Railway Scheduling

Alexej Lesser and Arne Jacoby

University of Potsdam, 14469 Potsdam, Germany

Abstract. The Railway Scheduling problem involves determining valid action sets that move trains through a railway network without collisions while considering time constraints. In this project, we used Answer Set Programming (ASP) together with the Flatland framework to develop a grid-based ASP solution and extended this encoding to include train speed variations. We experimented with random train malfunctions and implemented a simple malfunction-handling mechanism. Additionally, we explored a graph-based encoding that introduces a restrictive form of waiting on edges. Our results show that while the graph-based approach has potential, it requires further refinement to improve performance. A tool for clingo visualization and a batch testing script are presented. This work provides a foundation for future research on optimizing railway scheduling through ASP and graph-based modeling.

1 Introduction

This report presents the results of our group work on the semester project *Railway Scheduling*, where we address train scheduling problems within the Flatland framework[1] using Answer Set Programming (ASP)[2]. Our objective was to explore innovative solutions and experiment with new approaches, with a particular focus on incorporating train speed variations and malfunctions. The complete project code is available on GitHub at `https://github.com/arugu1a/AStar_Flatland`.
For better readability, we have changed the names of some facts from our original encoding (e.g. `(Y, X)` to `Position`) to avoid distracting from the relevant information. Additionally, we have omitted some atoms to keep the code examples concise. The omitted parts will be indicated with an ellipsis "...".

2 Grid Based Encoding

Throughout the course, we have developed multiple grid-based encodings, testing various approaches. In this chapter, we focus on the latest iteration of our grid-based encodings, called `flat_based`, which offers the most functionality. The only exception is the track encoding, a module designed to be compatible with all our grid-based encodings.

2.1 Track Encoding

The track ID encodes allowed connections in a 16 bit format [3]. 4 blocks of 4 bits encode the directions that a train can take going towards a specific direction. In the example below, a train traveling eastward has a choice between going further east or south, but it cannot go to the north, as there is no track and it cannot go back westward.

North	East	South	West
0001	0110	0000	0001

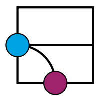


Fig. 1. 16 Bit encoding of track(5633), each 4-Bit pair represents N,E,S,W directions. This example shows the track to the right. Highlighted are two directions, ingoing and outgoing. In this example the connection rule produces the action move_right.

The track ID is converted to this 16-bit representation, so only new track facts with new IDs have to be added to without making any changes to the encoding. This is done with the following rules:

```
connection(Tr,Dir_in ,Dir_out ,Act)
tr_map(Dir_map, Dir_in, Tr)
dir_map(Dir_out,Dir_map)
action_from_dir_dir_map(Act,Dir_in ,Dir_out ,Dir_map)
```

The rule `tr_map` is used to isolate the Integer representing the 4 Bit tuple, that encodes possible choices for the train. The track ID is masked for one of the four directions and shifted to the left.

`dir_map` is used to obtain possible outgoing directions using the output from `tr_map`. Finally, rule `action_from`_dir_dir_map is used to get an action, given ingoing and outgoing directions and the output of `dir_map`.

With that, if an ingoing train direction on any given cell type is given, the connection rule can be used to obtain the outgoing train direction, if an action is provided. Alternatively, the action can be obtained if both directions are given. Wait actions are a separate case, because these are always allowed and have to result in a transition to the same cell while keeping ingoing and outgoing directions the same. For train movement, the inability to turn around is part of the track encoding and it is possible to add track IDs that would allow turnarounds.

