Alma Mater Studiorum · University of Bologna

Flatland Challenge

Deep learning course final project

Leonardo Calbi (leonardo.calbi@studio.unibo.it)
Alessio Falai (alessio.falai@studio.unibo.it)

December 18, 2020

Contents

List of Figures

Foreword

The Flatland challenge is a competition organized by AIcrowd [1] with the help of SBB (Swiss Federal Railways) to foster innovation with what regards the scheduling of trains trajectories in a railway environment.

As reported on the official challenge website, SBB operates the densest mixed railway traffic in the world. It maintains and operates the biggest railway infrastructure in Switzerland: today, as of 2020, there are more than 10 000 trains running each day, being routed over 13 000 switches and controlled by more than 32 000 signals.

The Flatland challenge aims to address the vehicle rescheduling problem by providing a simplistic grid world environment and allowing for diverse solution approaches. In particular, the first edition of the challenge was hosted during 2019 and the submitted solutions were mainly based on OR (Operation Research) methodologies, while the second edition of the competition, i.e. the NeurIPS 2020 edition, had the goal of favoring the implementation of RL (Reinforcement Learning) based solutions.

1 Introduction

At the core of this challenge lies the general vehicle rescheduling problem (VRSP) proposed by Li, Mirchandani, and Borenstein in 2007 [3]:

> *The vehicle rescheduling problem (VRSP) arises when a previously assigned trip is disrupted. A traffic accident, a medical emergency, or a breakdown of a vehicle are examples of possible disruptions that demand the rescheduling of vehicle trips. The VRSP can be approached as a dynamic version of the classical vehicle scheduling problem (VSP) where assignments are generated dynamically.*

The problem is formulated as a 2D grid environment with restricted transitions between neighboring cells to represent railway networks. On the 2D grid, multiple agents with different objectives must collaborate to maximize the global reward.

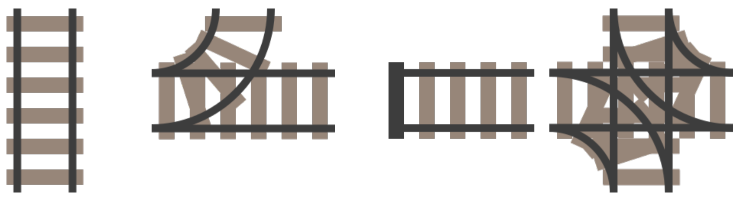
The overall goal is to make all agents (trains) arrive at their target destination with a minimal travel time. In other words, we want to minimize the time steps (or wait time) that it takes for each agent in the group to reach its destination.

2 Background

Railway

As already pointed out, the Flatland environment is represented as a 2D grid of dimension $W \times H$ and each cell in the grid can be one of many different types. The different types of cells can belong to the following categories: `rail` and `empty`.

The `rail` cells are the most intricated of the two, in that there exists different types of them. In particular, figure 2.1 shows examples of possible `rail` cells that can be used to build up a railway environment in Flatland. Other than the ones shown in figure 2.1 there are also diamond crossings (i.e. two orthogonal straight rails crossing each other), single slip switches (i.e. the same as double slip switches but with a single choice) and symmetrical switches (which are special kinds of switches that bifurcate to a left and right branch). Moreover, every `rail` cell can be rotated by 90° and mirrored along both axis, to allow more combinations between them to be made, in order to guarantee a greater degree of diversity between different railways.



(a) Straight (b) Simple switch (c) Deadend (d) Double slip switch

Figure 2.1: Different `rail` cell types

Moreover, `rail` cells can be occupied by the following entities (the ones shown in figure 2.2):

- Agent: one `rail` cell can be seen as a resource with availability equal to one, so that in each time step only one agent can occupy it
- Target: each target is statically assigned to one `rail` cell. Target cells represent the destination of one or more agents (different agents could have the same target). Moreover, the number of possible targets present in the environment is clearly limited by the number of agents

An important fact about the different types of `rail` cells is that only switches require an agent to make a choice. In Flatland (like in reality) a maximum of two options is available. There does not exist a switch with three or more options.

Finally, every cell that is not `rail` is `empty` and neither targets nor agents can

(a) Agent (b) Target

Figure 2.2: Agents and targets

fill it up. As shown in figure 2.3, it is interesting to notice that Flatland is a very sparse environment, meaning that there are a lot more `empty` cells than `rail` ones: because of this, representing the environment as a simple dense matrix could lead to overheads and efficiency issues, especially when dealing with relatively big environments.

In the end, the only cell types that we care about are the `rail` ones: the `empty` ones are useful only for visualization purposes. Because of this, we could think of representing the environment as a sparse matrix or as a graph containing only `rail` cells or a subset of them (we will address this issue in section 3.1).

