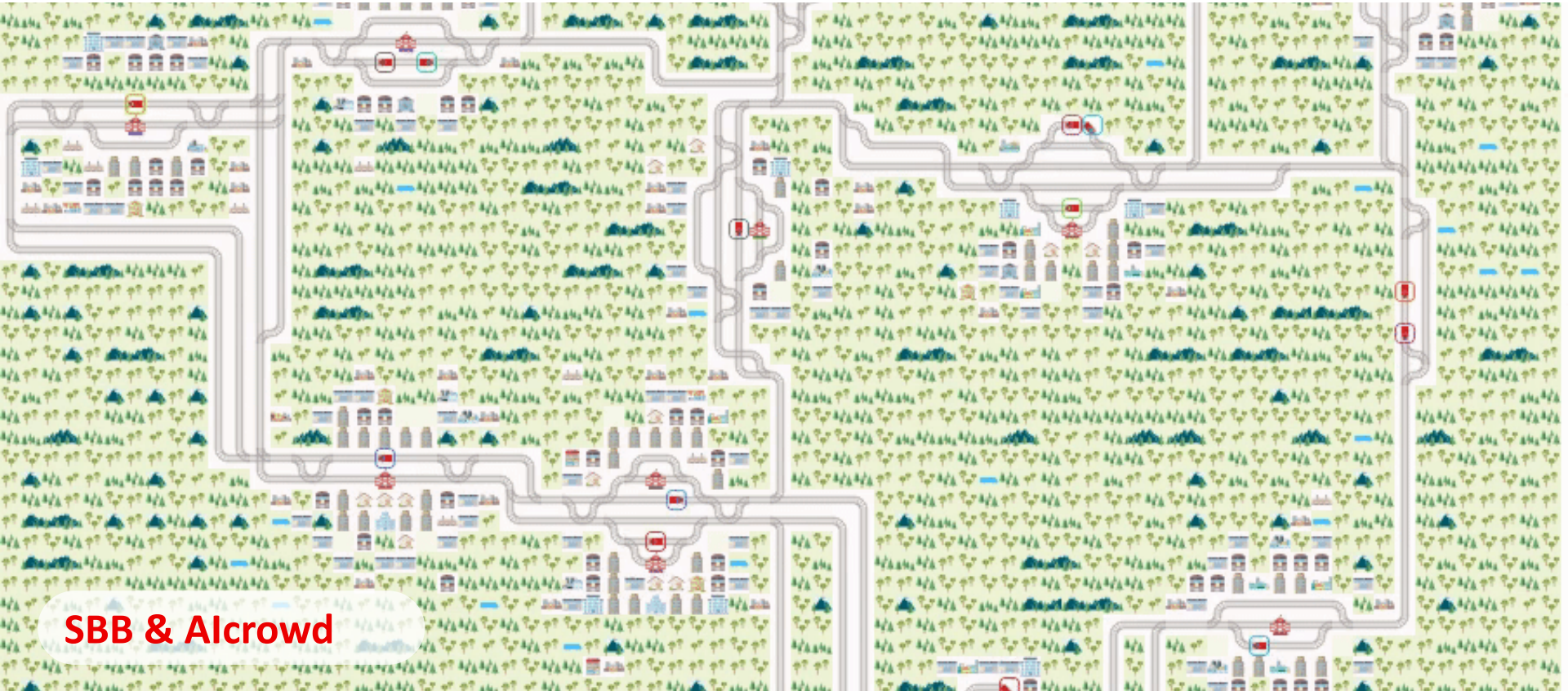
FLATLAND

Challenge:

SBB CFF FFS

Multi-Agent traffic optimization



SBB & Alcrowd

Swiss Federal Railways (SBB CFF FFS)

Facts and figures.

1’260’000

210’000 t

3’232 km

10’671

12’997

33’408

32’754

Most dense network

Weather

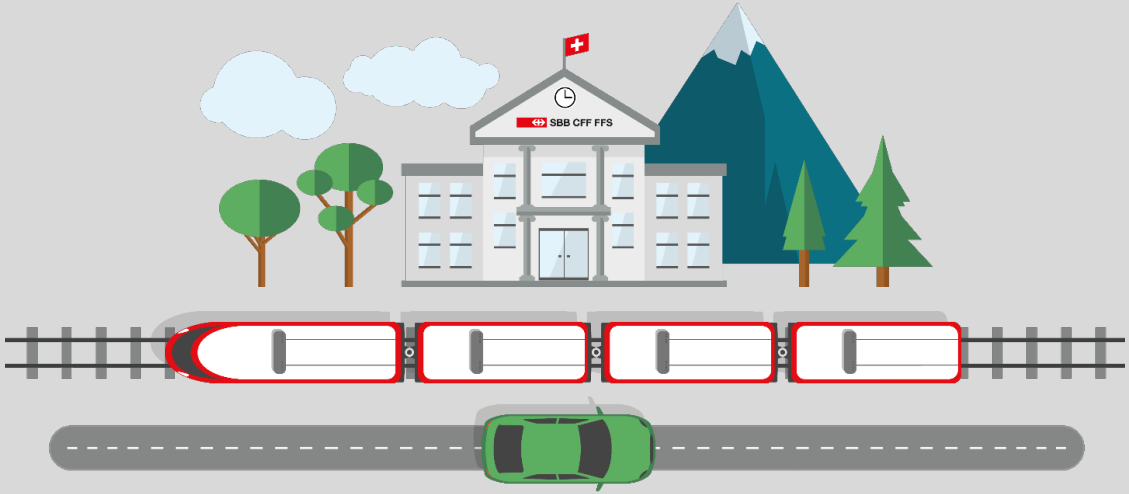
Passenger

Infrastructure

Events

Crowdsourcing Challenge on AIcrowd.com

Asking the world for help…



FLATLAND

https://www.aicrowd.com/challenges/flatland-challenge

Flatland **Railway Challenge.**
Objectives for participants.

Observation Builder

Find the best suited observation for the task at hand.

Predictor

Come up with novel ways to predict future in **FLAT**LAND rail environment and avoid conflicts in a timely manner.

Controller

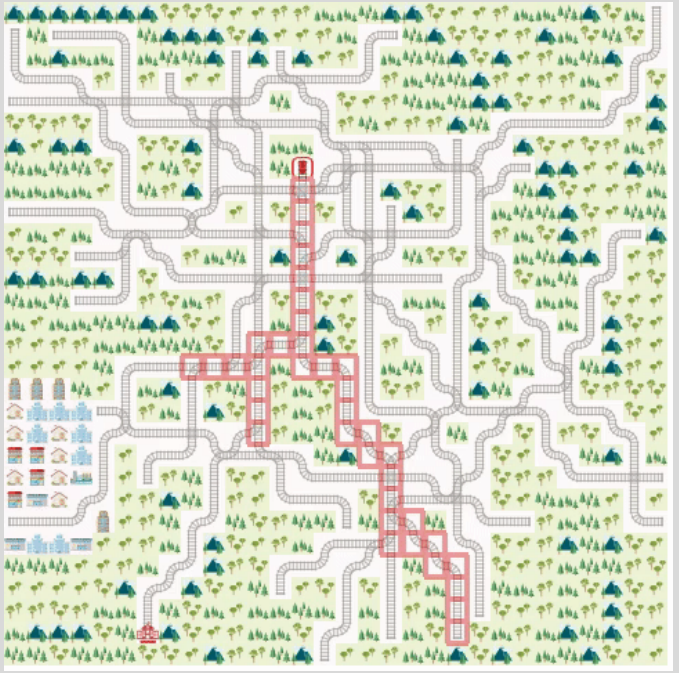
Use Machine Learning or Operations Research algorithms to tackle the problem.

Observation Builder.

What should an agent observe?

Local grid view

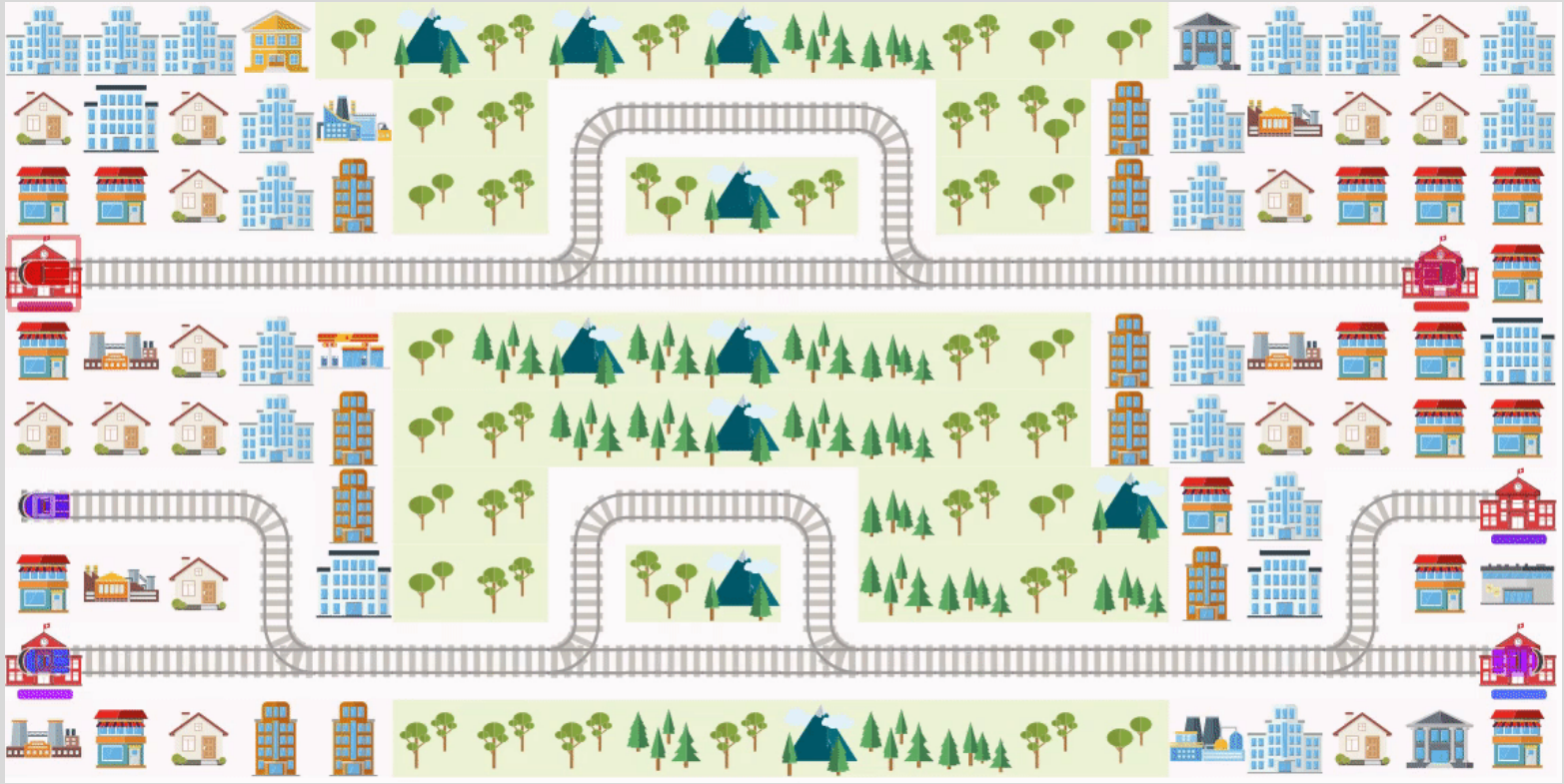
Local tree view



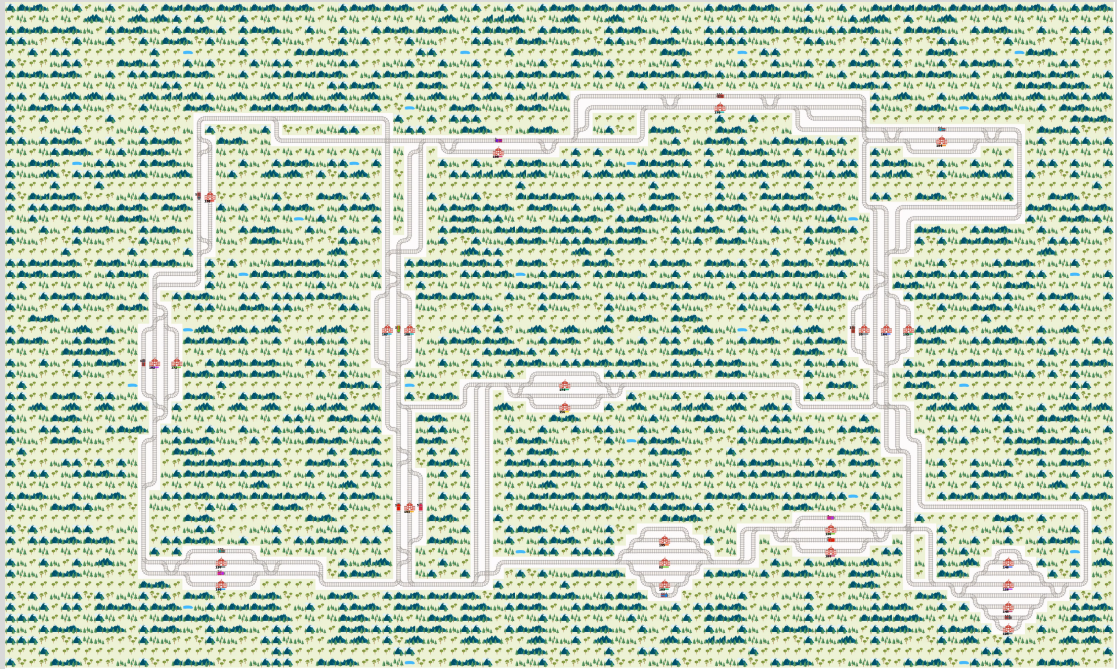
Round 0.

Single agent navigation.

Round 1.
Conflict detection.

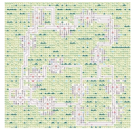
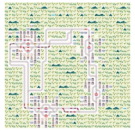
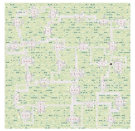
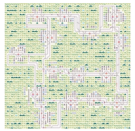


Round 2.
Optimizing traffic.



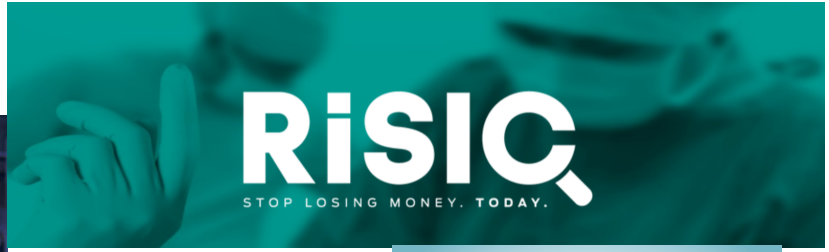
Leaderboard.

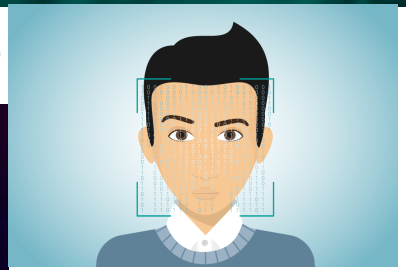
Top Solutions Submitted to Challenge.

Δ	#	Participants	Media	Fraction of done-agents	Mean Reward	normalized_reward	Entries	Last Submission	
	01	mugurelionut		0.990	-18213.620	-11.320	61	Sat, 4 Jan 2020 22:21	View
	02	CkUa		0.960	-23621.430	-14.620	51	Sun, 5 Jan 2020 01:38	View
	03	JelDor		0.951	-21998.130	-13.640	87	Sun, 5 Jan 2020 12:03	View
	04	wwwjon		0.794	-47373.620	-27.800	16	Sat, 28 Dec 2019 16:53	View
	05	ai-team-flatland-netcetera		0.555	-48437.860	-29.280	34	Fri, 3 Jan 2020 20:31	View

Projects

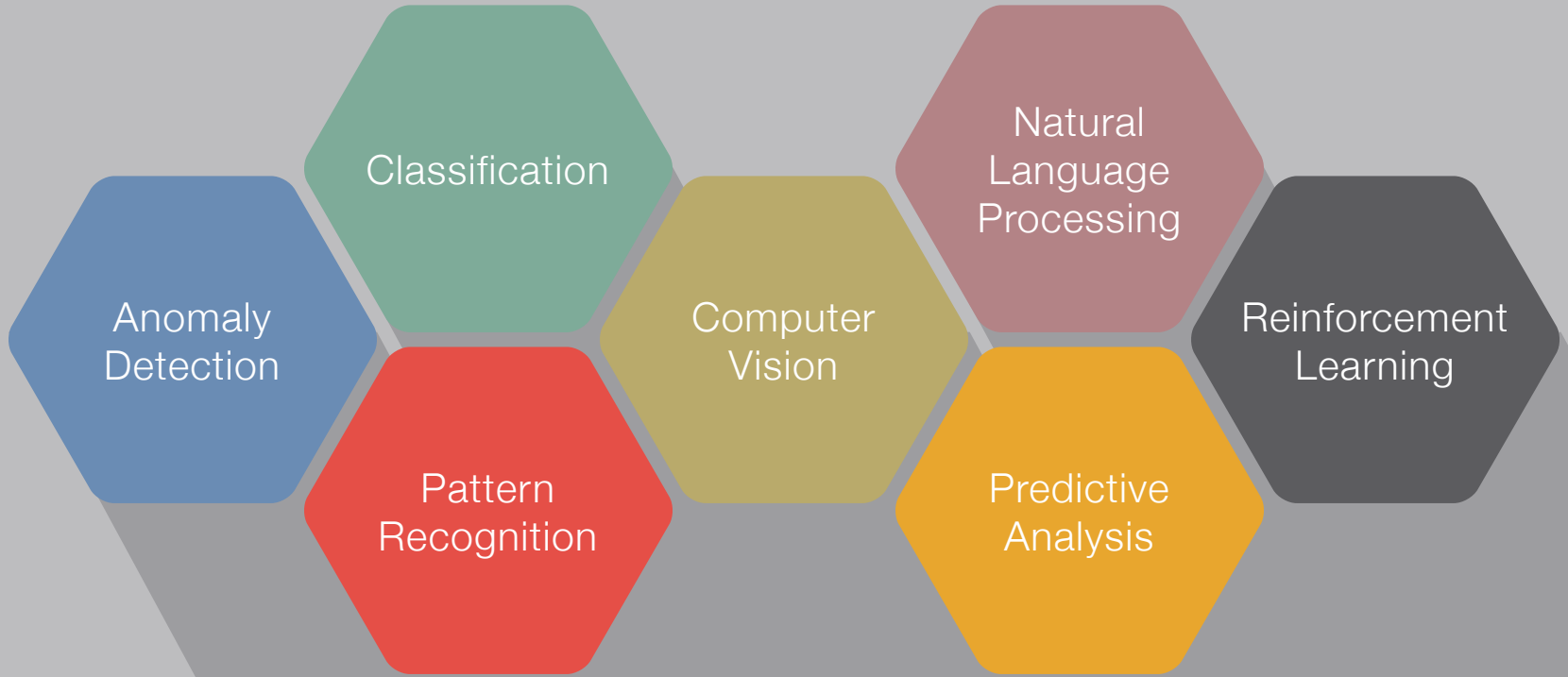
netcetera





هيئة الصحة
HEALTH AUTHORITY

Specializations



Team

Evgenija Spirovska
(tech lead)

Aleksandar Nikov
(Head of innovation)

Ramon Grunder
(project lead)

Nikola Velichkovski
(intern)

Zafir Stojanovski
(intern)

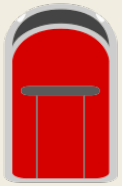
Oliver Tanevski
(intern)

Darko Filipovski
(intern)

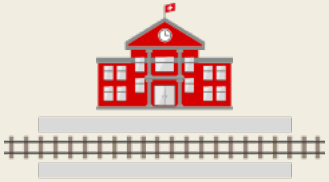
Reinforcement learning solution

Environment

Is reinforcement learning applicable?