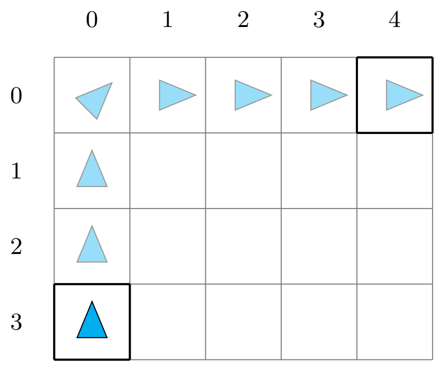
2.2 Train Spawning

A Flatland environment specifies a time window for each train within which it can spawn at any time step. To limit the model count, we reduce the spawn window to only feasible spawn times.This is done by subtracting the minimum travel time `MinTravelTime` from the latest arrival time. An example calculation of the minimum travel time is shown in figure 2.2. If a train spawns at the latest possible moment, it can still reach its target in time, provided that a direct path exists and no other trains interfere.

```
{spawn(ID, EarliestDeparture..ArrivalTime-MinTravelTime)} = 1 :-
                start(ID, (Y, X), EarliestDeparture, _),
                end(ID, (B, A), ArrivalTime),
                speed(ID, Speed),
                MinTravelTime = |Y-B|+|X-A|*Speed.
```

This choice rule selects exactly one spawn time per train from the eligible time window. Two rules handle the actual spawning. At the selected spawn time a `move_forward` action is issued, signaling Flatland to move the train onto the map. In the next time step, the train is placed on its starting cell and is ready to execute further actions. Additionally, a few formal rules are required to guarantee compliance with Flatland's constraints. We generate wait actions for all time steps before the spawn time. This ensures that the action list fed into Flatland is properly aligned with the time steps. However, this step may no longer be necessary in newer iterations of `solve.py`. Finally, an integrity constraint prevents any train from having a spawn time of 0. This constraint enforces a behavior in Flatland that was not widely known at the beginning of the semester but was identified through debugging and group discussions.

$train(blue).$
$speed(blue, 3).$
$start(blue, (3, 0), StartTime).$
$end(blue, (0, 4), EndTime).$



$$MinTravelTime = |Y - B| + |X - A| * Speed$$
$$MinTravelTime = |3 - 0| + |0 - 4| * 3$$
$$MinTravelTime = 21$$

Fig. 2. Calculation of `MinTravelTime` for a train with `Speed = 3`

2.3 Action Generation - Pathfinding

Action generation is handled by three rules. The first is a choice rule that generates all possible solution candidates by considering all potential actions:

```
{action((train(ID)), Action, Time)} :-
          position(ID, (Y, X), _, Time),
          command(Action),
          end(ID, _, EndTime),
          Time <= EndTime.
```

For each position of every train, an action is picked as long as it happens before the train's personal deadline (`EndTime`) and is one of the allowed commands (e.g. `move_forward`). The only position that's always guaranteed is the train's starting cell, though the exact time can vary. From there, new positions are determined based on the actions chosen by the choice rule. Using the connection facts, the position gets updated, and the choice rule picks again. For the case that a wait action is chosen, there is another position update rule, so that the train remains on the same cell with the same orientation.

2.4 Collisions

In general, there are two types of collisions that must be prevented. The first type occurs when two trains move into the same cell simultaneously, or when a moving train collides with a stationary one. The latter scenario is only possible if the encoding allows waiting, but it must still be accounted for. To maintain clarity, we refer to this type as a **standard collision**.
The second type is a head-on collision, where two trains occupy adjacent cells and move toward each other at the same time. A unique characteristic of this collision is that if it is not explicitly handled, the trains will swap places instead of stopping. Due to this behavior, the term **swap collision** became commonly used in course discussions. Figure 2.4 illustrates examples of both collision types.

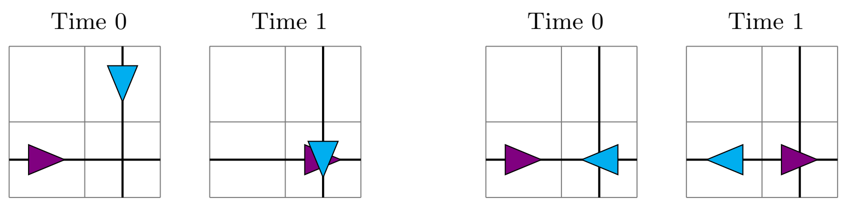


Fig. 3. The 2 Collision Types, left: Standard Collision, right: Swap Collision

In all our basic encodings without additional speed implementations, collision prevention is straightforward using two integrity constraints. The first constraint ensures that no two trains occupy the same cell at the same time, effectively preventing all standard collisions. The second constraint explicitly forbids trains from swapping places, thereby eliminating swap collisions.