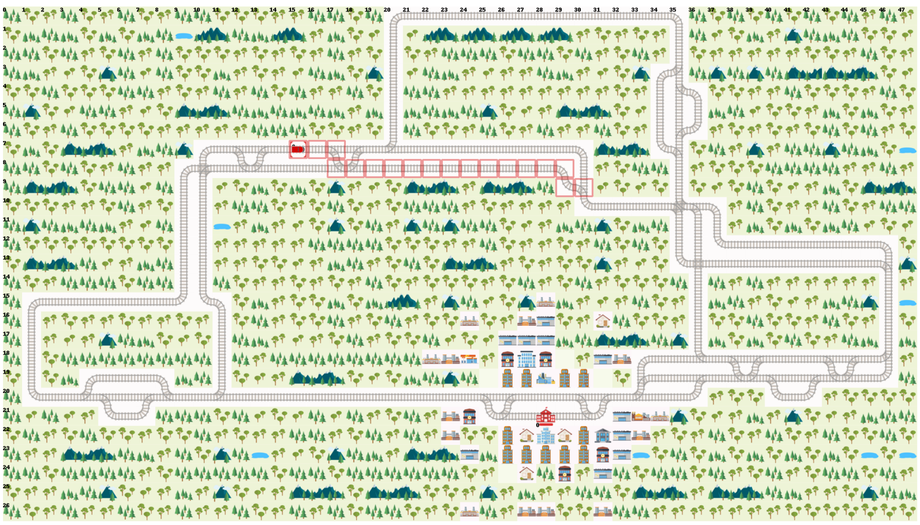


Figure 2.3: An example of a railway environment

Transitions

An agent in the Flatland environment is a train that starts from a random `rail` cell in the map and has to arrive to its assigned target in the minimum number of steps. To do so, the agent can only occupy `rail` cells.

To move from a cell to another one the agent has to make a choice and, depending on the cell type that they are on and on the connections between cells, an agent can transition from cell i, when looking towards direction d_i, to cell j, looking towards direction d_j, if and only if $T_i(d_i, d_j) = 1$, where T_i is the transition matrix associated to cell i, s.t. $T_i(d_i, d_j) = 0$ means that the transition from cell i, direction d_i to cell j, direction d_j is forbidden (likewise $T_i(d_i, d_j) = 1$ means that the transition is allowed). Directions d_* are represented as the 4 cardinal directions, i.e. North, East, South and West (N, E, S, W), so that each transition matrix T_* can be characterized as a 4×4 binary matrix. For example, a deadend cell like the one reported in figure 2.1c would have a transition matrix like the one shown in figure 2.4.

$$
\begin{array}{c c}
 & \begin{array}{cccc} N & E & S & W \end{array} \\
\begin{array}{c} N \\ E \\ S \\ W \end{array} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}
\end{array}
\xrightarrow[\text{bitmap}]{\text{to}}
\quad 0000\ 000\mathbf{1}\ 0000\ 0000
$$

Figure 2.4: Transition matrix and bitmap of a deadend

As we can observe, only one entry in the matrix has value 1, meaning that only one transition is possible, i.e. the one s.t. the agent enters heading East and exits heading West.

In the Flatland library, a transition matrix is represented by a bitmap, which can be seen as a linearization by rows of the reported matrix (the mapping between the transition matrix of the deadend cell 2.1c and its bitmap is again shown in figure 2.4). In this way, by simply counting the number of true values in the bitmap, we can understand the type of `rail` cell that we are examining (e.g. only one true value indicates a deadend, while exactly two true values indicate a straight rail).

Actions

Flatland has a discrete action space, meaning that only 5 possibilities have to be considered at each transition. In particular, Flatland uses the following convention:

1. `MOVE_FORWARD`: the agent maintains the current movement direction, if possible (i.e. if it was heading North, it will continue heading North)

2. `MOVE_LEFT`: if the agent is at a switch with a transition to its left, the agent will choose the left path, otherwise the action has no effect (e.g. if the agent was heading North, it will be directed towards East)

3. `MOVE_RIGHT`: if the agent is at a switch with a transition to its right, the agent will choose the right path, otherwise the action has no effect (e.g. if the agent was heading North, it will be directed towards West)

4. `STOP_MOVING`: the agent remains in the same cell

5. `DO_NOTHING`: the agent performs the same action as the last time step

Usually, only a handful of the reported actions can be perfomed on a given cell, meaning that most of the times the actual number of choices is much less than 5 (we will address this issue in section 3.3).

Complications

The main complications of the Flatland challenge are given by speed profiles, conflicts and malfunctions. In particular, about speed profiles, each and every agent could have a different velocity. The standard speed (and the maximum one) is 1, which means that the agent crosses one cell in one time step. Speeds can have values in range $(0, 1]$: if an agent has speed s, it means that it needs $\lceil \frac{1}{s} \rceil$ time steps to transition from one cell to the next one.

Speeds are assigned to agents based on a custom probability mass function, so that $P(S = s)$ represents the probability that the speed S of an agent is equal to s. For example, we could have the following *pmf* (representing a uniform distribution over values $\{\frac{1}{4}, \frac{1}{3}, \frac{1}{2}, 1\}$):

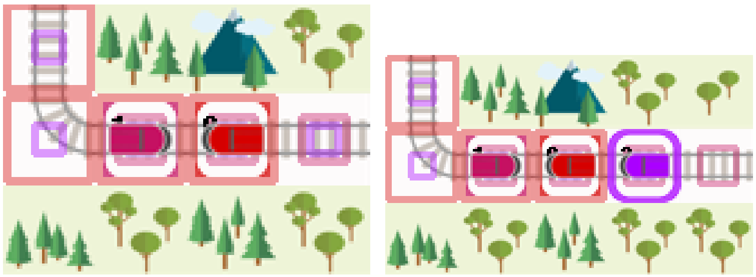
$$\begin{cases} P(S = s) = 0.25 & \text{if } s \in \{\frac{1}{4}, \frac{1}{3}, \frac{1}{2}, 1\} \\ P(S = s) = 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Clearly, the speed factor is a critical one to observe when trying to minimize the total number of time steps in a multi-agent scenario: for example, different agents have to understand that faster ones should go first if a decision has to be made. This is because we would like to avoid bottlenecks of any kind inside the railway network and slower agents can definitely become an issue in an environment with relatively long sequences of straight rails.