Agent



Reward: + 1 on getting into station
- 1 for each step

State of agent — Model — Action

State representation

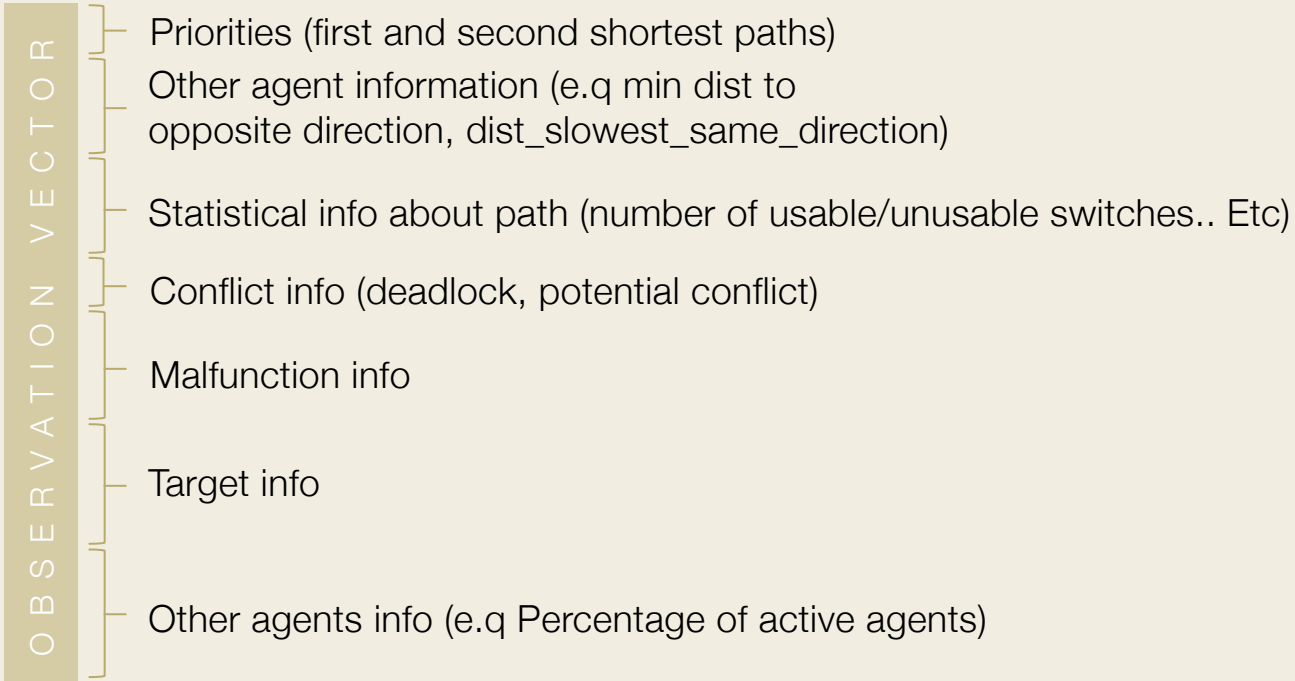
Many tries:

- Local observation
- k-shortest paths
- ….

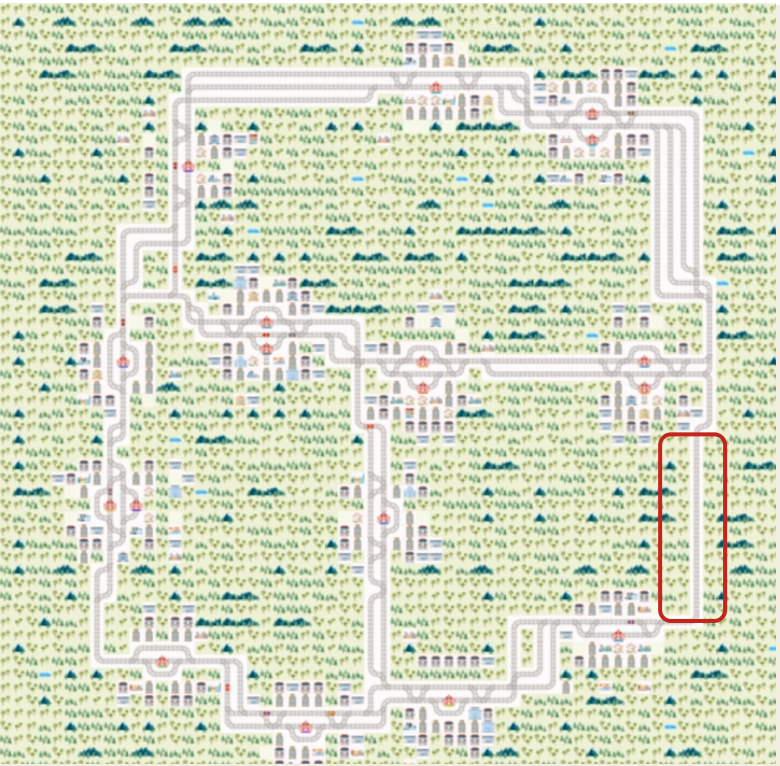
Best (in terms of time complexity and quality of the solution):

Global observation = Two shortest paths from the current switch

Observation vector

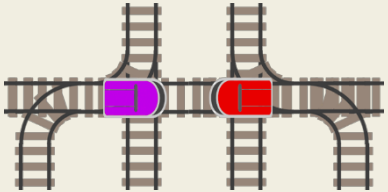


Round 2 – Problems and solutions



Problem:

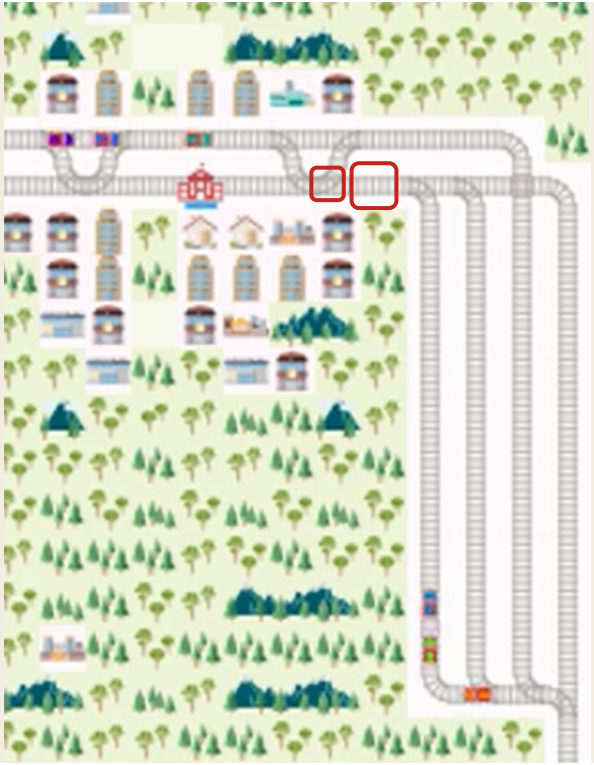
Easy to get into deadlock + deadlock is costly

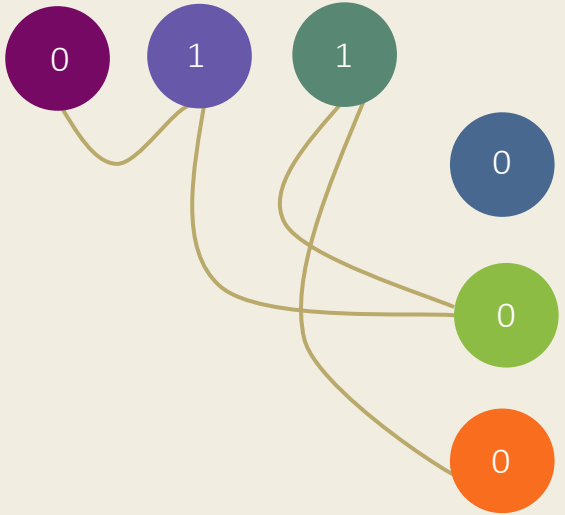


Solution:

· Guide the search with priority
· Add info about deadlock in the observation vector

Priority



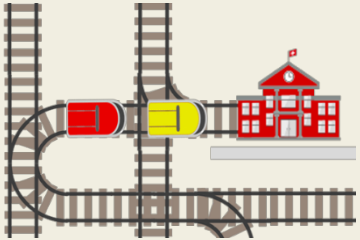


Round 2 – Problems and solutions

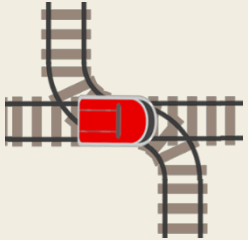
Problems:

Not taking the shortest path:

Blocking agents behind the train:



Random stopping



Solutions:

Penalty for each step

More information about agents behind the train

Penalty for random stopping

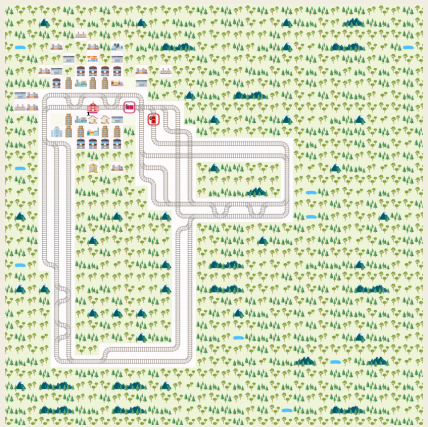
Reward

10	if agent is in the target
-0.35	if is getting closer to its target
-0.5	regular step
-1	stopping
-3	stopping on switch
-5	not following priority
-10	deadlock

Environment provider

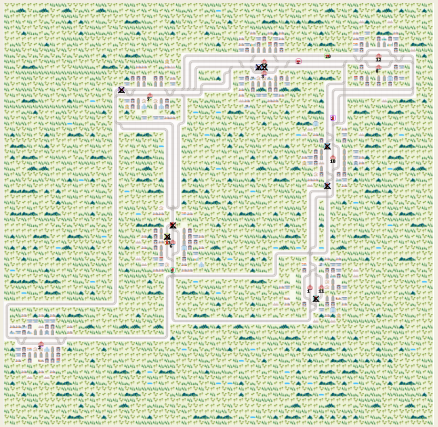
Different levels for learning different tasks

Level 1



Learn how to get to the station
Learn to follow priority

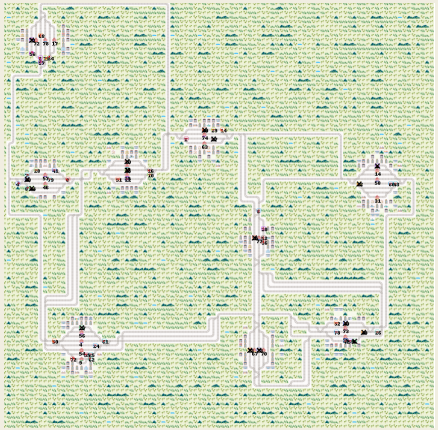
Level 2



Take speed and
malfunction into account

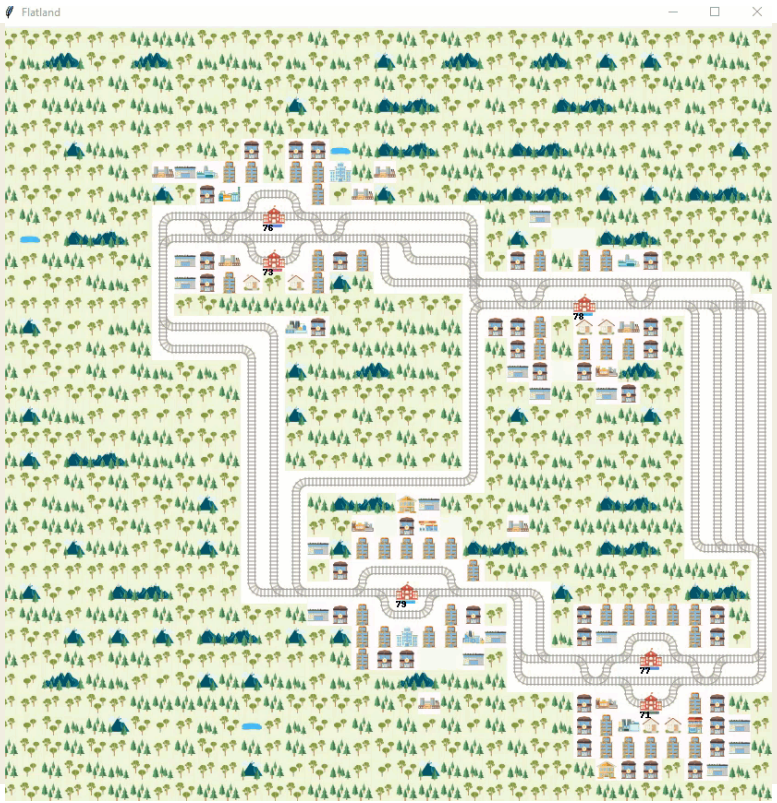
...