For a more detailed explanation, see Section 3.2, which provides an in-depth description of a complete collision handling approach, including the handling of speed-related collisions.

3 Speed

In Flatland, the Speed variable defines the time needed for a train to move from one cell to another. Since a train can only occupy a single cell at a time, it remains on its current cell for `Speed - 1` time steps before immediately arriving at the next cell. Extending the railway encoding to support speed functionality requires adjusting two key components: the encoding of the time step and collision handling.

3.1 Time Step Adjustments

In a standard encoding all trains travel with a speed of 1. In code that looked similar to this:

```
position(Train_ID, New_position, New_direction, Time+1) :-
    ...
```

To allow for different speeds from train to train, we change the constant +1 to a variable `Speed` depending on how fast the train is. Thanks to the simple speed fact representation, this is easily done.

```
position(Train_ID, New_position, New_direction, Time+Speed) :-
    Speed(Train_ID, Speed), ...
```

Intuitively, trains with a slower speed can wait for a single time step in the Flatland environment. Because of that, the wait position update doesn't need to be modified.

3.2 Collision Adjustments

Just changing the time step will break the encoding. The second and more difficult adjustment is to change the collision handling to allow for the different speeds.

Time 0 $position(blue, (0, 0), e, 0).$
$action(blue, move\_forward, 0).$
$position(violet, (0, 1), e, 0).$
$action(violet, move\_forward, 0).$

Time 1 $position(blue, (0, 1), e, 1).$
$action(blue, move\_forward, 1).$

Time 2 $position(blue, (0, 2), e, 2).$
$action(blue, move\_forward, 2).$

Time 3 $position(blue, (0, 3), e, 3).$
$action(blue, move\_forward, 3).$

Time 4 $position(blue, (0, 4), e, 4).$
$action(blue, move\_forward, 4).$
$position(violet, (0, 2), e, 4).$
$action(violet, move\_forward, 4).$

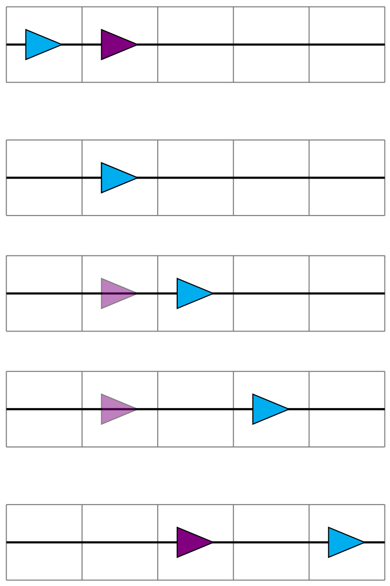


Fig. 4. Example of a non-working Collision after only doing the Time Step Adjustment. The faster Train ▶ with Speed = 1 moves through the slower Train ▶ with Speed = 4 because of the lacking fact representation of the slower Train.

Looking at figure 4, we can see what happens if we only change the time step and not the collision logic. The blue train passes through the slower violet train because in time steps 1 to 3 there are no facts representing the existence of the violet train that the faster blue train could "interact" with. In order to fix this, we need to introduce a new fact for the in-between time steps. It is important to note that we cannot simply use the established fact `position` since `position` is used to generate new actions, which is not what we want. When a train has already received an action it first carries it out, before new actions are allowed. Therefore, we introduce a new fact `in_motion` to represent the movement or motion of a train from one cell to another. The first integrity constraint can be adopted from the basic grid encodings: "No two different trains can share the same position at the same time." This still prevents moving into each other at the same time and moving into a waiting train.

```
in_motion(ID, (Y, X), (Y+B, X+A), Time..Time+Speed-1) :-
                position(ID, (Y, X), Direction, Time),
                action(train(ID), Action, Time),
                connection(Track, Direction, (B, A), Action),
                speed(ID, Speed),
                cell((Y, X), Track).
```