About the second complications, i.e. malfunctions, agents can experience defects or failures which do not let them go on in their path towards the target. A malfunction in the Flatland environment is modeled by a Poisson distribution $1 - P_\lambda(n = 0) = 1 - \frac{\lambda^n}{n!} \cdot e^{-\lambda} = 1 - e^{-\lambda}$, where $\lambda \in (0, 1]$ is the malfunction rate s.t. $\frac{1}{\lambda}$ represents the mean frequency of occurrence of malfunctioning events. For example, if malfunctions are to be expected once every 80 time steps for an agent, then $\lambda = \frac{1}{80} = 0.0125$ and the probability of a malfunction in each time step is equal to $1 - e^{-0.0125} = 0.01$, while if malfunctions are more rare (e.g. once every 200 time steps), then we would have a probability value of $1 - e^{-0.005} = 0.004$. Once an agent is malfunctioning, it stays so for a random number of time steps, bounded by parameters indicating the minimum and maximum duration of a malfunction period.

Again, the malfunction factor is critical, in that it could prevent one or more agents from reaching their targets in the minimum number of time steps. In this way, transitions in the environment become stochastic, thus leading to a slower and much more difficult optimization procedure.

About the third complication, i.e. conflicts, we can define it as a state in which agents cannot perform any action, because they are "blocked" by one or more neighboring agents. Different conflicting situations can arise in practice, where the most basic one is given by two agents heading in different directions in a sequence of straight rails (see figure 2.5a): this situation can in turn cause other deadlocks to happen, as shown in figure 2.5b.



(a) Two agents (b) Three agents

Figure 2.5: Examples of deadlocks

A full deadlock situation is a state in which every agent is in deadlock: in that case, no agent would be able to arrive at its target. These blocking situations are one of the most intricate and complex factors that should be addressed when designed autonomous agents in the Flatland environment, since they are quite frequent (and they get more frequent as the number of agents increases and the dimension of the grid decreases) and have disastrous consequences (at least in a real-world scenario).

Differently from malfunctions, deadlocks totally prevent agents from reaching their destination: when a malfunction occurs, agents that could previously reach their targets will still be able to reach them once the malfunction period is over, while when a conflict occurs, at least two agents will never arrive at their targets.

3 Environment

In this chapter we are going to explore better ways to encode the Flatland environment and how agents can interact with these representations.

3.1 Railway encoding

As already described in chapter 2, the default Flatland representation was not built with the goal of efficiency in mind, since it stores each and every cell of the 2D grid, while only `rail` cells are the ones that are actually used by agents. An alternative and more efficient representation could be to use some kind of sparse matrix implementation, where `empty` cells are not stored at all, but this would only be beneficial from the point of view of memory occupancy, while the usage of more specific data structures could also improve other aspects, like the computation of shortest paths from the agent's position to its target.

In this work, we decided to rely on a graph structure. In particular, we were inspired by the great work described in [2] where the author presents the so called Cell Orientation Graph (COG), in which nodes represent cells in the 2D grid as a triple (x, y, d), where (x, y) are the coordinates in the grid (origin in the top-left corner with x-axis looking right and y-axis looking down) and d is one of the four cardinal directions (representing the direction of entrance in the cell). In this way, one `rail` cell is represented by at most 4 nodes in the graph: this could seem like a major drawback, but in practice we can observe that the number of nodes is roughly equivalent to the number of cells in the corresponding grid, since we got rid of the `empty` ones.

Instead, edges in the graph are directed and represent legal transitions, so that no transition matrix or bitmap has to be stored in each node: it is simply encoded in the topology of the network. Moreover, the usage of directed edges greatly simplifies the computation of paths between nodes in the graph.

In order to further simplify the representation of the Flatland environment, we decided to entirely delete nodes which represented straight rails (with the exception of keeping straight rails containing targets and deadends). In this way, what we end up with is something that could be defined a Cell Orientation Junction Graph (COJG), since the remaining nodes are the ones in which an agent either has to make a decision or finishes its trip.

Because of the deletion of almost all nodes associated to straight rails, edges actually represent a connection between an interesting cell and another one (e.g. they link a junction to a target or a junction to a second junction). In order to maintain the same topology as before, the number of deleted straight rails between each pair of interesting nodes is used as the weight of the edge connecting them, so that the computation of shortest paths can automatically take that into account.