Level 3



Learn complex behaviors

Conclusion



Round 1: 99% of agents done

Round 2: 55% of agents done

Future work:

- Different models for different levels
- Different reward structure in the training process
- Include information about other agents

…

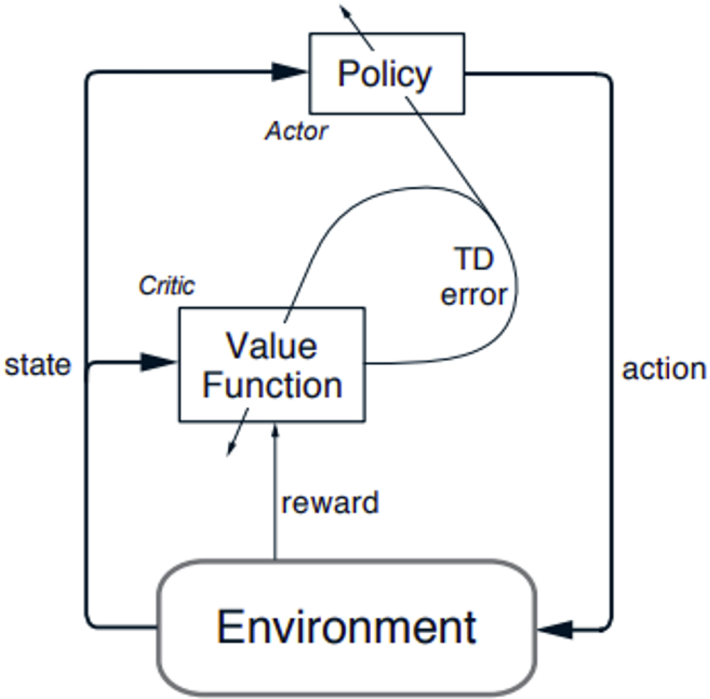
Thank you and let's continue the conversation:

ai-team@netcetera.com

Flatland Challenge
2nd place solution

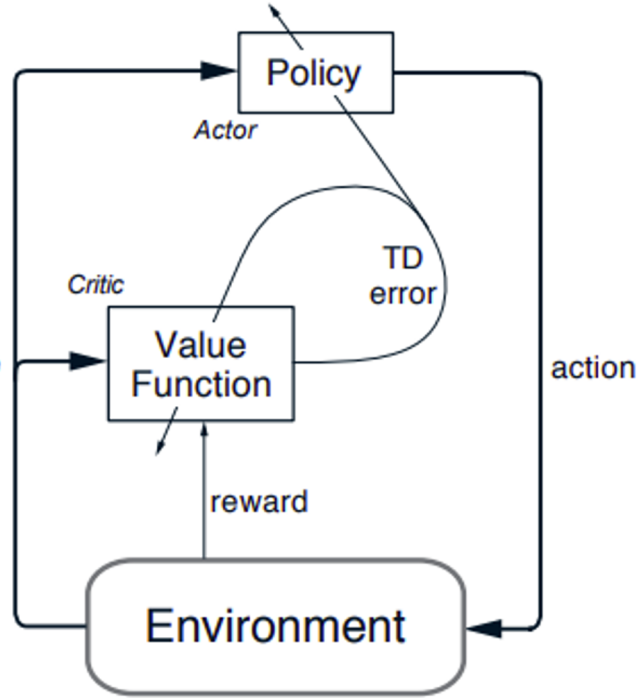
Team: CkUa (= Cherkasy, Ukraine)
Roman Chernenko & Vitaly Bondar

Reinforcement learning



Sutton and Barto, 1998

Reinforcement learning



Sutton and Barto, 1998

The approach

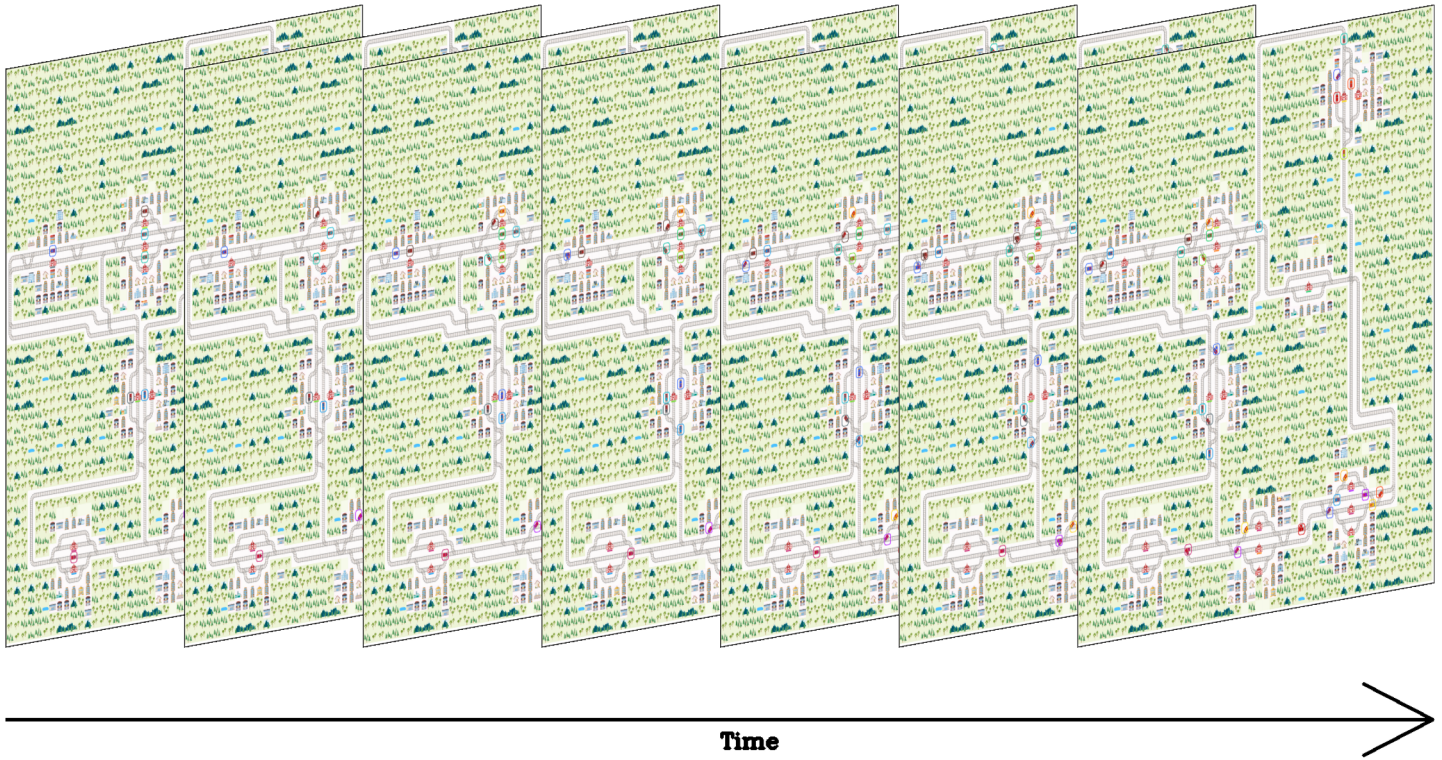
Initialization:

- create graph based on environment
- precalculate shortest path to all targets from all (position; orientation) pairs

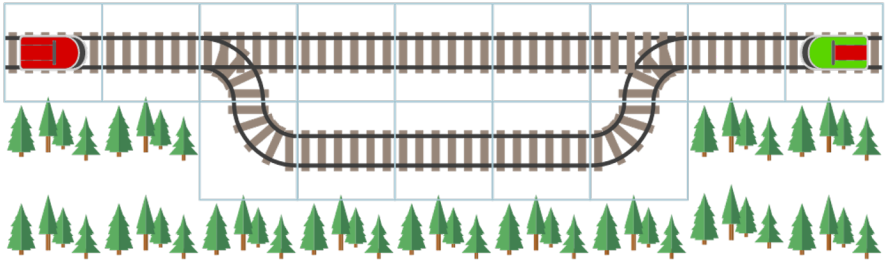
On every step:

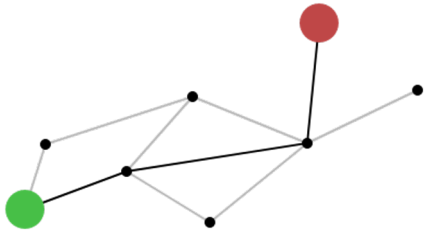
- Rescheduling all started trains if any new malfunction
- Partial rescheduling if full rescheduling failed
- Try to find new ways for rescheduled/blocked and not started trains

Space-time data structure



Plan train after train with A* search



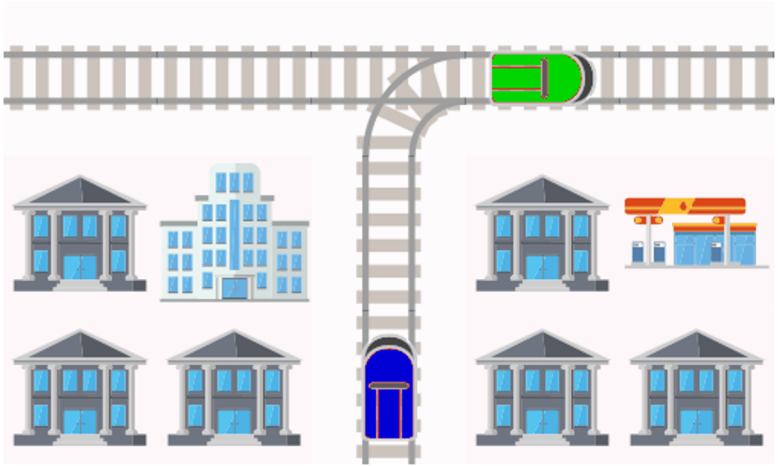


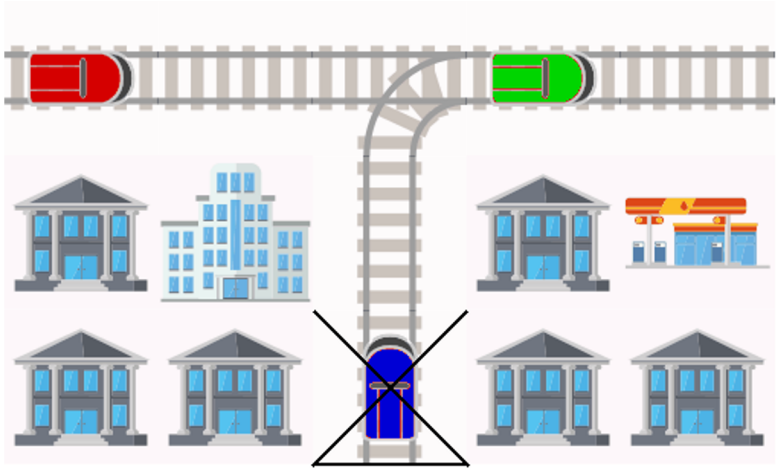
f(x) = g(x) + h(x)

g(x) - found travel time in seconds to the current vertex + waiting time

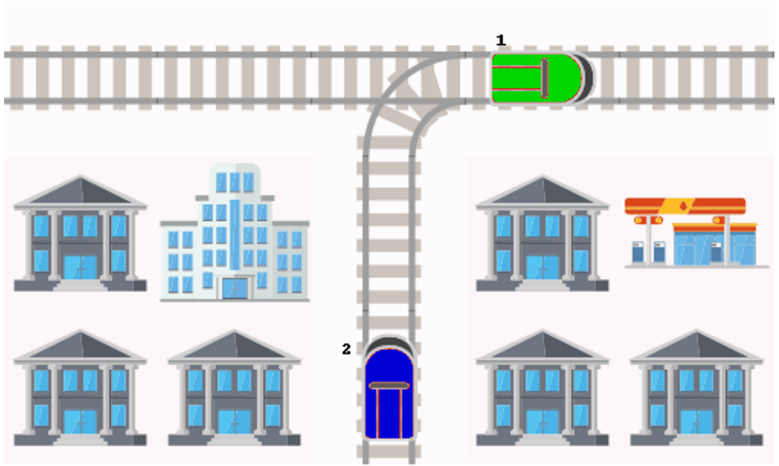
h(x) - minimal possible travel time to the target

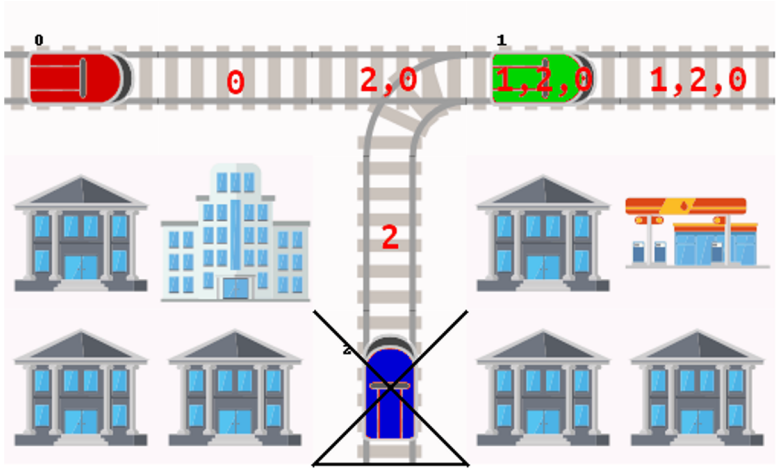
Rescheduling



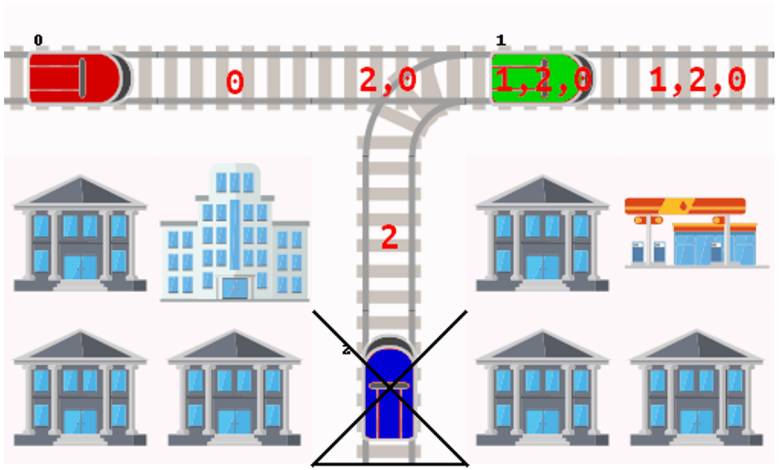


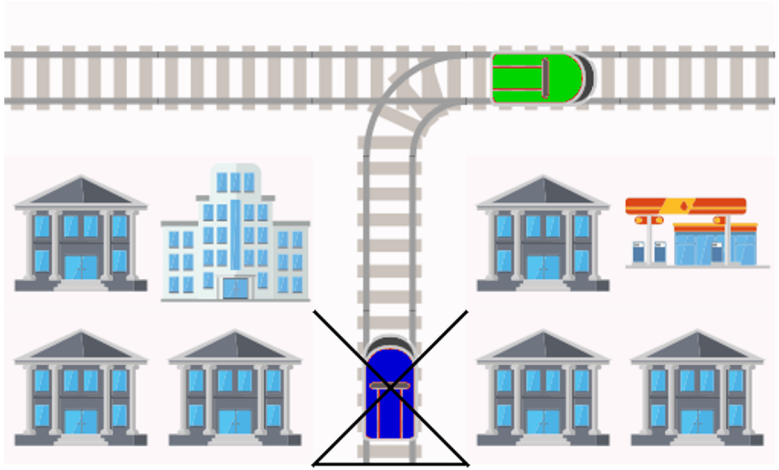
Rescheduling (Invariant)



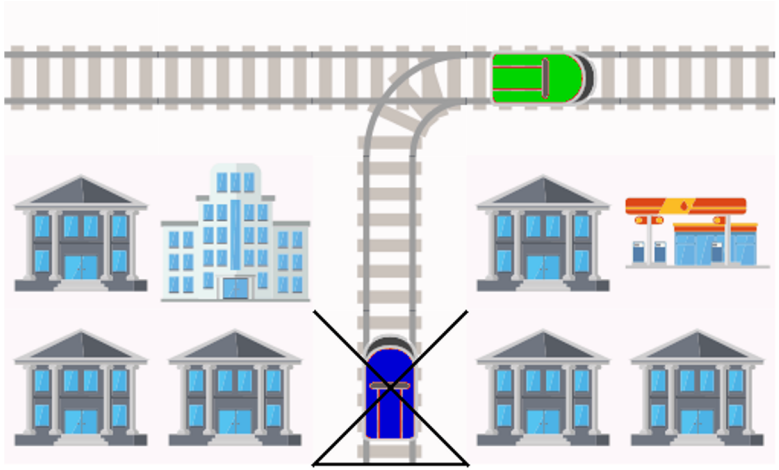


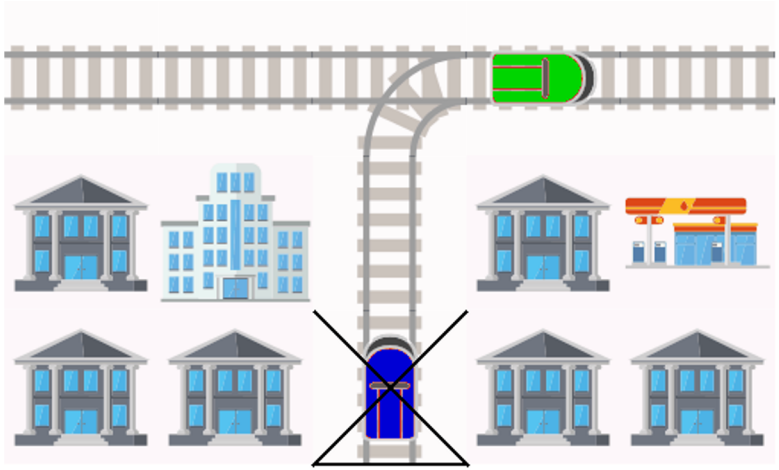
Rescheduling (Naive but valid plan)



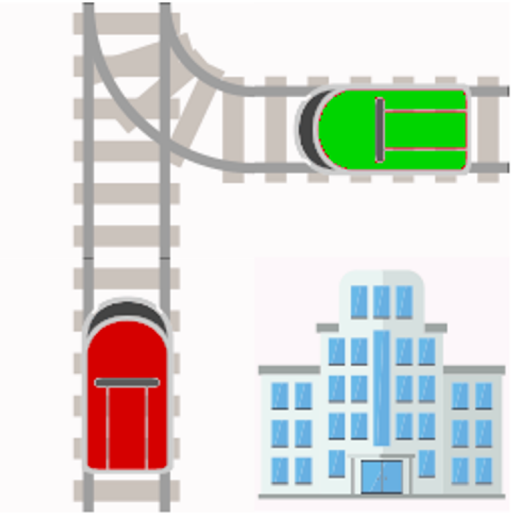


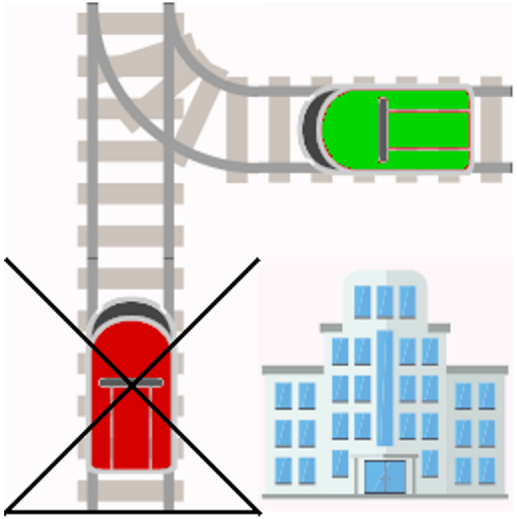
Search better path for updated trains (one by one)