The body of this rule is the same as the position update body. A key difference and why we cannot easily derive one from the other is that there are multiple `in_motion` facts per action if the train is slower than 1. For trains with `Speed = 1` there is exactly one `in_motion` fact per action. This means that the following integrity constraints apply to all trains equally.

```
:- in_motion(ID, (Y, X), _, Time),
    position(ID2, (Y, X), _, Time),
    ID != ID2.
```

This integrity constraint reads as: "A train cannot occupy the same cell at the same time that another train is in the process of leaving it." It prevents situations as pictured in figure 4.

```
:- in_motion(ID, (Y, X), (B, A), Time),
    in_motion(ID2, (B, A), (Y, X), Time),
    ID != ID2.
```

This rule prevents the swap collision (2.4). To avert standard collisions (2.4) we still need to remove all candidate models in which two train share the same cell at the same time. If a train is waiting on a cell, it doesn't have any `in_motion` facts at that time. The following rule addresses this case::

```
:- position(ID, (Y,X), _, Time), position(ID2, (Y,X),_,Time),
    ID != ID2.
```

4 Malfunctions

Train malfunctions are a Flatland feature to simulate spontaneous train breakdowns that cause a malfunctioning train to stop for a random duration in a set time frame.

4.1 Malfunction Context

If a Malfunction rate greater than 0 is set in environment generation, Flatland will interrupt the simulation if a malfunction occurs. This parameter defines how many time steps it takes on average before the next malfunction.

The simulation is restarted with the secondary encoding if one is provided. Otherwise, the primary encoding is run again. In addition to the problem instance, a list of facts is added to the encoding that provide context of past actions, that have already occurred, the present state describing malfunctions and the future actions, that have not been parsed by the Flatland simulation:

```
:- not action(train(2),move_right,57).
:- not action(train(3),move_forward,57).
malfunction(0,4,58).
:- not action(train(0),wait,59).
:- not action(train(0),wait,60).
:- not action(train(0),wait,61).
:- not action(train(0),wait,62).
planned_action(train(0),move_right,0).
planned_action(train(1),move_forward,0).
```

The list of past actions guarantees that the encoding is generating the same actions up to the point of the malfunction. The malfunction fact provides information on the train at which time step is malfunctioning for a set duration. The set of planned actions are all actions that the previous solution would have given after the malfunction.

It is important to note that there is no integrity constraint for an action at the time step of the malfunction in the current ASP-Flatland framework. This means any action is allowed at time step 59. Although this is correct in the Flatland simulation, as this one action is lost and not evaluated by the simulation, it will create a divergence in the ASP simulation. Implementing malfunction support needs to ensure that a wait is forced at this time step.

4.2 Simple Approaches to Handle Malfunctions

As a baseline, the simplest approach is to reuse the default encoding and generate a set of additional integrity constraints that force the train to wait during the malfunction time. This ignores the planned action set and only requires that the encoding can generate wait actions at any point.

If you want to use the planned actions, simply shifting the time of these actions by the duration of the malfunction can work in specific cases. In the trivial case, there is only one train in the environment and no collisions could arise. Alternatively, if the delay of one train applies to all trains, this trivial solution would work as well.

```
{action(train(ID),A,T) : act(A)} = 1
:- T=0..T_malf-1 , malfunction(ID,_,T_malf).
{action(train(ID),wait,T_) : T_=T..T+Dur+1}=Dur+1
:- malfunction(ID,Dur,T).
action(train(ID),Act,T + T_malf + Dur +2)
:- planned_action(train(ID),Act,T), malfunction(ID,Dur,T_malf).
```

If more than one train is in the environment, checks must be performed to determine if the new set of delayed actions would cause collisions. In practice, the pathfinding needs to be re-run, as no information outside of the generated actions is saved.

An encoding that can generate wait actions at any timesteps can handle train malfunctions, as the new output retraces previous steps enforced by the added integrity constraints. The malfunction time window has to enforce wait actions.

4.3 Ideas for Future Implementations

The malfunction feature will have additional ways of saving context information in future updates of the Flatland-Python interface. A possible way to improve performance is to remember which sets of train paths are always collision-free.