Figure 3.1 shows the comparison, in terms of visual representations, of the standard grid environment and both COG and COJG graphs, in a 32×16 map. In table 3.1 we show the improvements that can be gained by leveraging the usage of the Cell Orientation Graph, and in particular of its modified version, i.e. the Cell Orientation Junction Graph, by computing the number of nodes, edges and `empty` cells in each representation, in the same 32×16 map of figure 3.1.

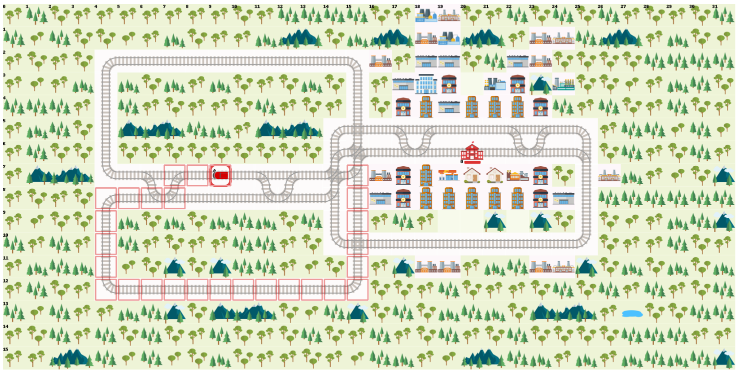
	Nodes	**Edges**	**Empty**
Grid	512	-	413
COG	226	254	0
COJG	78	106	0

Table 3.1: Grid, COG, COJG comparison in a 32×16 map

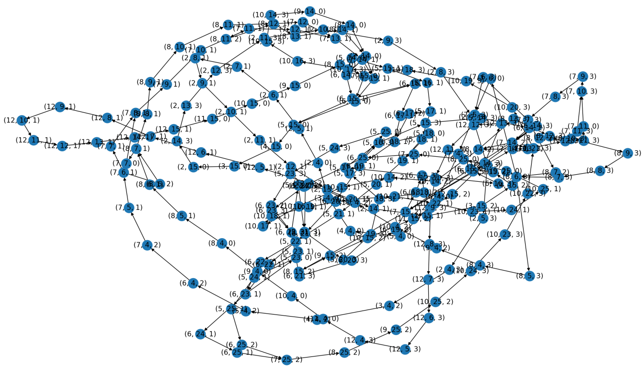
Since a $32x16$ map is too small to observe big improvements, in figure 3.2 we also report another example of a sparse grid environment of dimension 128×64 and we compare it with its encodings in table 3.2.

	Nodes	**Edges**	**Empty**
Grid	8192	-	7732
COG	1050	1096	0
COJG	136	182	0

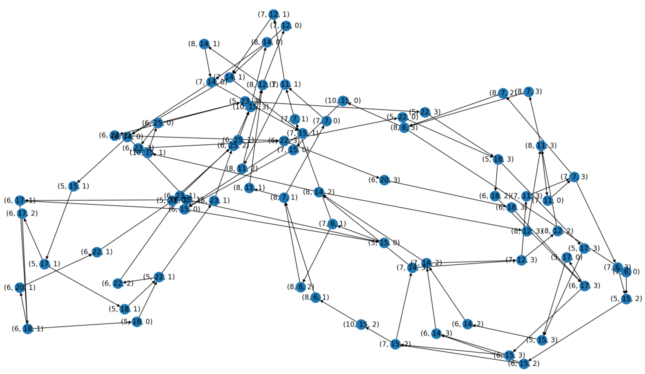
Table 3.2: Grid, COG, COJG comparison in a 128×64 map