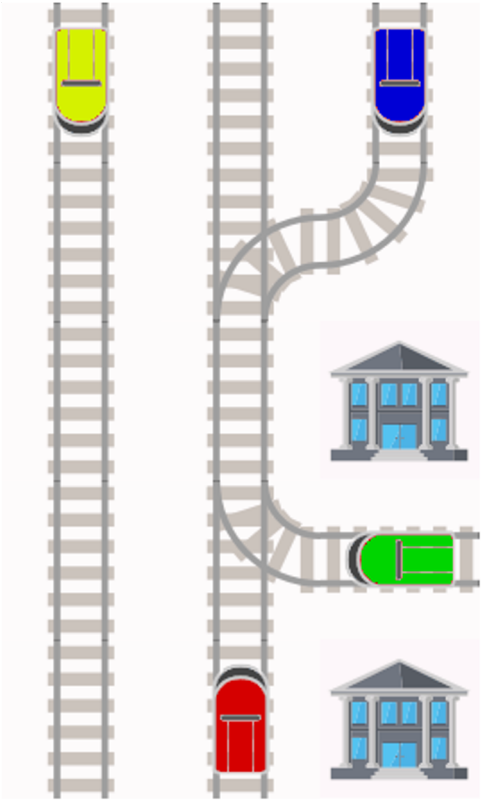


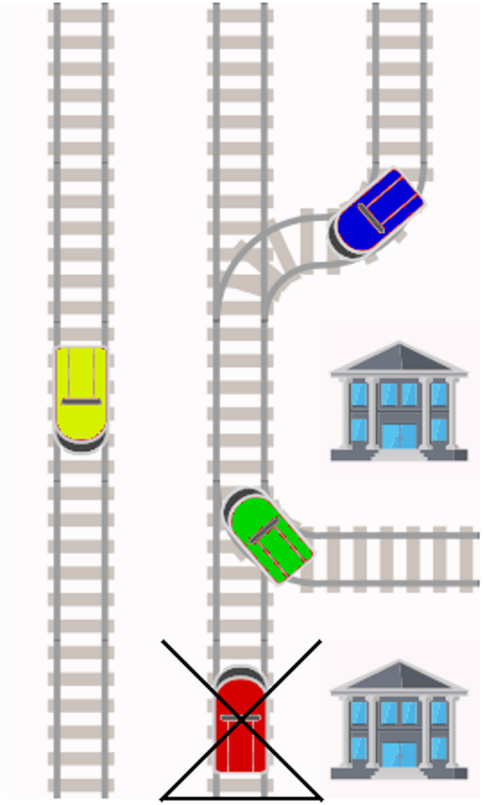
Partial rescheduling (problem)



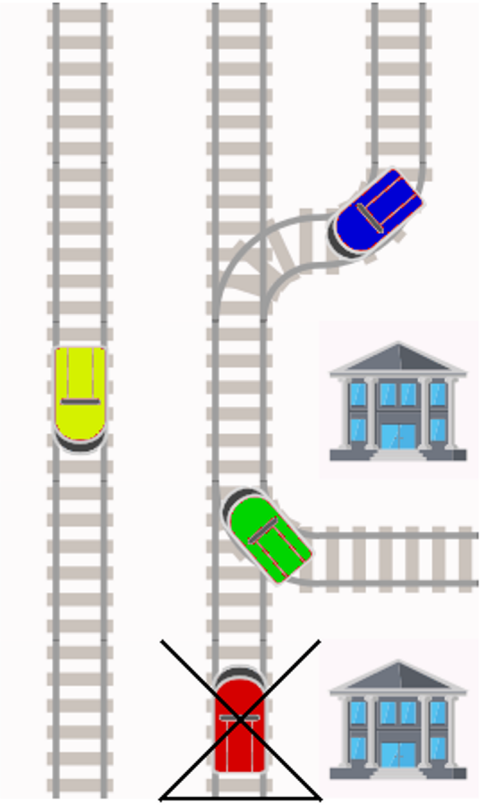


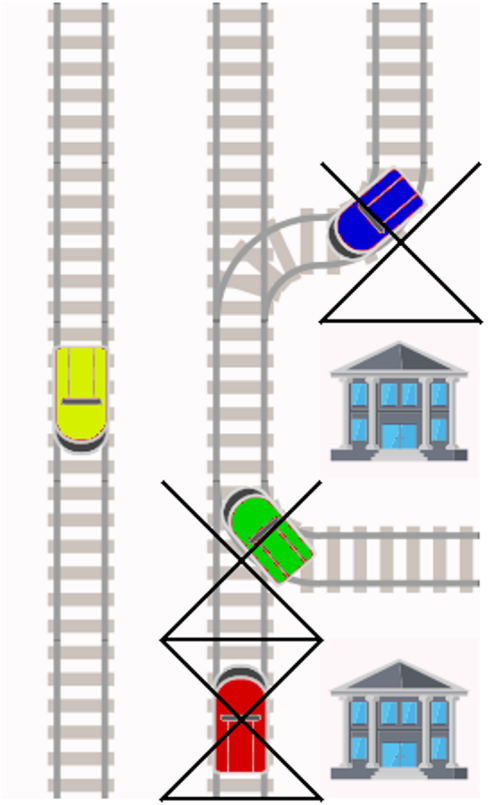
Partial rescheduling (problem)

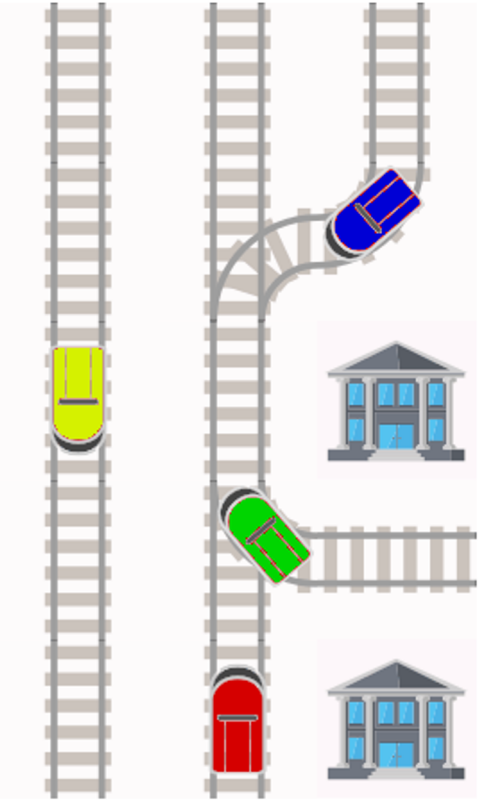




Partial rescheduling







Thank you

AICrowd SBB Flatland Challenge

Solution by mugurelionut@

Mugurel-Ionut Andreica
Google Switzerland Gmbh*

*Participation in this challenge was performed outside working hours, using only personal hardware and software. The topic of the challenge and the work performed within it are unrelated to the author's day-to-day job and to Google's business in general.

Solution Overview

- 2 main components:
  - (Re-)generating agent paths
    - Over a time-expanded graph
  - Updating agent paths after a malfunction
    - To (try to) avoid deadlocks
    - To bring the system back to a consistent state
      - Where each agent has a feasible path over the time-expanded graph

(Re-)generating agent paths (1/5)

- Precondition: Each agent already has a feasible path over the time-expanded graph
  - Default (initially) = an agent always stays outside the environment
- Only tries to improve the set of paths
- Time-expanded graph:
  - (cell, time) = a node in the graph
  - 0 <= time < TMAX
    - Official TMAX = 2560
      - I used a slightly larger limit to allow most agents to have feasible schedules
  - But there can be up to 150x150=22500 cells!
    - Actually, the network is sparse: always <= 3000 used cells

(Re-)generating agent paths (2/5)

- Up to **max_runs** runs
- At each run: **num_threads** parallel threads
- Each thread:
  - Generates **num_permutations** of the N agents
  - In permutation order:
    - The path of an agent is removed from the time-expanded graph
    - A new (possibly shorter) path is found (A*-style) and inserted
  - The new set of paths is scored => keep the best one found
  - If the best set found is better than the initial one:
    - Start next run with the best set as initial set

(Re-)generating agent paths (3/5)

- Scoring a set of paths:
  - Main objective: Number of agents reaching the destination
  - Tie-breaker: Sum over all agents of (time to reach destination)^{E}
    - Experimented with multiple values of E (from 0.25 to 4)
      - E=1 produced the best results (it corresponds directly to maximizing the reward)
- Generating permutations:
  - Experimented with multiple orderings
  - Random, but all agents with higher speeds before all agents with lower speeds (agents with speed 1 before agents with speed 0.5, etc.)
    - Intuition: Remove as many agents ASAP so they don’t malfunction inside the environment

(Re-)generating agent paths (4/5)

- Shortest path algorithm for a single agent:
  - A heap with entries sorted by estimated (optimistic) time to arrive at the destination
  - Main objective: Minimize time to reach the destination
  - Tie-breaker: Maximize the moment when the agent enters the environment
  - Only entries where the agent can make a decision are added to the heap
    - If it’s malfunctioning or executing an ongoing move, only the time when it can start making decisions is added

(Re-)generating agent paths (5/5)

- Shortest path algorithm for a single agent:
  - Uses all the rules of the simulator when computing moves, e.g.
    - Allows moving to (cell, t) is the agent located at (cell, t-1) leaves it at time t, but has lower agent id
    - Avoids frontal collisions (passing “through” each other)
    - Allows multiple agents to enter the same cell if all but the last of them have it as destination (and they enter it in increasing agent id order)
    - etc.
  - Also contains logic to avoid/minimize deadlocks (will be discussed later)

Updating agent paths after a malfunction (1/6)

- The path assignment becomes invalid
  - Cannot be used for making agent decisions
  - Cannot be used as initial assignment by the path (re-)generation component
- Repair the paths by maintaining agent visiting order at each cell
- With the new paths, run the path (re-)generation logic to improve them
- For every cell **C**: an ordering **a(C, 1), a(C, 2), …, a(C, num_cell_visits(C))**
  - Each visit of an agent **a(C, i)** has an associated time **tc(C, i)**
    Similar ordering for each agent **A**: **c(A, 1), c(A, 2), …, c(A, num_agent_visits(A))**
  - **ta(A,i)**=the time step when the agent A enters the cell **c(A,i)**

Updating agent paths after a malfunction (2/6)

- Direct correspondence between agent and cell visits:
  - **agent_to_cell_visit(A, i) = j**: the **i-th** visit of agent **A** (to cell **c(A, i)**) is the **j-th** visit at that cell
    - i.e. **a(c(A,i), j) = A** and **tc(c(A,i), j) = ta(A,i))**
- Similarly: a **cell_to_agent_visit(C, j) = i** mapping
- When an agent malfunctions:
  - Try to delay all the visits of all other agents which are impacted by the malfunction (to preserve the visiting order)
    - Can be computed via a DFS/BFS of the precedence graph defined by the visiting order (I used BFS)
  - Easily doable, if we could **pause** an agent at any time
    - And we can, except… some agents may have ongoing moves

Updating agent paths after a malfunction (3/6)

- Assume 2 agents have ongoing moves towards the same cell **C**
  - Scheduled to visit it in the order **A1**, followed by **A2**
    - E.g. **A1** is faster than **A2** or started the move earlier, so the schedule is: **A1** arrives at **C**, then leaves it, then **A2** arrives
  - If **A1** malfunctions before it enters **C** but after **A2** started moving towards **C**, then the ordering of the visits **cannot be preserved**.
- If 2 visits are swapped: potential deadlock
  - it doesn’t have to, but it’s hard to control the outcome
- How to handle deadlocks?

Updating agent paths after a malfunction (4/6)

- Option 1: If they occur, just expand the involved agents' paths to be blocked where deadlock occurs and continue with the algorithm as is
  - Problem: Unpredictable how may agents involved in a deadlock => large score variance caused by random events
    - Small parameter change in the algorithm can fully avoid deadlock or cause new ones
    - Hard to evaluate the quality of a change without using many tests
  - My first non-failing submission: 96.5%** agents done

** Most of my submissions had a bug in the heap implementation, which I discovered close to the end of the competition (just fixing this bug increased the score then from 98.2% to 98.9%).

Updating agent paths after a malfunction (5/6)

- Option 2: Avoid them altogether
  - Make sure no 2 agents move in parallel towards the same cell C
  - Needs support when delaying visits, as well as in the path (re-)generation component's shortest path algorithm
  - Problem: Too conservative
    - Agents are spaced too much in time even when malfunctions are rare
    - The "throughput" is not high enough
  - Scored approx. 97.5%**

** Most of my submissions had a bug in the heap implementation, which I discovered close to the end of the competition (just fixing this bug increased the score then from 98.2% to 98.9%).

Updating agent paths after a malfunction (6/6)

- Option 3: Avoid only (some of) the most common cases causing deadlocks
  - Avoid nested time intervals of agents going to the same cell
    - [t1,t2] and [t3,t4] with t1 <= t3 and t4 <= t2
  - In theory, deadlocks could still occur, but it's unlikely
    - Verified on local tests
    - Also on the official 250 test cases (added asserts to make my submission fail in case of deadlocks)
  - Score improved to 98.2% (98.9% after the bug fix)
  - Final push to 99%:
    - Use 3 threads (was using only 1 or 2 before)
    - Run the path (re-)generation logic more often

Final Algorithm

- Initially run path (re-)generation logic with (**max_runs=4, num_threads=3, num_permutations=20**)
- Whenever a malfunction occurs at some time step:
  - Run the path update logic
  - If maximum time to destination increased or the number of done agents decreased: increase a counter
    - If counter>**3**:
      - Run the path-regeneration logic in “full mode” (**max_runs=4, num_threads=3, num_permutations=10**)
      - Reset the counter
    - Else: run path (re-)generation in “restricted mode” (**max_runs=2, num_threads=3, num_permutations=2**).

Many Unexplored Ideas (1/2)

- Construct the precedence graph online during the shortest path algorithm and allow overlapping moves if they are guaranteed to not cause cycles
- Allow changing the visiting order in the update logic if it doesn’t cause cycles
  - Local search style, by swapping some visits and checking if the solution improves
- Use past malfunction data to estimate generator params and run local simulations (during the “real” simulation)
- Generate paths for groups of agents simultaneously

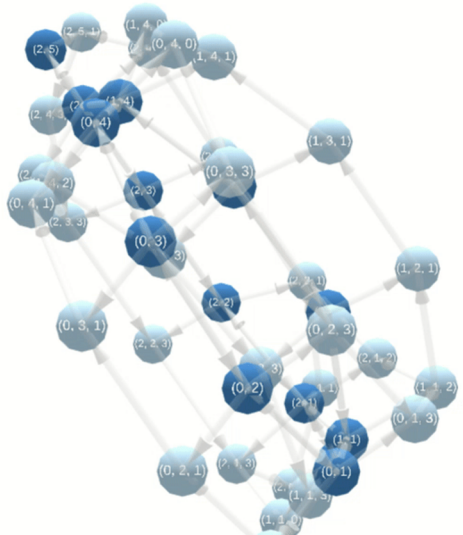
Many Unexplored Ideas (2/2)

- And others…
- So why didn’t I try them?
  - Non-trivial amount of work for uncertain reward
  - Not enough motivation from the leaderboard :)
  - Likely to make everything much slower (exceeding the 8 hours time limit), so they would need to be used sparingly

Thank you!

Flatland - Graph Representations

Jeremy Watson, AI Crowd



Why use graphs?

- Reduce the complexity in the observation
  - Efficient representation of “sparse” environments
- A way of thinking about the problem
- Draw on existing libraries
  - Common representation & manipulation
  - Existing algorithms
  - Graph Graphics - d3js, 3d-forcegraph
- Some nice EDA - Exploratory Data Analysis!

Libraries & Code

- Python - **NetworkX** - Graph library.
- Static images from Flatland, PIL, Matplotlib.
- Animation using (basic) Javascript with these libraries:
  - **Vasco Asturiano** - https://github.com/vasturiano/3d-force-graph
  - **Three.js, d3js**

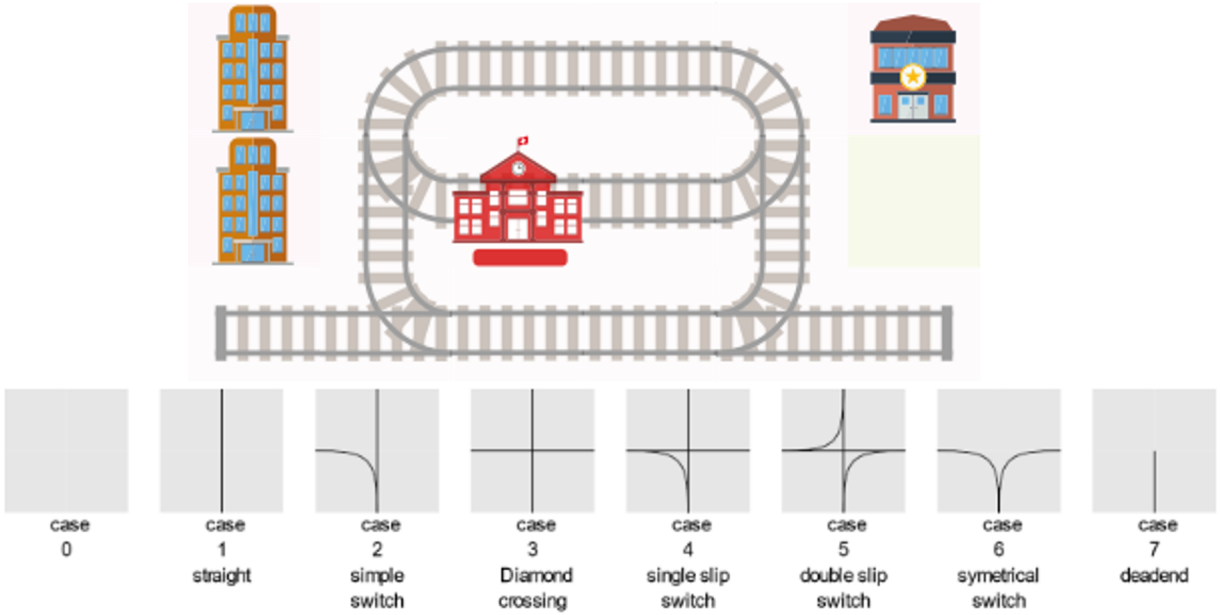
Notebooks & html/js wrappers in **flatland repo** under `flatland/notebooks`

Currently in **branch** `223_UpdateEditor_55_notebooks`

(Sadly we can't run Javascript animations from CoLab etc)

How can we use Graphs - “nodes and edges” - for flatland?