This is given, when the cells of a path are not visited more than once. If such a condition is satisfied for a specific train path, the approach of the simple malfunction handler can be used for this path.

5 Graph Based Encoding

The previously used encodings have advantages from their uniform grid, as additional features such as malfunctions and different train speeds are easier to implement. But in this section, we want to explore a graph-based approach and see how it compares to the grid-based encodings.

We want to transform the environment cell data to a graph-based representation of the train network. This opens up the use of graph-based approaches for path finding and further transformation of the graph - for example, removing edges and nodes not required for a solution.

5.1 Transformation of Grid Data to Graph Representation

The cell data is transformed to a Graph G=(V,E) with V containing all track types, that are not straights or curves. All train start and destination positions are also nodes. The graph Edges only include cells interconnected by straight and curve tracks.

Link facts represent two adjacent connected cells. Because ingoing train directions decide if a movement from one cell to the next is allowed, the connection rule is reused here and link facts require ingoing and outgoing directions.

```
link(A,B,Dir_in,Dir_out) :-
    connection(Tr,Dir_in,Dir_out,Action) , ...
```

The body of this rule is similar to that of a train movement in the cell-based encoding, but we forbid possible cell connections based on wait actions.

Paths are chained links from one point in the rail network to another, with the restriction of only including straight and curved track types. Each link from one cell to an adjacent one is counted as one step in the length of the path.

```
path(A,C,Dir_in,L+1) :- link(A,B,Dir_in,Dir_out),
    path(B,C,Dir_out,L) , ...
path(A,B,Dir_in,1)   :- link(A,B,Dir_in,Dir_out).
```

These rules are used to build edges recursively. We do not want to include the nodes themselves in this rule to ensure that the recursion terminates one cell before the next node. For the last step to the node cells, we use edge facts.

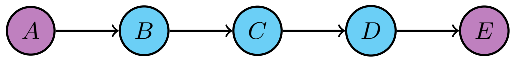


Fig. 5. One edge connecting cells A and E. The inner section is a path

Edges connect nodes, either directly with up to two links or over paths. The facts of the form `edge(A,B,A_in,A_out,B_in,L)` can be generated with three rules:

```
edge(A,B,_,_,_,L+2) :- link(A,A',_,_),~link(B',B,_,_),
    path(A',B',_,L) , ...
edge(A,B,_,_,_,2)   :- link(A,A',_,_),~link(B',B,_,_), ...
edge(A,B,_,_,_,1)   :- link(A,B,_,_), ...
```

These edges are directed and require additional directional information on the start and end nodes. Each edge saves track directions at the ends, so that a single track is represented with at least two edges: A → B , B → A and additional edges if there are multiple input directions.

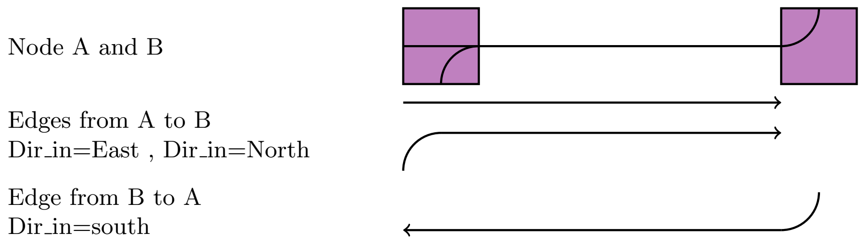


Fig. 6. The pathway of two Nodes A and B is represented with three directed edges

In the following example (5.1), Nodes D and E are connected with two edges in each direction. In the case D → E the outgoing direction from D differs and the ingoing direction is in both cases eastward, while in the case E → D both ingoing and outgoing directions change together.

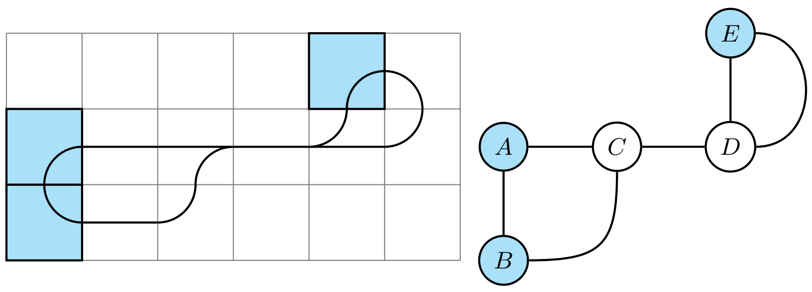


Fig. 7. A simple track and it's graph representation.