(a) Grid

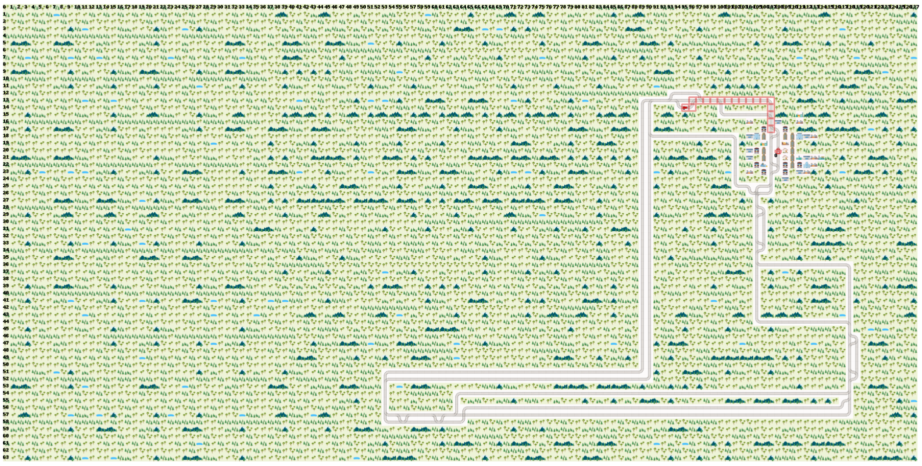


(b) COG

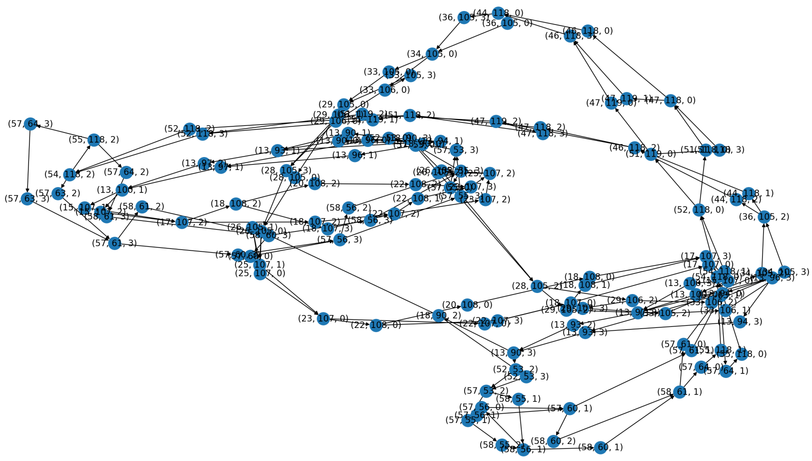


(c) COJG

Figure 3.1: Different railway encodings of a 32×16 map



(a) Grid



(b) COJG

Figure 3.2: Grid and COJG comparison in a sparse 128×64 map

3.2 Predictions

As reported in Flatland official FAQs, because railway traffic is limited to rails, many decisions that an agent has to take need to consider future situations and detect upcoming conflicts ahead of time. Therefore, Flatland provides the possibility of predictors that predict where agents will be in the future. The stock predictor simply assumes that each agent travels along its shortest path.

3.2.1 Shortest and deviation paths

One of the key advantages of COJG is that it enables us to apply standard shortest path algorithms on directed weighted graphs without any kind of overhead. In particular, we decided to use Dijkstra's algorithm since we do not have negatively weighted edges (in that case, Dijkstra's algorithm could get stuck in a cycle containing at least one edge with negative weight, since it may cycle an infinite number of times and sometimes go down with the total cost as much as it likes).

To build our predictor, we took inspiration from the standard predictor given by the Flatland library as a baseline, which computes shortest paths directly on the 2D grid representation. Our predictor instead takes into account two possibilities: an agent either follows its shortest path or deviates from it. The computation of shortest paths is incremental, meaning that once it is done it only gets updated (i.e. the first node is removed). The only event that causes a re-computation of the shortest path is when the agent makes a choice that does not follow the stored path.

Deviation paths are instead computed on the fly and they represent alternatives from each node of the shortest path. The computation of deviation path i, given the shortest path $n_1, \ldots, n_i, n_{i+1}, \ldots, n_d$, is simply another call to the Dijkstra's routine, where edge (n_i, n_{i+1}) is forbidden. In this example, d represents the maximum depth of both shortest and deviation paths.

In this way, the prediction becomes a list of paths, where the first one is the shortest path, composed by at most d nodes, and the following ones are $d - 1$ deviation paths, still composed by at most d nodes. When an agent is relatively close to its target (i.e. less than d nodes away from it), the actual path could have a dimension which is less than d: to avoid troubles when dealing with dynamically sized vectors, paths are padded to the full depth with special symbols.

In case an agent cannot reach its target anymore, the prediction only comprises a path of two nodes, i.e. the agent's position and the next node in the COJG graph. This mini-path is stored because predictions are used to compute possible deadlocks and, even if an agent cannot arrive at its target anymore, it is still present in the railway environment and it could be a possible source of conflicts.

3.3 Real decisions and choices

3.3.1 Real decisions

As already hinted in the previous sections, Flatland railway environments are structured in such a way that most of them are composed by long sequences of straight rails, where an agent does not have to commit to any thoughtful choice, but it simply has to follow the path on which it is positioned. The agent is instead required to make a decision in two simple cases: whenever it reaches a fork, i.e. a cell that has more than one successor, or immediately before a join, i.e. before entering another path that crosses its next cell.

Following the reasoning described in [2], we categorize as real decisions those times in which an agent has to select an action when it is positioned in a fork, before a join or in a combination of the two cases. An agent in a fork has to decide on which branch to go next (and ideally, it should not choose to stop on that cell), while an agent before a join has to decide if it wants to stop (maybe to let another agent pass through the join cell) or continue moving.

This idea of introducing real decisions in the Flatland environment greatly reduces the number of calls to the model that is tasked to map states to actions, since the number of real decisions to take is much lower than the number of default actions that one has to provide in the standard framework. In particular, the default number of actions to select is equal to the number of steps that the agent performs in the environment, while the number of real decisions to take is entirely disentagled from the steps parameter, since it is completely dependent on the structure of the environment (e.g. sparser railways tend to have less real decisions w.r.t. denser ones, given the same grid size).

The concept of real decisions leads to a modification in the structure of the COJG graph reported in section 3.1, so that both types of decisions are encoded as nodes. In particular, COJG is missing most nodes related to cells before a join, since they are usually part of straight rails. Currently, our implementation patches the COJG graph, but a cleaner choice would be to actually add the before join nodes from the start of the building process, which could be something to try for future improvements.

3.3.2 Choices

Further analysis related to decisions, again presented in [2], shows a smart action space reduction that could be beneficial to reinforcement learning models. In particular, Flatland uses a discrete action space composed of 5 actions, which can be reduced to a total of 3. From now on, we will refer to elements of this new action space as "choices".