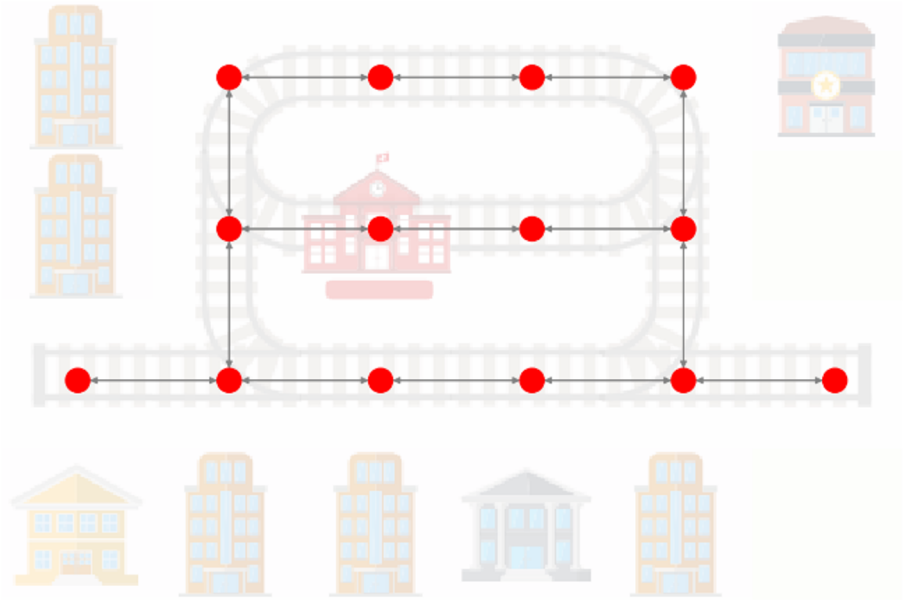
Looks simple enough, but how to capture the junctions?



Simple Nodes for rails - can’t handle junctions

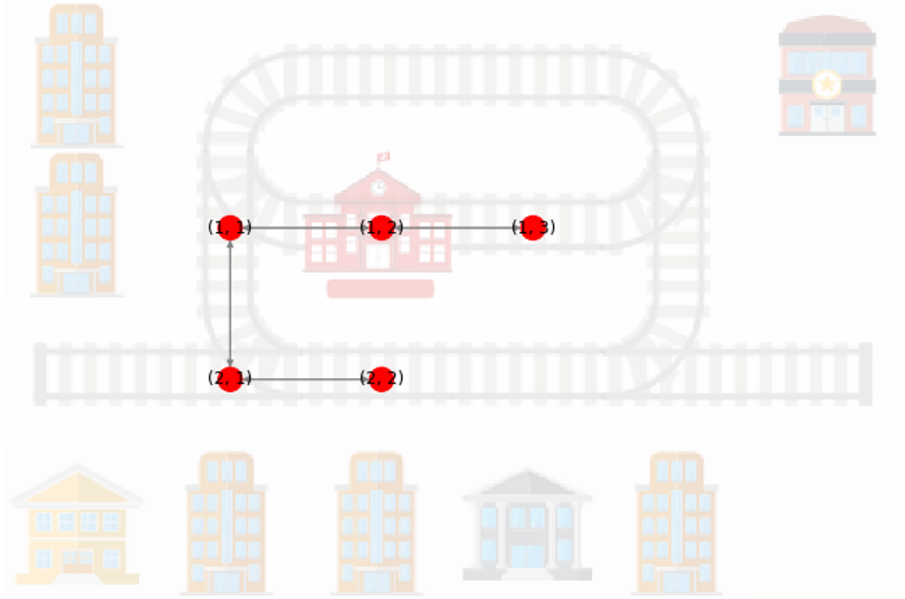
Rail switches are not simple nodes connected to other nodes.

The admissible exit directions depend on the entry direction.



Simple Nodes for rails - can’t handle junctions

“Shortest Path” algorithm picks an illegal route



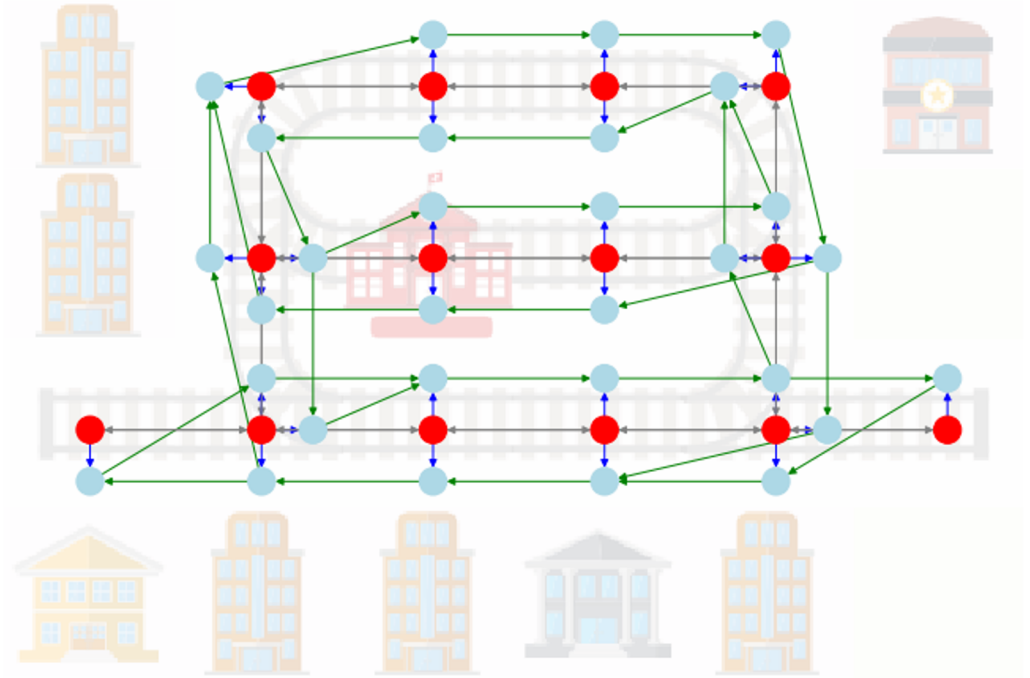
Approach - DiGraph with Grid nodes + Rail Nodes with directed edges

"rail" nodes - light blue

These have directed edges indicating direction of movement. There are typically two of them for each "grid" node.

For a junction / slip, we have three - one for each direction of entry.

You should be able to trace the green arrows around a junction.



“grid” nodes - red - provide a structure or layout.

“grid” edges between grid nodes,

“hold” edges hold rail nodes to grid nodes.

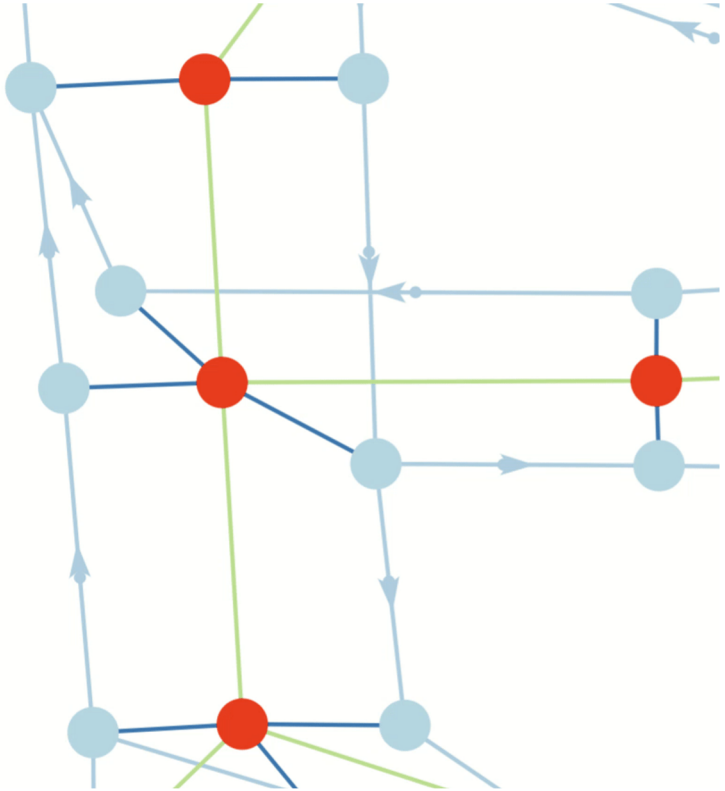
A Slip / Switch in detail

Entry direction -> Exit direction(s)

- North -> North
- South -> South or East
- West -> North

Flatland direction numbers:

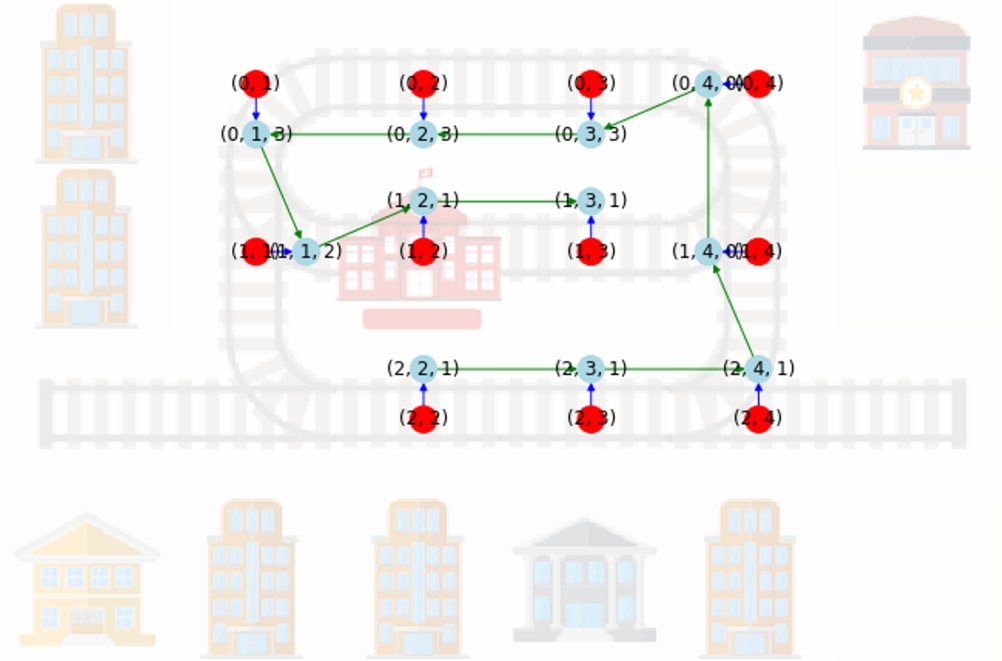
- 0 -> 0
- 2 -> 2, 1
- 3 -> 0



Shortest Path with correct transitions

We can now use standard algorithms from NetworkX to find the shortest path.

Here the algorithm has correctly picked a longer path.



Note:

Start and target directions are important. If we don't care about them, then we need to pick the shortest path across all combinations of admissible start + end directions.

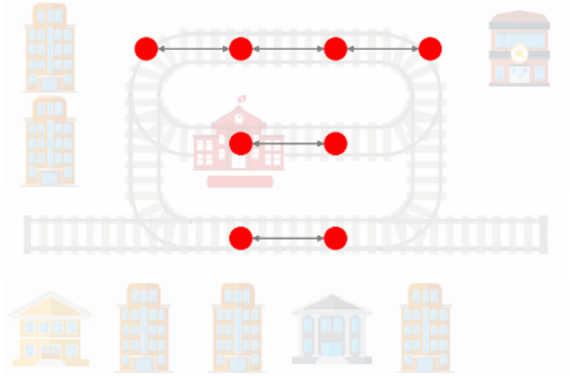
Shorten (contract) linear paths

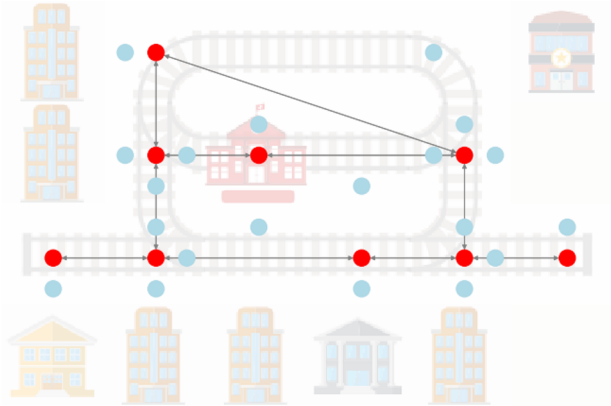
We can now simplify the graph, by removing unbranching “linear” paths.

We preserve a single grid node to represent each contracted path.

(Perhaps we could just use an edge but here we keep a node)

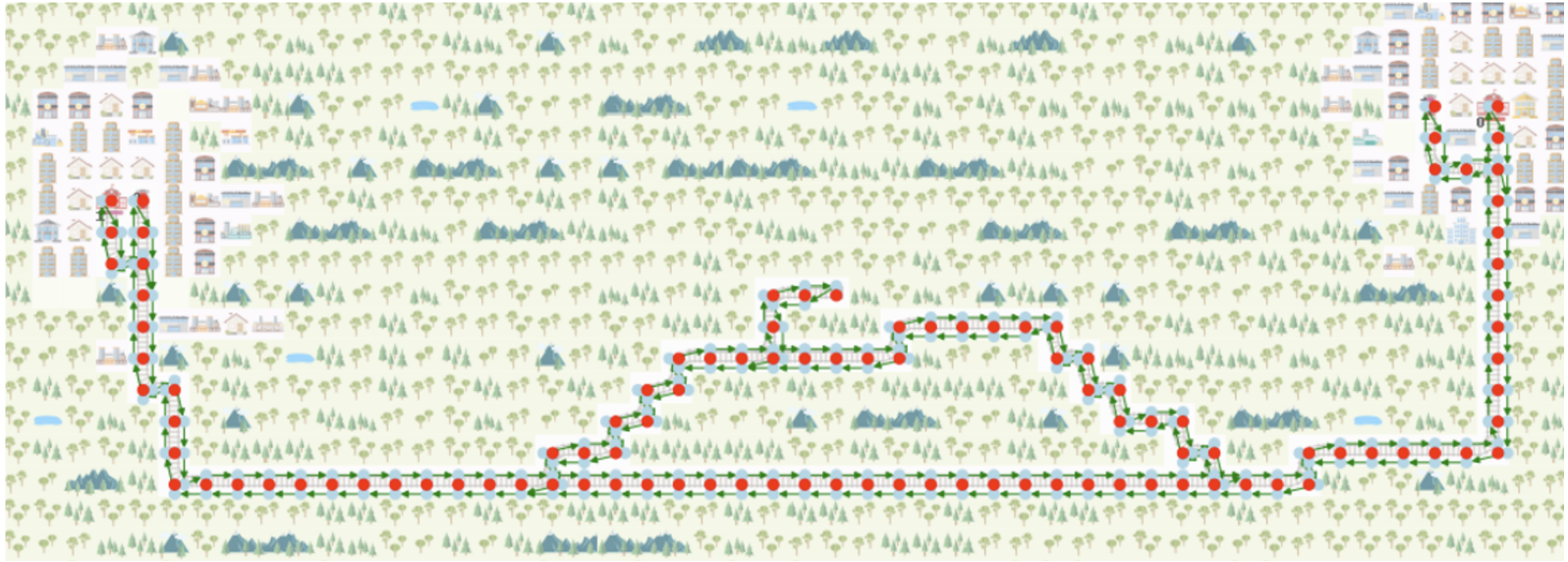
(See repo for code. Wasn’t as simple as I’d hoped)