5.2 Pathfinding in a Graph

Each train traverses the graph in a similar way as the standard grid-based approach: on each node at a certain time step the train decides for an outgoing edge and the resulting new position is the next node after L time steps plus a wait time.

```
on_node(Train ID,A,Direction,Time,Wait Time)
in_motion(ID,A,A_in,A_out,B_in,B,Start,End,Wait_time)
```

The available outgoing edges are restricted by the direction of the train. This edge provides the next node and information about the length and additional wait time. The `in_motion` rule is used for the selection of an outgoing edge. Compared to the previous grid-based approaches, waiting is more restrictive. Trains do not have a collision check for each cell. Instead of allowing waiting on every cell, waiting is currently only allowed on the last cell on edges before the next node.

Actions are not used and all train movements are a sequence of node positions and chosen edges. Actions can be generated from this sequence.

5.3 Collisions

Collision checking is now required in three different cases:

– Node collisions, where two trains occupy the same node at time T
– Edge collisions, that arise when two trains pass through each other
– Waiting collisions, if two trains travel on the same edge, but waiting occurs

The first type of collision does not differ from the vertex collisions in the cell based approaches and is solved the same way. Edge collisions are registered by comparing the time windows of two trains traveling from node A to node B and the other way around. Both time windows must not have an overlap to avoid a collision.

```
critical_edge(A,B,ID1,ID2,T1,T2,T1',T2') :-
    in_motion(ID1,A,A_in,A_out,B_in,B,T1,T2,Wait_time) ,
    in_motion(ID2,B,B_in',B_out',A_in',A,T1',T2',Wait_time') ,
    mirrored_dir(B_in,B_out')´,´mirrored_dir(A_out,A_in') ,
    ID1!=ID2 , T1<=T1'<=T2.
```

To find two edges that contain the same path with one having the swapped direction, the ingoing and outgoing directions are used to select the correct edge. The direction a train is arriving on it's second node is mirrored and used as the outgoing direction of the train traveling the opposite way. The time intervals in which the trains are traveling on these edges must not overlap.

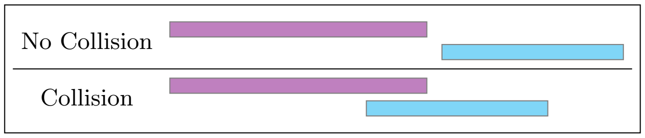


Fig. 8. Two time frames from trains traveling on the same edge in opposite directions. If the time frames of the train traveling from A → B ▶ and the train going from B → A ▶ overlap, an edge collision occurs.

Wait collisions can occur, when two trains use the same edge with overlapping time frames but the train further ahead has a waiting time at the end of the edge. In the figure below, a collision would occur if the violet train waited one additional step. For collisions to be avoided, a train must be in a distance of at least the number of waits from the following train.

```
:-  on_node(ID1,B,B_in,T,W) , on_node(ID2,B,B_in,T',W') ,
    ID1!=ID2 , T>T' , T<=T'+W.
```

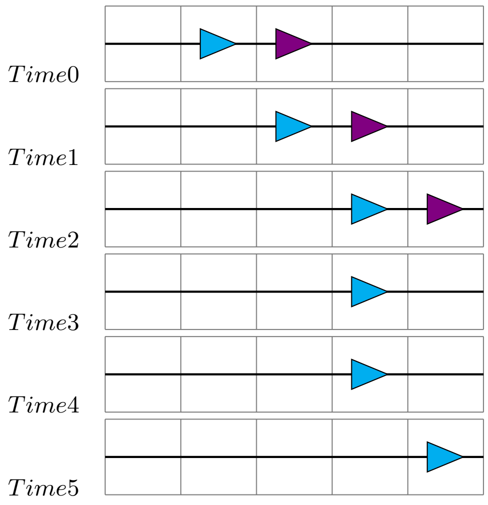


Fig. 9. The first train has a arrival time of 3 with a wait time of 1. The second train needs to arrive one timestep after the first, because it needs to factor in it's own wait time

The time window of allowed wait actions per edge is set to a low amount as it heavily affects the performance of edge collision checking.