Choices are mapped in such a way that they correspond to the left-most and right-most directions available to the agent at each time-step. In this way, our model only has to predict whether the agent has to follow the track to its left or the track to its right or simply stop. After a choice is selected, it is mapped back to the initial action space by looking at the type of cell the agent is currently on, to ensure compatibility with the Flatland environment.

The new action space is composed by the choices CHOICE_LEFT, CHOICE_RIGHT and STOP.

So, we have to specify two different mappings, the forward and the backward one. The action to choice mapping works as follows:

1. If the selected action is MOVE_LEFT and that action is legal from the current cell of the agent, then return CHOICE_LEFT
2. If the selected action is MOVE_RIGHT and that action is legal from the current cell of the agent, then
   (a) If only MOVE_RIGHT can be perfomed, then CHOICE_LEFT is returned
   (b) Otherwise, CHOICE_RIGHT is returned
3. If the selected action is MOVE_FORWARD and that action is legal from the current cell of the agent, then
   (a) If MOVE_LEFT can be perfomed, then CHOICE_RIGHT is returned
   (b) Otherwise, if MOVE_RIGHT can be perfomed, then CHOICE_LEFT is returned
   (c) Otherwise, the last resort is CHOICE_LEFT
4. If STOP_MOVING, we have a one-to-one mapping with the STOP choice, so that the agent decides to stand still

Instead, the choice to action mapping works as follows:

1. If CHOICE_LEFT, then priorities are MOVE_LEFT, MOVE_FORWARD, MOVE_RIGHT
2. If CHOICE_RIGHT, then priorities are MOVE_RIGHT, MOVE_FORWARD
3. If STOP, then the only possible outcome is STOP_MOVING
4. Otherwise, last resort is DO_NOTHING

In the previous bullet points, the word "priority" represents the sequential way in which actions should be selected. For example, if a CHOICE_RIGHT has to be mapped back and MOVE_RIGHT is a legal action from the current cell of the agent, then that action would be selected, otherwise MOVE_FORWARD would be chosen (again, if legal).

3.4 Observations

An observation is a representation of the environment in a particular state. Ideally, this snapshot should be rich enough to convey the right amount of information (so as to distinguish between different configurations), but also small enough that it would be efficient to compute at any given time-step. An observation should also be able to integrate information from local (in our case a train) and global (in our case the grid and all the agents) perspectives.

Flatland provides different observators out of the box (the ones roughly depicted in figure 3.3). In particular, the implemented global observation is composed of the following elements:

- Transition map tensor with shape $(h, w, 16)$, assuming 16 bits encoding of transitions
- A tensor with shape $(h, w, 2)$, containing the position of the given agent's target in the first channel and the positions of the other agents' targets in the second channel (flag only, no counter)
- A tensor of shape $(h, w, 5)$ with the following channels:
  1. Agents position and direction
  2. Other agents positions and direction
  3. Agents malfunctions
  4. Agents fractional speeds
  5. Number of other agents ready to depart

The implemented local observation should instead gather information of the rail environment around the given agent. This observation is composed of the following elements:

- Transition map tensor of the local environment around the given agent, with shape $(v_h, 2 \cdot v_w + 1, 16)$, assuming 16 bits encoding of transitions
- A tensor with shape $(v_h, 2 \cdot v_w + 1, 2)$ containing the target position of the given agent (if in the agent's vision range) in the first channel and the positions of the other agents' targets (again, if in the agent's vision range) in the second channel
- A tensor $(v_h, 2 \cdot v_w + 1, 4)$ containing the one-hot encoding of directions of the other agents at their position coordinates (if in the agent's vision range)
- A 4 elements array with a one-hot encoding of the current direction of the given agent

Here, the parameters v_w and v_h define the rectangular view of the agent (of dimension $(2 \cdot v_w + 1) \times v_h$) around its position. One thing to notice about the standard local observation is that it does not contain any clues about the target location, if it's out of range: thus, navigation on maps where the radius of the observation does not guarantee a visible target at all times will become very difficult.

Moreover, Flatland discourages the use of both the given global and local observations, since the first one does not seem to encode enough information to be effective in a multi-agent scenario, while the second one is simply deprecated in the newest version of their Python library, which makes it practically useless. Because of this, more advanced observations must be used in order to achieve good results on a wide range of environments. In particular, in the following sections we will take a look at another observation present in Flatland library, i.e. the tree observation, and we will use it as the starting point to build our custom binary tree observation, which also integrates all the knowledge and findings that we described in the previous chapters.