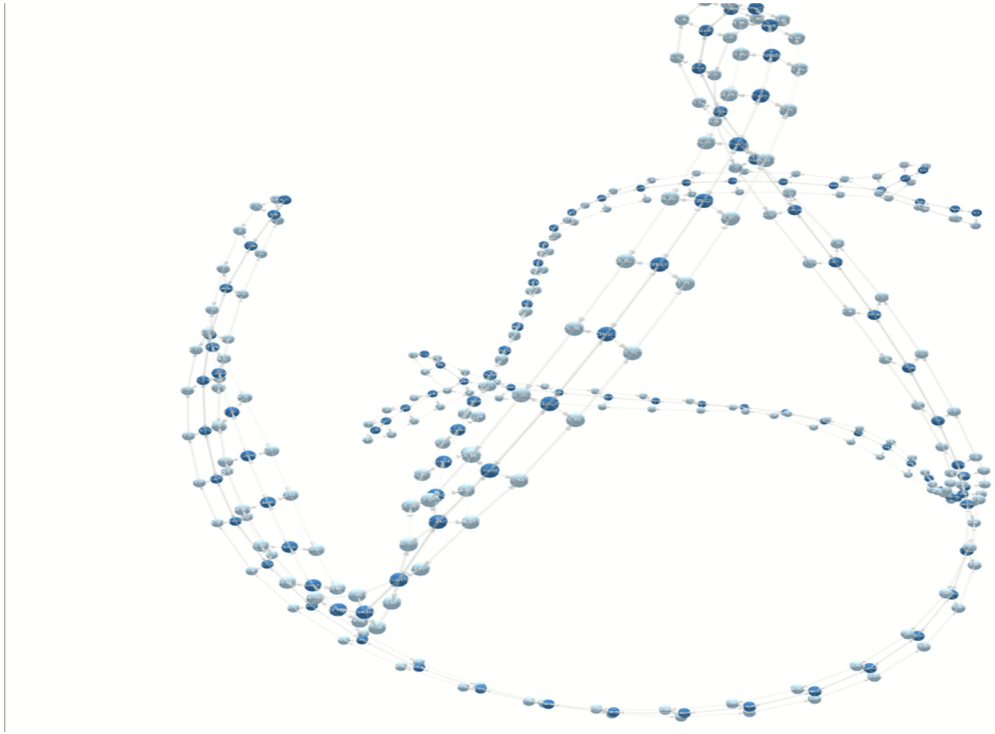


Contracting paths - efficient for sparse envs

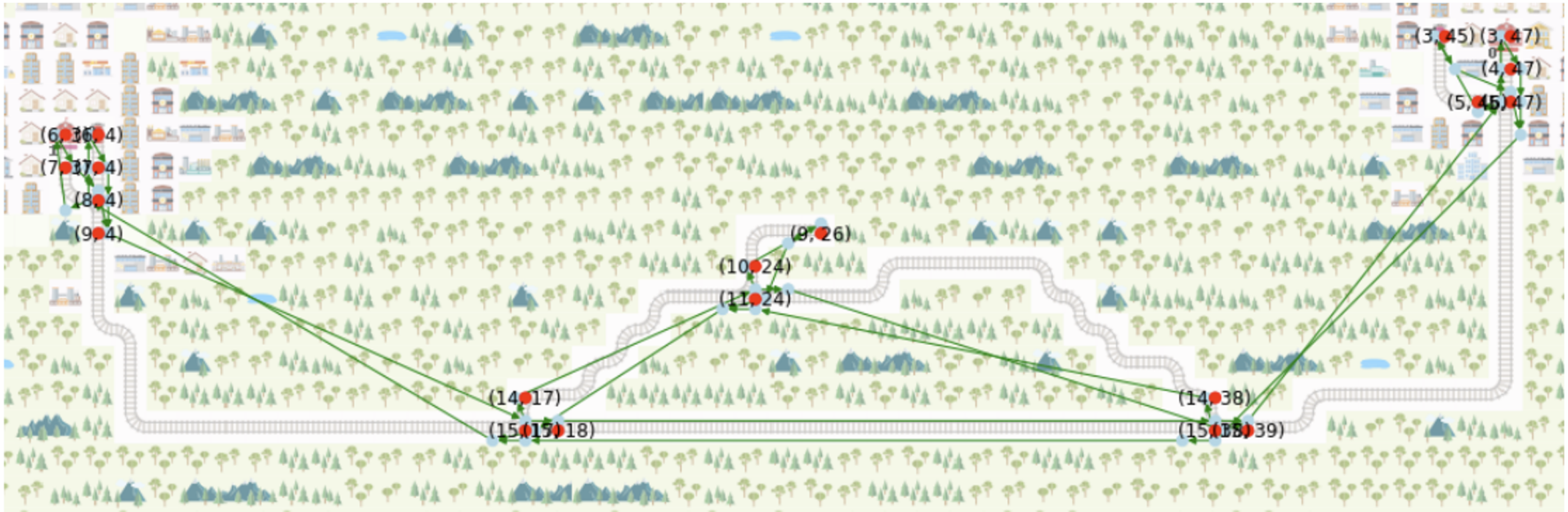
Graph for all rail cells



Looks nice but still a bit complicated...

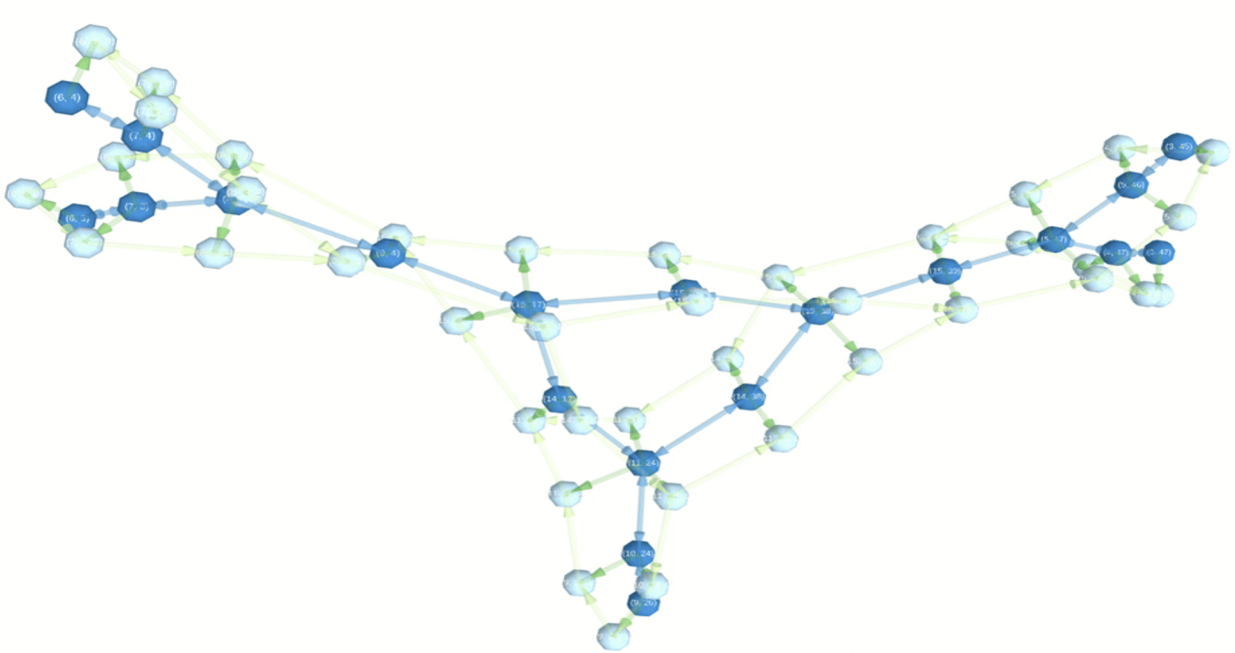


Contract the linear paths

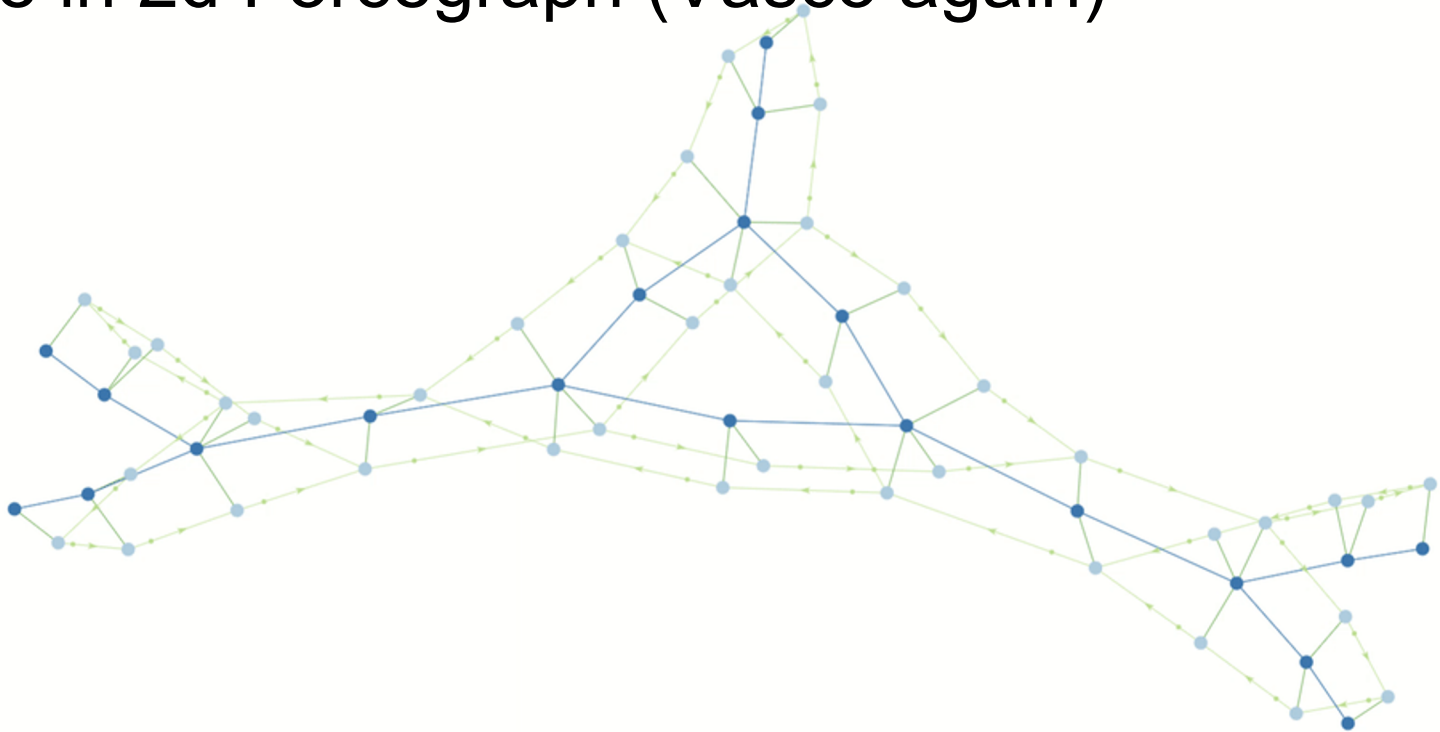


Neater with paths contracted

Graph form starts to look simpler (Vasco - 3d ForceGraph)



Same in 2d Forcegraph (Vasco again)



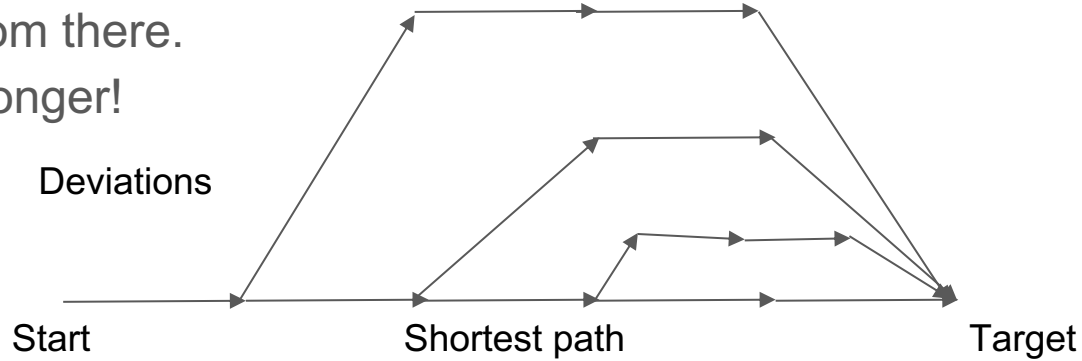
Generating Alternative Paths

- SBB state that shortest-path is well-established - no need for AI there
- Therefore we can take “shortest path” from any point as “atomic”
- This logic already used in Tree Observation

Rather than generating a tree, we suggest generating “deviation” paths:

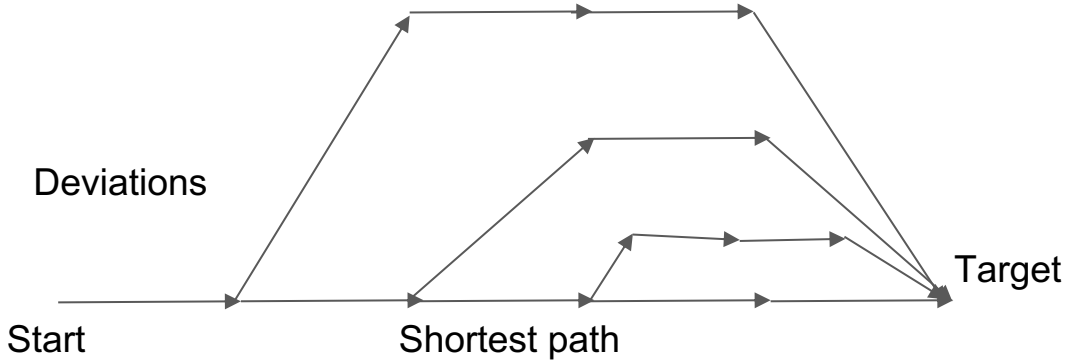
- For each decision point along the shortest path, take the “other choice”
- Take the shortest path from there.
- The deviation paths are longer!

Remember there are only ever a max of 2 choices for an agent at a junction.



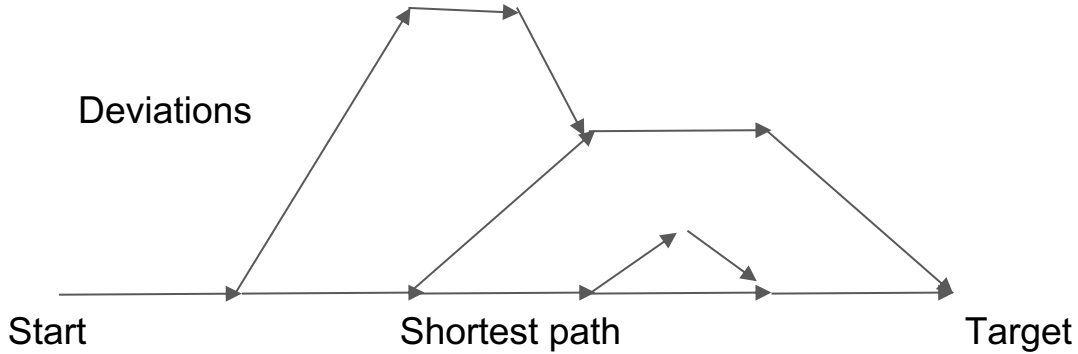
Alternative Paths - Often they recombine

Agent sees separate paths:



Reality is paths recombine.

Doesn’t really matter.



Alternative paths: Simple example

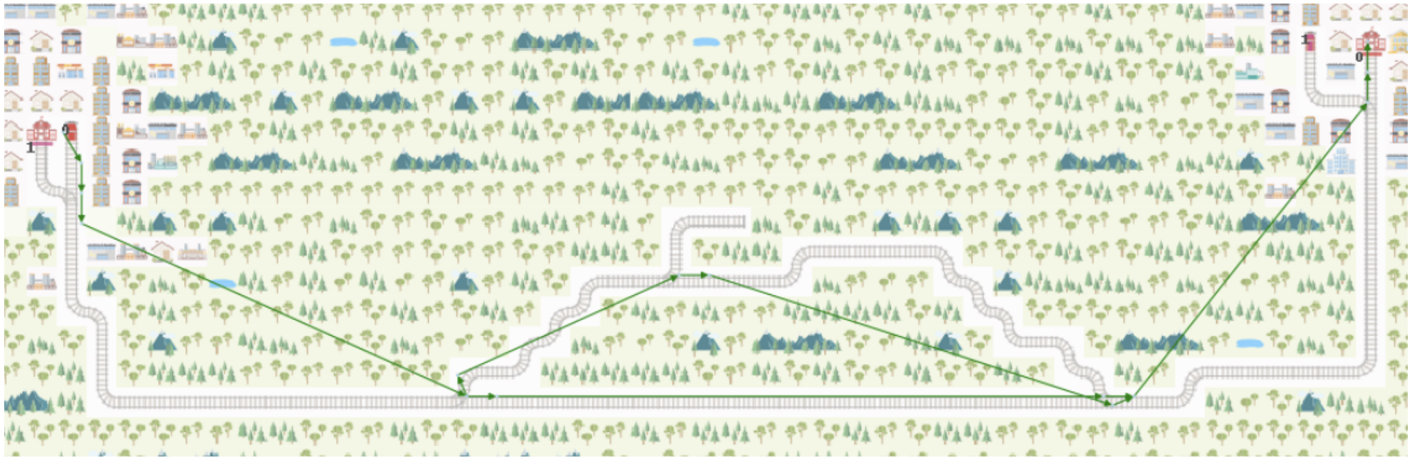
Two agents whose shortest paths will conflict.

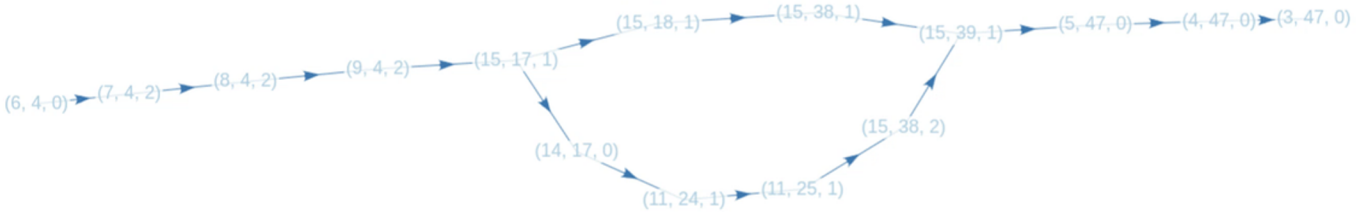
They both have alternative paths.

