

Chapter 5

Conclusion

In this bachelor thesis, an approach to solving a specific MAPF instance was shown. This takes into account additional challenges such as maintaining the direction of the agents, varying velocities or malfunctions during the agents' path to their respective destinations. The above approach was carried out as part of FLATLAND competition, founded by SWISS FEDERAL RAILWAYS (SBB) and DEUTSCHE BAHN AG (DB). The basis of this approach is the Conflict Based Search (CBS) [Sha+15], which was further extended in this work by defining specific rules to resolve certain agent conflicts. These rules then ultimately prevent the algorithm from requiring unnecessary computation time with respect to the given FLATLAND environment and properties.

After the evaluation and comparison of the presented algorithm, the intended main goal of the FLATLAND competition must be retrieved. This goal was to get as many agents as possible to their destination as fast as possible. Immediately before, empirical results show that this goal was only **insufficiently achieved**. One drawback of the presented algorithm arises directly from the goal of the challenge, namely that as many agents as possible should reach their goal. The algorithm only computes valid solutions for the given MAPF instances where all agents reach their target location. Therefore, the solutions where some agents would not reach their goal but could possibly be computed in a much faster time are not considered at all. This could be an advantage in another type of problem, but in this competition it is not necessary at all.

The next drawback is that the presented algorithm can solve a given instance only for a relatively small number of agents. If the number of agents becomes too large, the algorithm will **not terminate** in a reasonable time and thus would not provide a solution where even one agent reaches its goal. Added to this already poor solution calculation is the fact that if the solver can determine a solution for the given instance, it takes **significantly longer** on average than comparable solvers.

As you may have wondered, there is no official evaluation in the intended manner of the FLATLAND competition and only own tests were performed in terms of computation time. The reason is that the presented algorithm is not suitable for the majority of tests and would not provide a solution and would simply exceed the given time limit. In most of the provided official tests, the number of agents is set to at least 30. In the chapter Evaluation 4 it can be seen that the solver has problems to get a solution even with a much smaller number of agents. Since the solver would not be able to compute a solution, it will not gain any rewards from these tests, which basically makes it **completely useless** for instances with a high number of agents.

Summary of results collected:

The presented algorithm is only able to solve instances of the FLATLAND competition where the number of agents must be kept very small. However, it is not able to efficiently provide results as required by the main objective of the challenge. If this algorithm were to compete with other approaches, it would **not stand a chance**. The presented algorithm is not a great addition to previous Multi-Agent Path Finding algorithms, as solvers relying on other MAPF algorithms may be able to compute better and faster solutions for instances of the FLATLAND competition.

List of Figures

List of Tables

List of Algorithms

Bibliography

[Atz+18] Atzmon, D., Stern, R., Felner, A., Wagner, G., Barták, R. and Zhou, N.-F. (2018). Robust multi-agent path finding. In: *Eleventh Annual Symposium on Combinatorial Search.*

[Bau+21] Baumberger, C., Eichenberger, C., Egli, A., Ljungström, M., Mohanty, S., Mollard, G., Nygren, E., Spigler, G. and Watson, J. (2021). *Flatland documentation.* `https://flatlandrl-docs.aicrowd.com/`. Accessed: 2021-07-07.

[Cra17] Craw, S. (2017). Manhattan Distance. In: *Encyclopedia of Machine Learning and Data Mining.* Ed. by C. Sammut and G. I. Webb. Boston, MA: Springer US, pp. 790–791. ISBN: 978-1-4899-7687-1. DOI: `10.1007/978-1-4899-7687-1_511`. URL: `https://doi.org/10.1007/978-1-4899-7687-1_511`.

[Koz92] Kozen, D. C. (1992). Fibonacci Heaps. In: *The Design and Analysis of Algorithms.* New York, NY: Springer New York, pp. 44–47. ISBN: 978-1-4612-4400-4. DOI: `10.1007/978-1-4612-4400-4_9`. URL: `https://doi.org/10.1007/978-1-4612-4400-4_9`.

[Li+20] Li, J., Chen, Z., Zheng, Y. and Chan, S.-H. (2020). *Jiaoyang-Li/Flatland Solution of NeurIPS 2020 Flatland Challenge from the team An_old_driver.* `https://github.com/Jiaoyang-Li/Flatland`. Accessed: 2021-07-07.

[Li+21a] Li, J., Chen, Z., Zheng, Y., Chan, S.-H., Harabor, D., Stuckey, P. J., Ma, H. and Koenig, S. (May 2021a). Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge. *Proceedings of the International Conference on Automated Planning and Scheduling* 31.1, pp. 477–485. URL: `https://ojs.aaai.org/index.php/ICAPS/article/view/15994`.

[Li+21b] Li, J., Harabor, D., Stuckey, P. J. and Koenig, S. (2021b). Pairwise Symmetry Reasoning for Multi-Agent Path Finding Search. *CoRR* abs/2103.07116. arXiv: `2103.07116`. URL: `https://arxiv.org/abs/2103.07116`.

[ML21] Mohanty, S. and Laurent, F. (2021). *Flatland Multi Agent Reinforcement Learning on Trains.* `https://pypi.org/project/flatland-rl/`. Accessed: 2021-07-07.

[Moh+20] Mohanty, S., Nygren, E., Laurent, F., Schneider, M., Scheller, C., Bhattacharya, N., Watson, J., Egli, A., Eichenberger, C., Baumberger, C., Vienken, G., Sturm, I., Sartoretti, G. and Spigler, G. (2020). *Flatland-RL : Multi-Agent Reinforcement Learning on Trains.* arXiv: `2012.05893 [cs.AI]`.

[Sha+15] Sharon, G., Stern, R., Felner, A. and Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219, pp. 40–66. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/j.artint.2014.11.006`. URL: `https://www.sciencedirect.com/science/article/pii/S0004370214001386`.