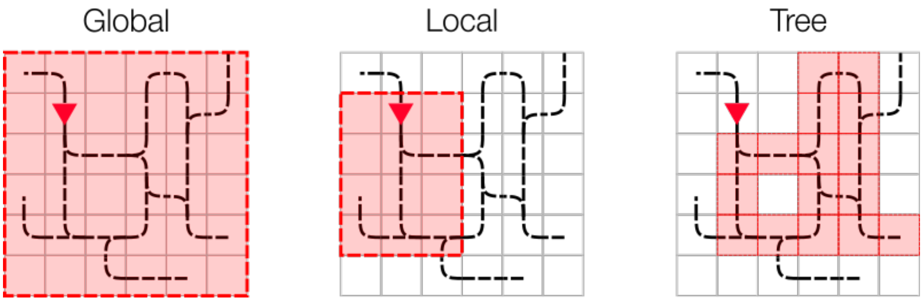


Figure 3.3: Default observators

3.4.1 Tree

The tree observation is the last one provided out of the box in the Flatland package and it's definitely the most effective overall (w.r.t. the already mentioned global and local ones). This observator builds a tree with arity 4, s.t. the children of a node always follow a left, forward, right and backward ordering. In particular, a node in the tree represents a deadend, a switch, a target destination or a blank (meaning that no information is carried by that specific node). The children of a node represent its successor nodes when following the left/right/forward branch on a switch or the backward one on a deadend. The mentioned movements are sorted relative to the current orientation of the agent, rather than using standard transition directions (i.e. the four cardinal directions). Whenever a node does not have one of the mentioned successors, that branch becomes useless and gets filled with padding values.

Figure 3.4 shows a mapping between a state and the corresponding tree observation, from the point of view of the agent represented by the red triangle. The colors in the figure illustrate what branch the cell belongs to (red means left, pink means forward, green means right and blue means backward). If there are multiple colors in a cell, then this cell will be present multiple times in the tree. The symbols in the figure have instead the following meaning:

- Cross: no branch was built, i.e. the parent does not have a valid transition to represent the child
- No children: the node is a terminal node (i.e. a deadend, a cell without possible transitions or a leaf of the tree, indicating that the maximum depth was reached)
- Circle: a node filled with information

Whenever a node is valid, it will be packed with the following features:

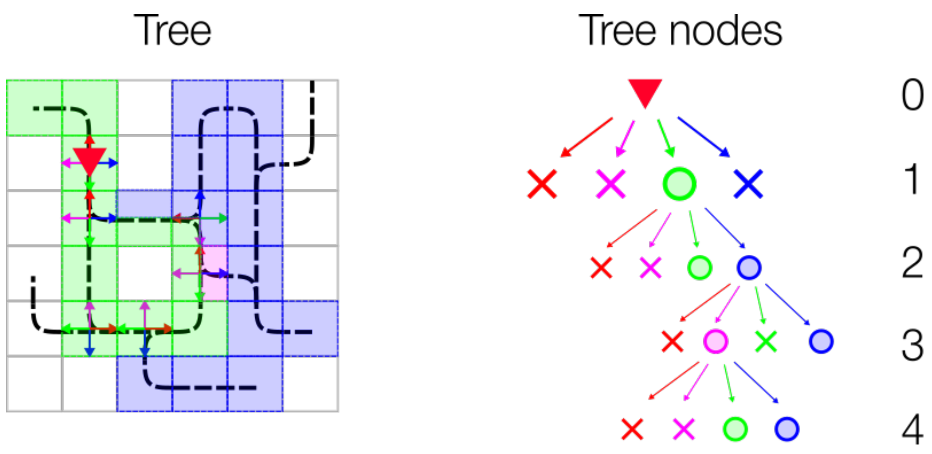


Figure 3.4: Tree observator

1. If the target of the given agent lies on the explored branch, the current distance from the given agent (in number of cells) is stored
2. If another agent's target is detected, the distance in number of cells from the given agent's current location is stored
3. If another agent is detected, the distance in number of cells from the given agent's position is stored
4. If another agent predicts to pass along this cell at the same time as the given agent, the distance (in number of cells) from the given agent's position is stored
5. If an unusable switch is detected for the given agent, the current distance (in number of cells) from the given agent's position is stored
6. Distance in number of cells to the next branching
7. Minimum distance from the node to the given agent's target, considering the current direction of the agent
8. Number of agents present in the same direction as the given agent
9. Number of agents present in the opposite direction as the given agent
10. Number of time steps that the given agent remains blocked, if a malfunctioning agent is encountered
11. Slowest observed speed of agents in the same direction as the given agent
12. Number of agents ready to depart, but not yet active

One key issue related to the described tree observation is related to both time and memory efficiency. In particular, we mentioned a maximum depth parameter that controls how far the observation should reach, in terms of consecutive switches, targets and deadends visited. Clearly, since each node has

4 children and the output of the observator should be consistent, running it requires to build a complete tree, which we know to have a number of nodes equal to:

$$n = \sum_{i=0}^{d-1} a^i,$$

where d is the height of the tree (i.e. our maximum depth) and the arity a is equal to 4. Just to make a quick example, if $d = 7$ then $n = 4^0 + 4^1 + 4^2 + 4^3 + 4^4 + 4^5 + 4^6 = 5461$. In this way, in the worst case, for each time-step we would have to examine $t \cdot n$ nodes, where t is the number of trains, since one observation is needed for each agent. Following the previous example, in a scenario with $t = 10$ agents and a tree observation with depth $d = 7$, the reported features should be computed for each of the $5641 \cdot 10 = 56\,410$ analyzed nodes at every time-step.

3.4.2 Binary tree

In order to exploit what we've built so far and have an hopefully more efficient computation of each observation, a new observator was created, starting from the same notions described in section 3.4.1.