Agent 0 - Left to Right

Agent 1 - Right to Left - same route

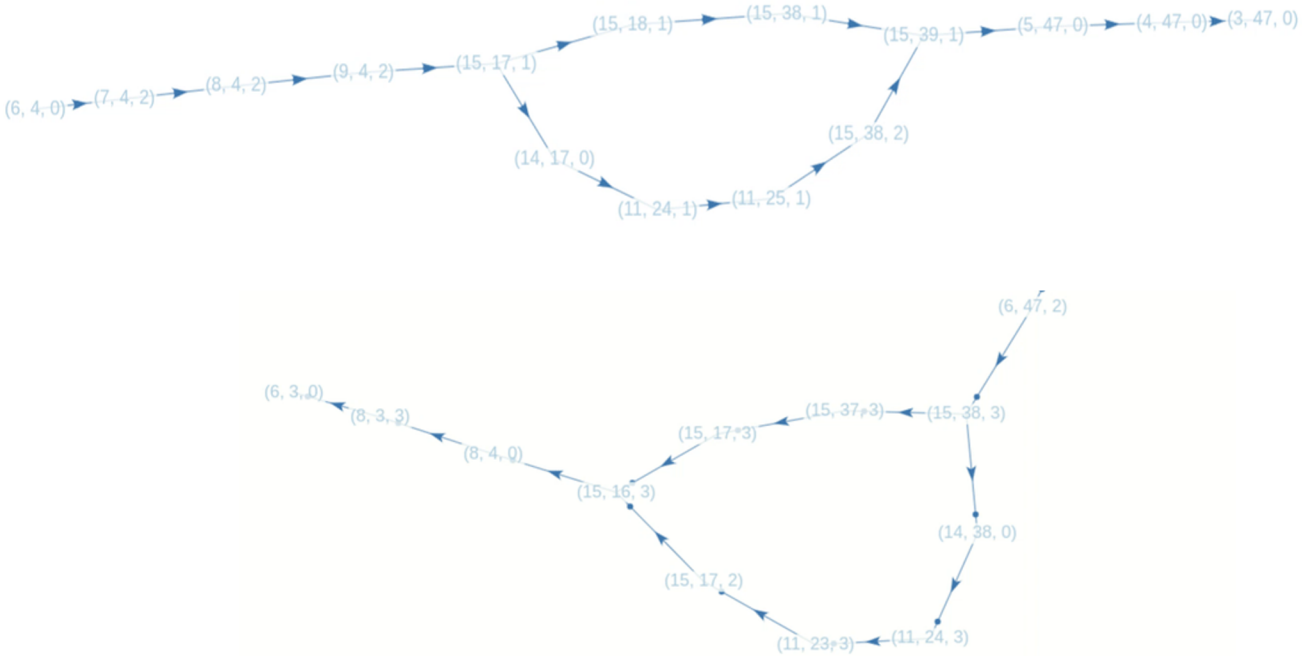
Alternative path created using a deviation





Alternative paths

We now have alternative paths for both agents - how to use?



How to detect future conflicts?

We have alternative paths so we can use them to avoid a conflict (in simple cases!)

How can we detect it in advance?

Our agents' paths are using distinct directed edges and "rail nodes" - they travel along distinct nodes.

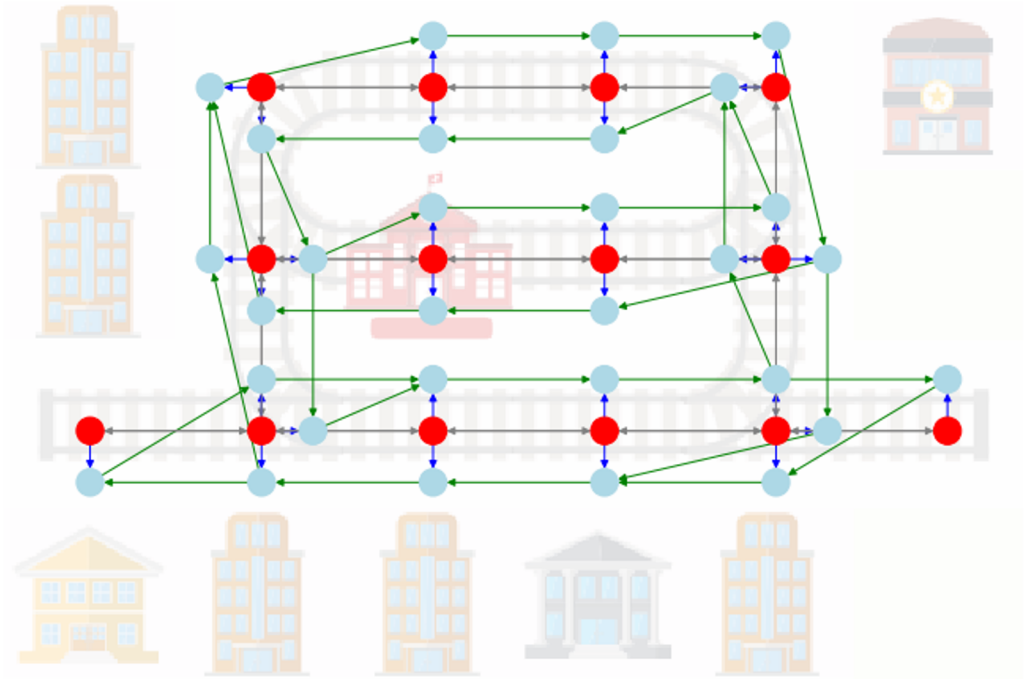
How can we tell that those nodes are in the same cell or linear path?

Detecting Conflicts

For Flatland, the “resources” are the cells.

In our graph representation, these are the red “grid” nodes which tie together the blue “rail” nodes.

Agents on different “rail” nodes but the same “grid” node imply a conflict.

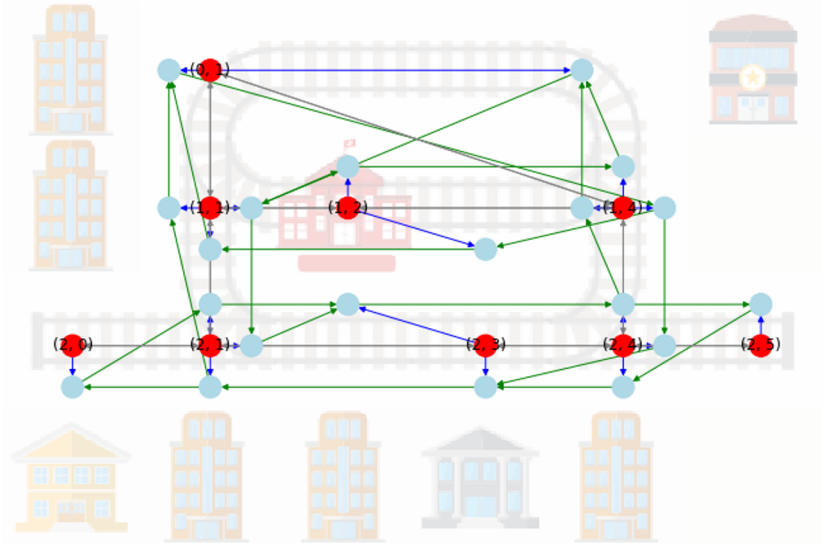


Detecting Conflicts

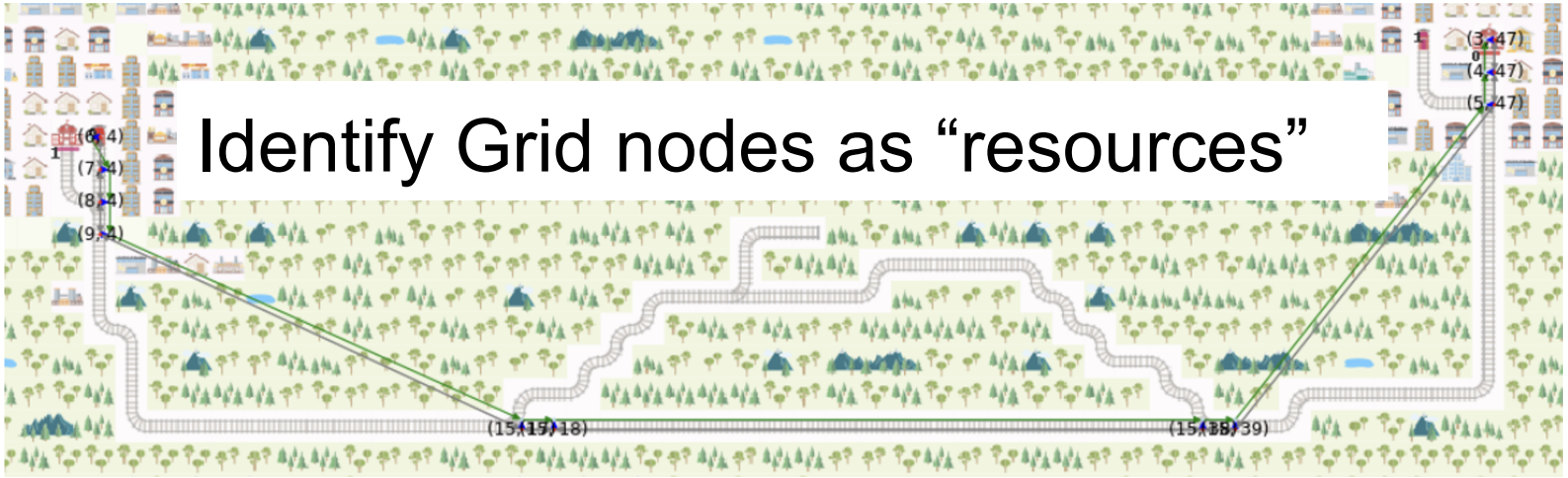
After contracting the linear paths -

We still need to detect conflicts on the red grid nodes, which now represent a whole linear path between two junctions.

We particularly need to detect agents travelling in opposite directions.

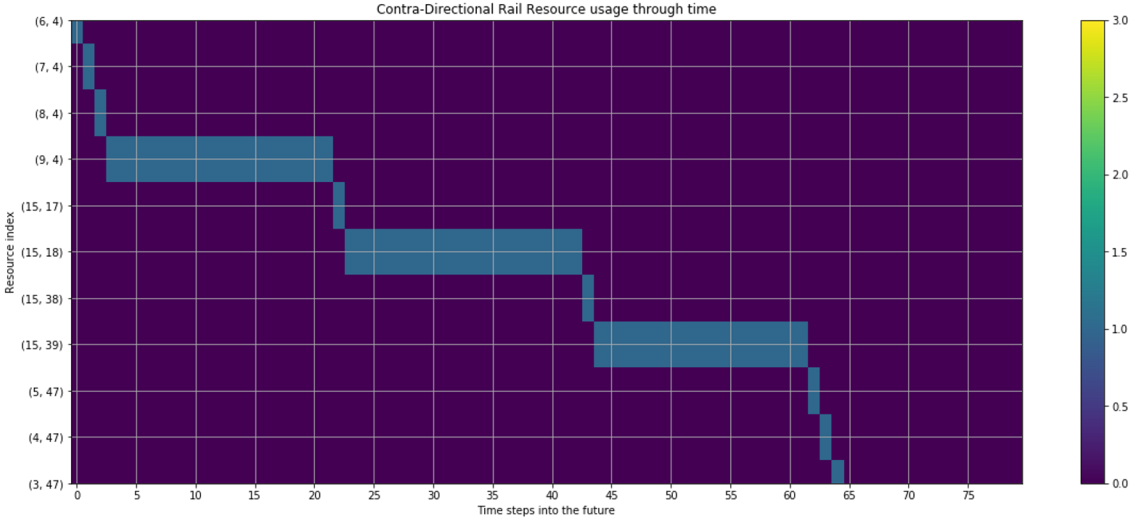


Identify Grid nodes as “resources”



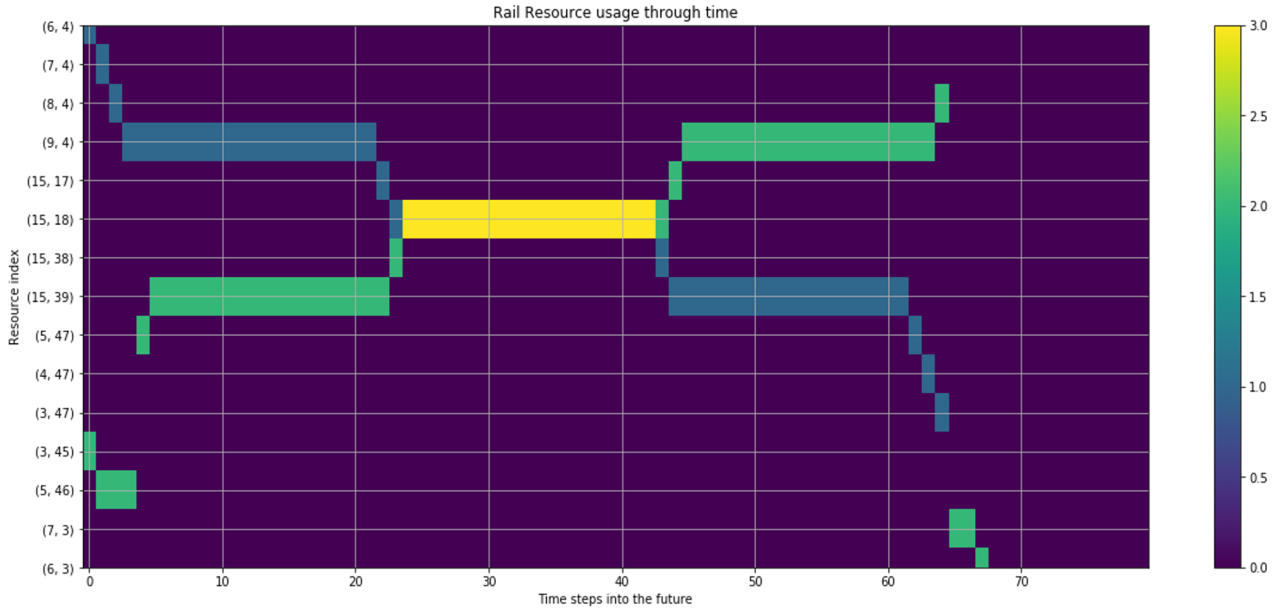
Grid / Resource Nodes down Y-axis

“Length” comes from the number of contracted nodes. (aka weight)



Identify grid nodes as resources

- Y axis has the resource nodes - ordered top-down here by agent 0's traversal (blue)
- Agent 1 (green) traverses in the opposite direction - mostly up.
- Node occupation time determined by length ÷ speed.
- They will conflict (yellow) at node "row=15, col=18" which has a "length" of about 20



15,17 (row, col)	1		Both agents in same node:	
15,18		1	2	1
15,38		1	(but can't tell direction here)	1

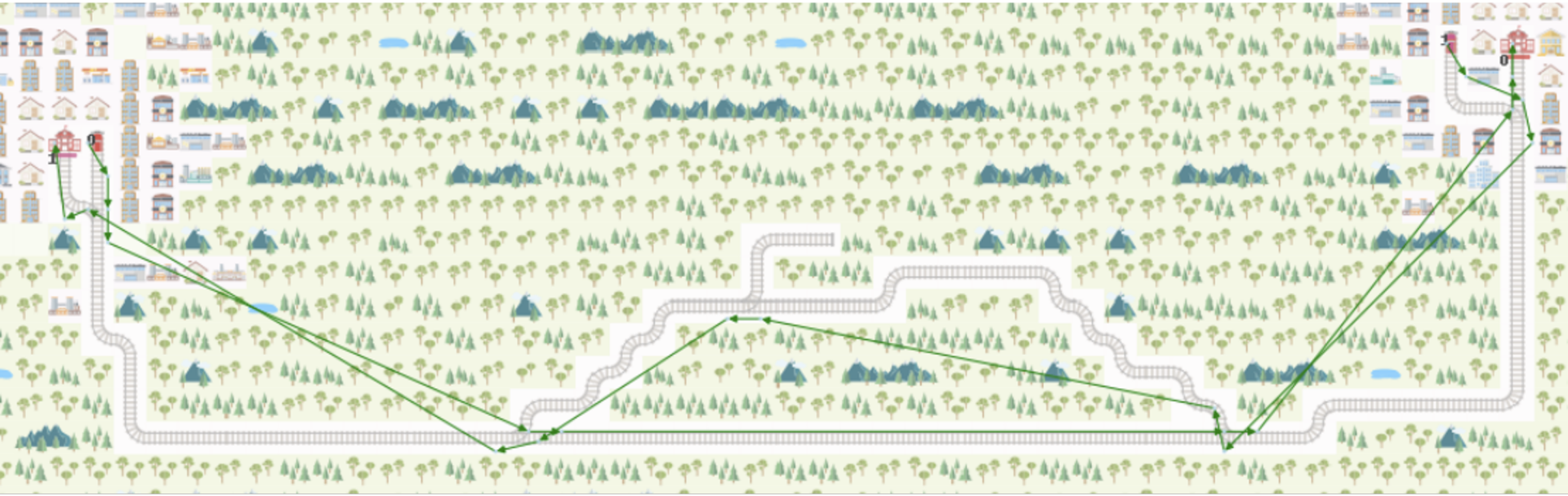
Same or opposite directions?