While this implementation allows waiting, the restriction of only allowing wait actions on the end cell of an edge makes this encoding not compatible with Malfunctions and omits possible solutions. A less restrictive form of waiting can be done because explicit actions are not used. Instead, only the arrival time at the next node matters. It takes a set amount of forward actions to reach the next node and wait actions need to be placed to ensure trains queue without collisions. In addition, a check must be introduced that restricts the number of trains that use the same edge at overlapping times to ensure that a single edge never holds more trains than its length.

6 Evaluation

In this section, we want to compare the performance of the different encodings. Used are only environments with no speed differences or train malfunctions.

We increase the complexity of test cases by increasing the number of trains for each batch of generated environments.

Encodings can take much longer to solve an instance, depending on whether waiting is allowed or not. The departure delay must also be taken into account. Encodings where the allowed departure delay and wait time are too restrictive might not be able to solve instances.

The following graphs show average time values for a group of 10 environments per number of trains. In figure 10, we compare the `flat_based` and the `graph_based` encoding. On the left side, both encodings do not allow wait actions. The spawn time window is handled the same way with `MinTravelTime`.

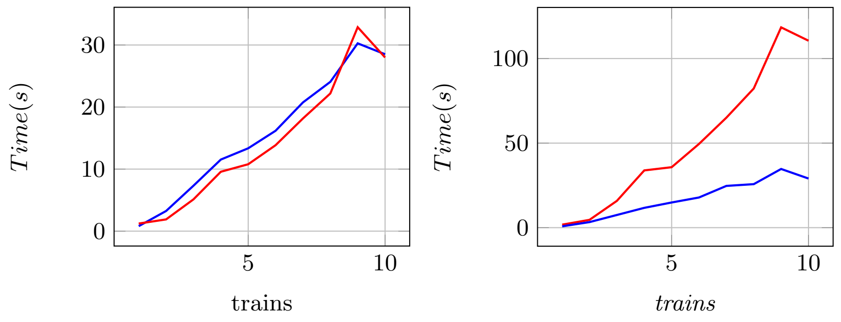


Fig. 10. flat_based in blue and path_based in red. Left side shows no wait actions, right with waiting

On the left side, we can see that both encodings perform similarly if waiting is disabled. On the right side, we compare both encodings again, but this time `flat_based` allows unrestricted wait actions, while `graph_based` uses it's restricted edge waits with a maximal wait time of two steps. It can be seen that the current implementation of edge waiting is inefficient and quickly causes performance problems.

The time used for graph building is significant for specific smaller instances with a low amount of trains on large maps. For larger train values, it becomes negligible compared to the pathfinding time.

7 Tools

We created two tools to gain a better understanding of the performance and behavior of our encodings.

7.1 BatchTest.py

This can be used to run many problem instances with an encoding in parallel. It saves the results of each in a given output folder, and additionally creates a CSV that collects some metrics included in each output, an example is shown in figure 7.1.

Instance	Trains	Stations	Satisfiability	Time (s)	Time Solving(s)	rules	choices
$\text{env}_0 01 - -1_1 6.lp$	1	16	SATISFIABLE	3.91	$6 \cdot 10^{-2}$	$1 \cdot 10^{6}$	1,194
$\text{env}_0 02 - -2_1 6.lp$	2	16	SATISFIABLE	88.66	0.25	$5.59 \cdot 10^{6}$	11,631
$\text{env}_0 03 - -3_1 6.lp$	3	16	SATISFIABLE	7.19	0.35	$9.56 \cdot 10^{5}$	5,249
$\text{env}_0 04 - -4_1 6.lp$	4	16	SATISFIABLE	166.18	0.23	$8.06 \cdot 10^{6}$	10,586
$\text{env}_0 05 - -5_1 6.lp$	5	16	SATISFIABLE	37.24	0.1	$2.94 \cdot 10^{6}$	4,656

Fig. 11. An example output of batch_test.py

These metrics can be used to analyze and compare encodings as seen in the previous section. Currently, this tool only interacts with clingo and does not automatically generate environments. This also means the correctness of encodings can not be guaranteed and seperate tests in Flatland are still nessesary.