In particular, our observator builds a feature tensor of shape (d, d, f), where d is the maximum number of hops in the COJG graph to consider and f is the total amount of features for each node. This tensor contains the features of the nodes in the shortest path as the first row and the features of the nodes in the deviation paths (which are exactly $d - 1$) as the following rows.

Each node has the following features:

1. Number of agents (going in the same direction as the one of the given agent) identified in the subpath from the root up to each node in the path
2. Number of agents (going in a direction different from the one of the given agent) identified in the subpath from the root up to each node in the path
3. Number of malfunctioning agents (going in the same direction as the one of the given agent) identified in the subpath from the root up to each node in the path
4. Number of malfunctioning agents (going in a direction different from the one of the given agent) identified in the subpath from the root up to each node in the path
5. Minimum distances from an agent to other agents (going in the same direction as the one of the given agent), in each edge of the path
6. Minimum distances from an agent to other agents (going in a direction different from the one of the given agent), in each edge of the path
7. Maximum number of malfunctioning turns of other agents (going in the same direction as the one of the given agent), in each edge of the path

8. Maximum number of malfunctioning turns of other agents (going in a direction different from the one of the given agent), in each edge of the path
9. Distances from the target, from each node in the path
10. Path weights (in number of turns) to reach the given node from the root one
11. Number of agents using the node to reach their target in the shortest path
12. Number of agents in deadlock in the previous path, assuming that all the other agents follow their shortest path
13. How many turns before a possible deadlock
14. If the node is a fork or not
15. How many turns the current agent has been repeatedly selecting the stop action

Initially, we tried to use the feature tensor as our final observation. In particular, we linearized it as a $d \times d \times f$ vector and fed it into one of the neural models that will be described in the next chapters. Clearly, the model was not converging, since we were ignoring the fact that different inputs for neural networks should be consistent, meaning that each and every input neuron should always represent the same measure. Instead, since our feature tensor was built by simply considering paths in the environment, one specific node could have represented either the left or the right branch being taken from the previous node, in different runs. This was definitely confusing the agents, so that no convergence was reached.

Because of this, we resorted to the same idea that was presented in 3.4.1, but, thanks to the new action space introduced in 3.3.2, we were able to shape the feature tensor like a binary tree (i.e. a tree with arity $a = 2$), where branches identify choices, i.e. `CHOICE_LEFT` or `CHOICE_RIGHT`.

Since the resulting tree has half the arity of the one given by Flatland, but we also need a complete tree for consistency, let's consider again the example reported in 3.4.1 to roughly check for efficiency improvements, where a maximum depth of $d = 7$ was chosen, with $t = 10$ trains. Our complete binary tree would have $n = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 127$ nodes (compared to 5641) and the number of total nodes for which features would have to be computed at each time-step would be around $127 \cdot 10 = 1270$ (compared to 56 410). Moreover, the reported number (1270) is an upper bound on the actual number of nodes that should be analyzed, since different time-steps will require different agents to compute an observation, i.e. only those agents that find themselves in a real decision cell.

To sum up efficiency improvements, each feature is built from scratch, by leveraging the performance gains obtained with the COJG graph. Moreover, the observation needs to be computed only when the agent faces a real decision (as described in section 3.3.1), since in every other scenario the action that the agent should take is implicit. By combining this with the much lower size of the tree (compared to Flatland's implementation), we are able to compute more observations with less resources overall. Table 3.3 shows a comparison of the standard tree observation and our custom binary tree one, in terms of both running times and memory occupancy for a single observation. As we can see, the binary tree observation is around 6 times faster on average and occupies 30 times less memory on average than the tree one.

	Time (s)		**Memory (MB)**	
	Avg	**Std**	**Avg**	**Std**
Tree	0.1978	0.0199	0.4806	0.0
Binary tree	0.0335	0.0124	0.0153	0.0

Table 3.3: Tree vs. binary tree efficiency in a 48×27 map, with $d = 7$

DIRE PATH NULLI PER VIA DELLA COSTRUZIONE DAL DEVIATION PARLARE DELLA NORMALIZZAZIONE

3.5 Rewards shaping

4 Policy

4.1 Action masking

4.2 Action selection

4.2.1 ϵ-greedy

4.2.2 Boltzmann

4.3 Replay buffers

4.3.1 Uniform

4.3.2 Prioritized

5 DQN

5.1 Architectures

5.1.1 Vanilla

5.1.2 Double

5.1.3 Dueling

5.2 Bellman equation

5.2.1 Max

5.2.2 Softmax

6 GNN

7 Results

8 Conclusions

Bibliography

[1] AIcrowd. *AIcrowd*. URL: https://www.aicrowd.com/.

[2] Wälter Jonas. “Existing and novel Approaches to the Vehicle Rescheduling Problem (VRSP)”. MA thesis. HSR Hochschule für Technik Rapperswil, Mar. 2020.

[3] Jing-Quan Li, Pitu B. Mirchandani, and Denis Borenstein. “The vehicle rescheduling problem: Model and algorithms”. In: *Networks* 50.3 (2007), pp. 211–229. DOI: https://doi.org/10.1002/net.20199.