7.2 Plot.py

When testing our encodings, we often faced the problem that we could not easily verify our results. Especially Flatland environment instances start with a minimum size of 30 x 30 cells. Looking at clingo output for such large maps is rather tiring.
The Python script `plot.py` takes two arguments: a clingo encoding as the first and a Flatland environment as the second, both provided as `.lp` files. The order is important. It then solves as many models as specified (default: `-n 0`, meaning all models). In the encoding, train positions need to be encoded using this fact format: `pos(Train_ID, (Y, X), Time)`. The script visualizes all `pos` facts for every train across all models as a heatmap using Matplotlib. An example output is shown in figure 12, where darker color tones indicate more recorded train

positions. This visualization tool has helped us many times, allowing to see the difference between the Flatland simulation output and clingo output.

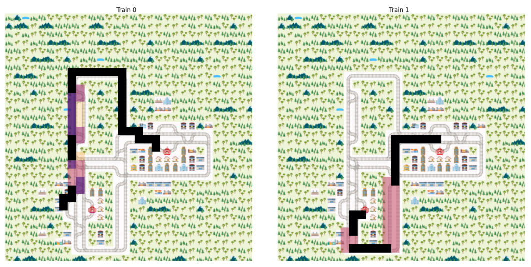


Fig. 12. Example Output of plot.py with 2 Trains

8 Drawbacks and Future Outlook

Throughout this project, we faced several challenges. Some were due to our limited experience at the start of the semester, while others came from the new and complex nature of the project. Parts of the Python interface connecting Flatland and clingo had not been widely tested, which led to some unexpected issues. In this chapter, we want to highlight some of these problems and suggest ideas for future research.
At first, some Flatland details were unclear, such as how trains spawn at the zero time step. This became even harder to understand because the output CSV was misaligned, showing actions at the wrong time steps. Debugging and verifying our models was also quite difficult. The success of a model was determined by the `done` states of trains in the output CSV, but these were only recorded if an action was issued after the train had already arrived. This added extra restrictions to our encodings. To help with this, we proposed a fix by suggesting that the final train states should always be displayed. We submitted this as a GitHub issue. Sometimes, we also overlooked certain coding details. For example, overloading the action fact was not allowed, which could cause incorrect action lists. We suggested a simple check (`len(atom.arguments) == 3`) to enable ASP overloading while preventing errors. Another issue was that malfunctions could make instances unsolvable after a few simulation interruptions, especially if the malfunction rate was too high. The exact reason for this is still unclear.
We hope that future research will explore graph-based abstractions of the Flatland environment in more depth and analyze their impact on performance. Our `flat_based` encoding includes speed functionality, but we have just started scratching the surface of how speed and malfunctions can be handled together in the same encoding. Despite these challenges, this project has been a valuable and fun learning experience. We look forward to seeing how future research improves these ASP railway scheduling solutions.

References

1. Flatland framework (2025), `https://github.com/flatland-association/flatland-rl`, Flatland Association, [Accessed: 05.03.2025]
2. Potassco, the Potsdam Answer Set Solving Collection (2025), `https://potassco.org/`, University of Potsdam, [Accessed: 05.03.2025]
3. Mohanty, S.P., Nygren, E., Laurent, F., Schneider, M., Scheller, C., Bhattacharya, N., Watson, J.D., Egli, A., Eichenberger, C., Baumberger, C., Vienken, G., Sturm, I., Sartoretti, G., Spigler, G.: Flatland-rl : Multi-agent reinforcement learning on trains. CoRR **abs/2012.05893** (2020), `https://arxiv.org/abs/2012.05893`