- expand the table with an extra dimension, labelled with the “rail” nodes which include direction. Just set to 1, rather than increment.
- Count the number of *different* directions per “grid” node

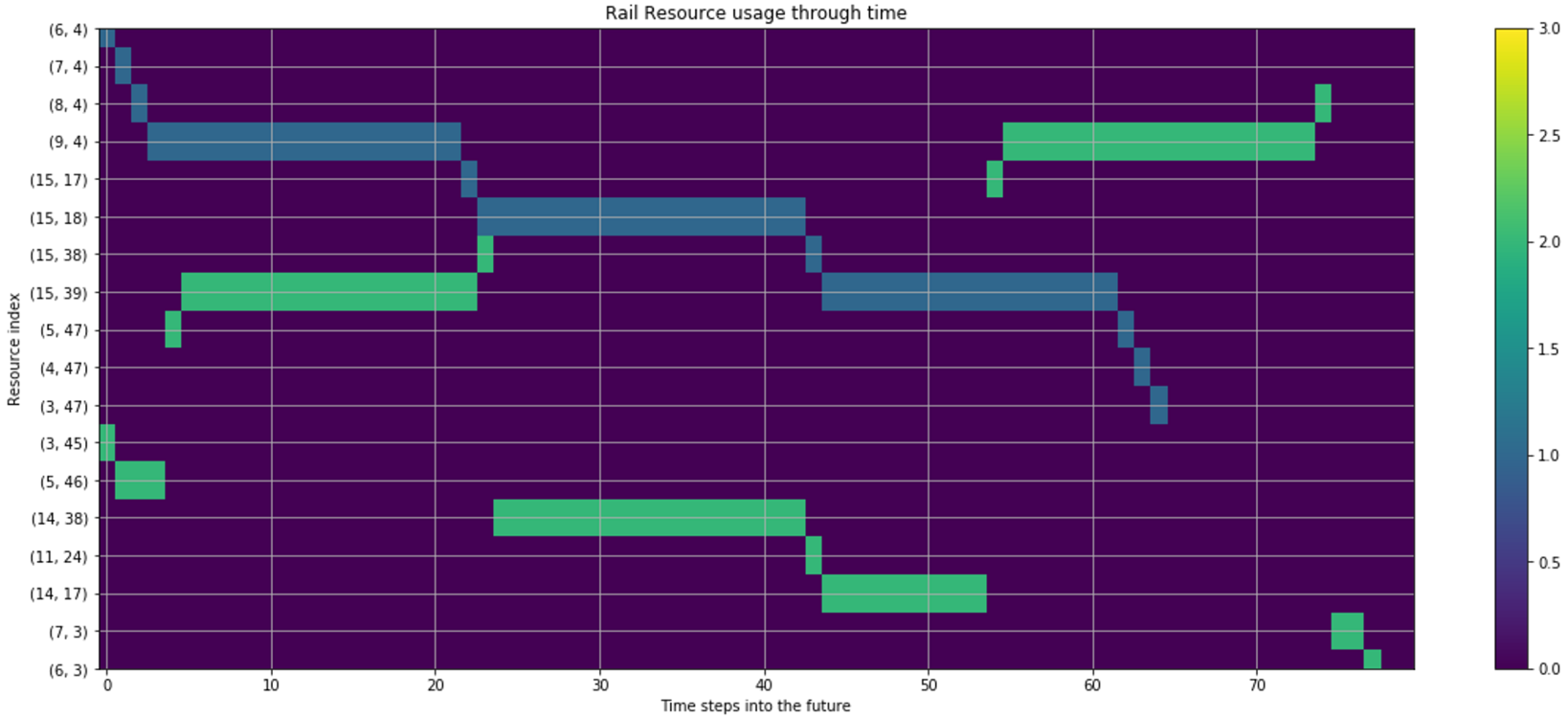
Grid	**Rail**	**t=0**	**t=1**	**t=2**	**3**
15,18	15,18,1	1	1	0	
	15,18,3	0	1	1	
	Sum:	1=ok	2=conflict	1	

- (Agents travelling in the same direction just count as one in this table)
- Dimensions of table: resources x 4 directions x timesteps

Agent 1 takes alternative Path



Resource Chart now shows no conflict

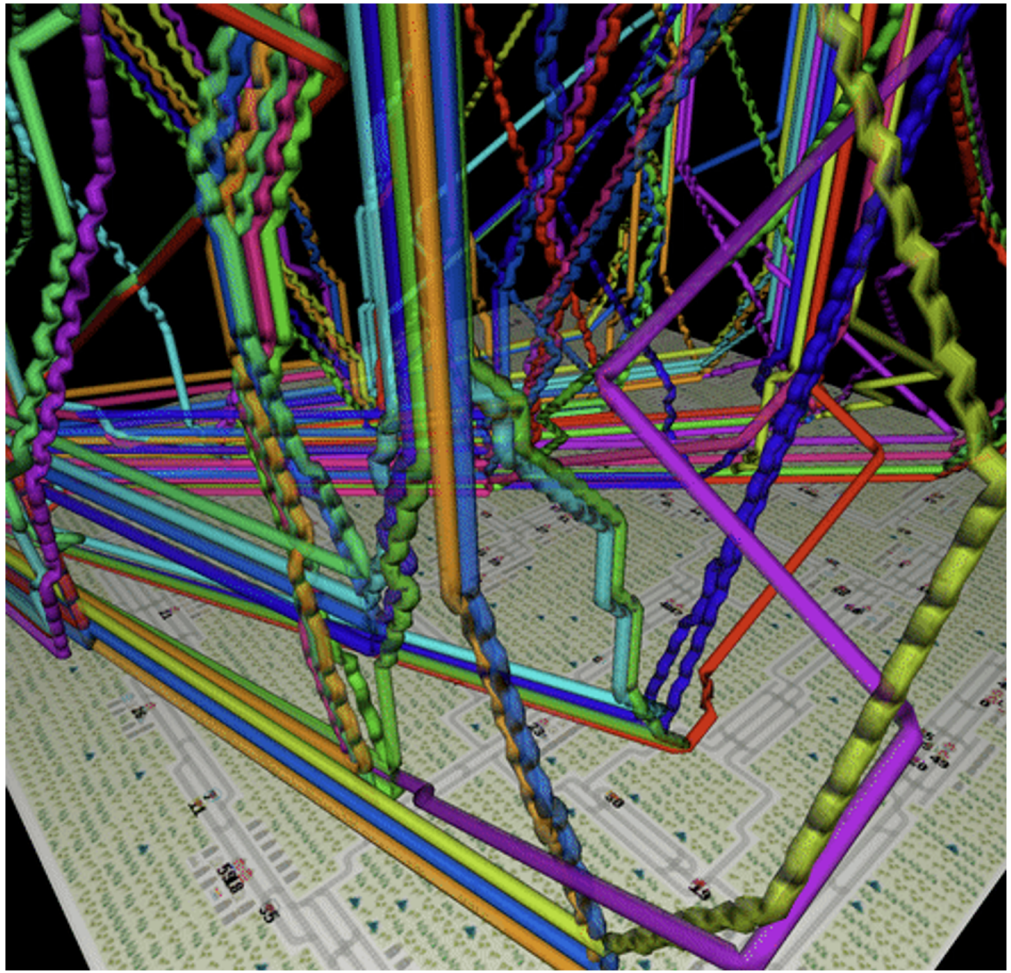


Did try to represent resource conflicts in 3d...

- 2d space + time = 3d
- Paths in “Space-Time”
- Lots of programming…
- Visually hard to interpret
- Not that meaningful:
- Just showing off...

Did try to represent resource conflicts in 3d...

- 2d space + time = 3d
- Paths in “Space-Time”
- Lots of programming…
- Visually hard to interpret
- Not that meaningful:
- Just showing off...



Resource Representation

- Contracting a path implicitly solves the deadlock detection, of agents entering a long path in opposite directions

Additional Logic needed - not covered here

- To identify fast agents stuck behind slow
  - We have the speeds, the entry time, and the “length” of each path node
  - We can calculate analytically how many steps each agent is held up
- To spot agents “jumping over each other”
  - Full-speed agents heading in opposite directions
  - “Real” Flatland env prevents this by only moving one agent at a time
  - In our static view with time dimension, it is possible again for two adjacent length-1 nodes
  - Possible solution:
    - “fill in” two time steps for each agent
- The “Pause” action
  - Represent a paused path in resource space-time
- Malfunctions
  - Calculate a probability distribution for a possibly failing agent in its predicted path
  - Probability is “pushed” into later timesteps

Larger Env

The “real” envs are quite sparse.

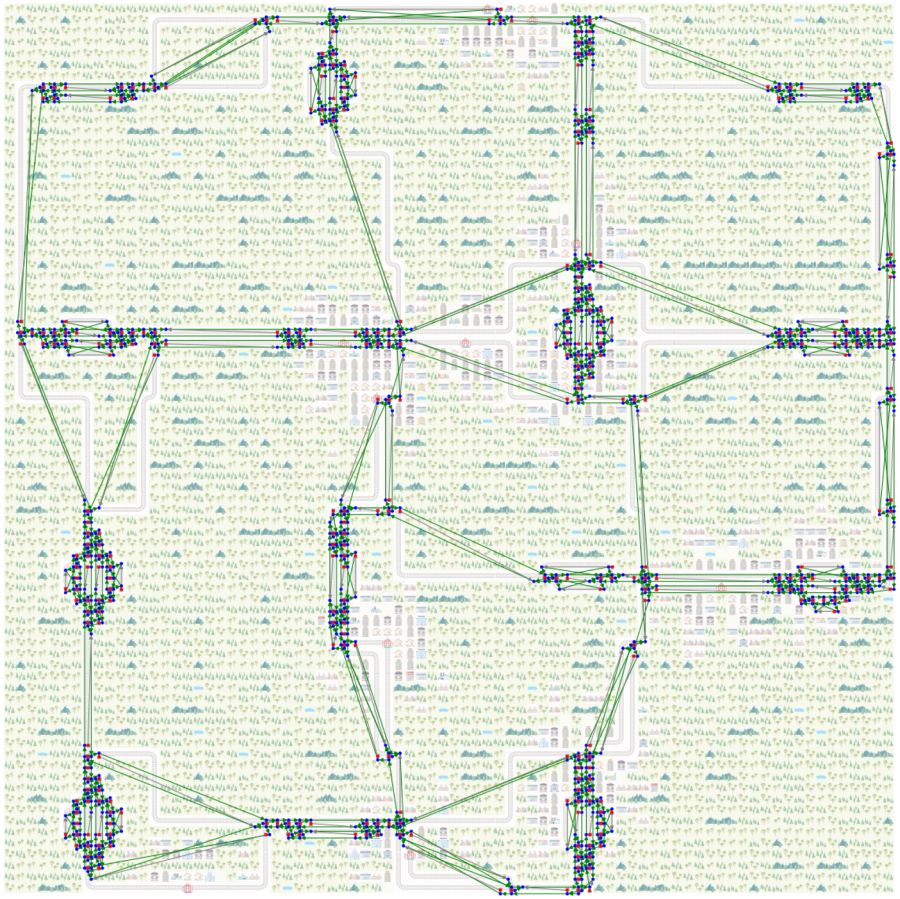
We can start to see how a graph can represent them efficiently.

80x80 env still manageable in a notebook. This has:

- 354 grid nodes / resources
- About 6% of the 6400 cells

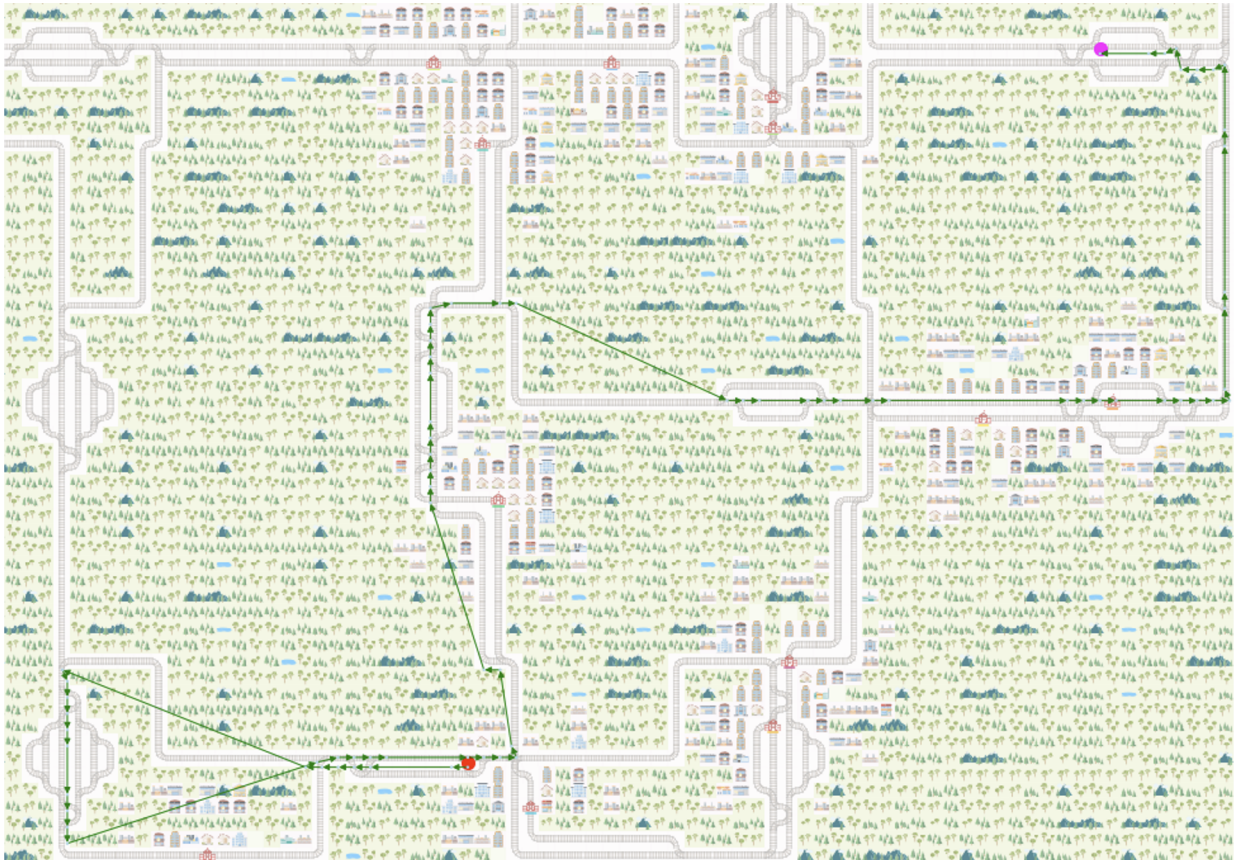
Scaling not yet formally measured.

Limits of scaling not tested.



Shortest Path still works

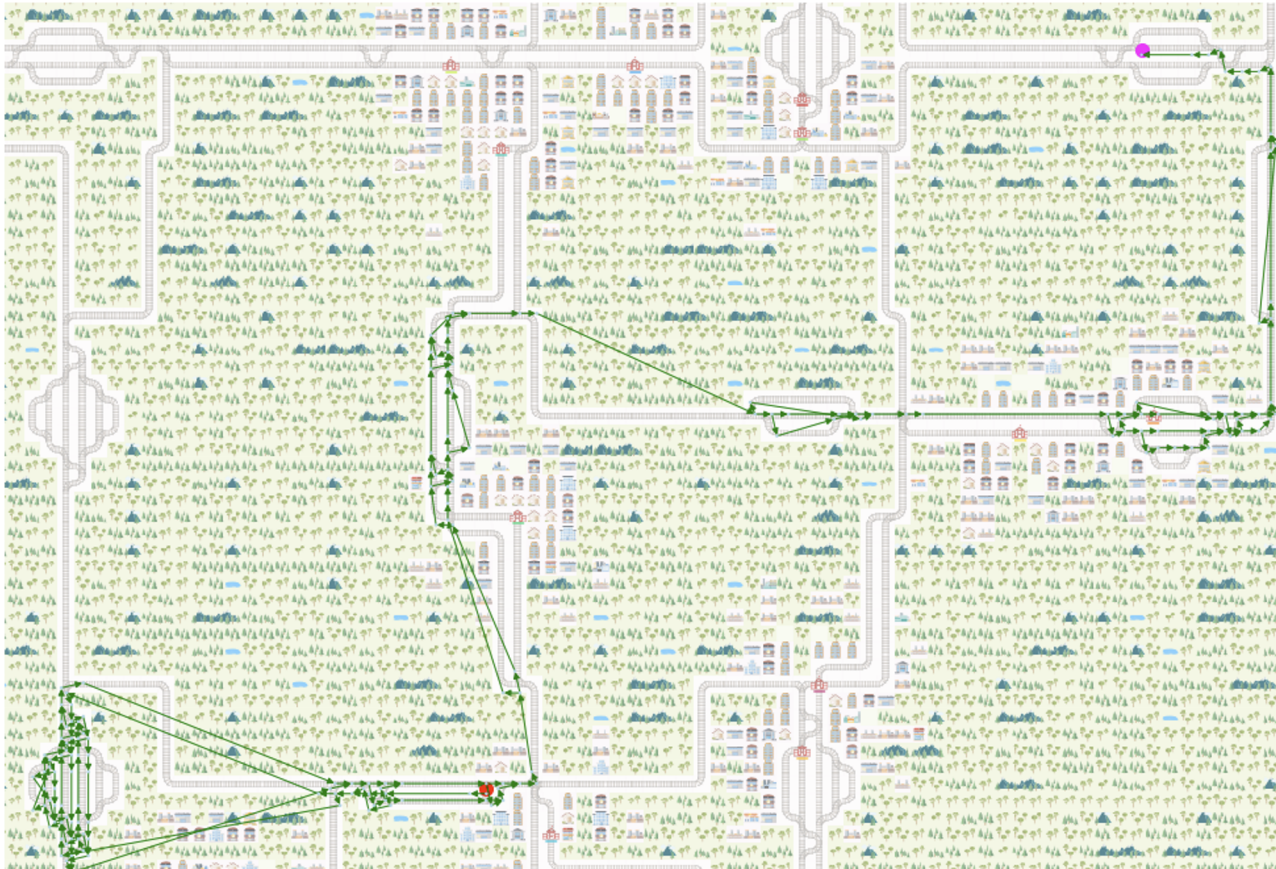
Here the initial direction is West, forcing the agent to make a loop.



k-shortest paths as a source of deviations...

Does not appear to “explore” much. This is 200 paths!

(Assume it’s generating permutations of the alternative routes - seems inefficient. Not really checked.)

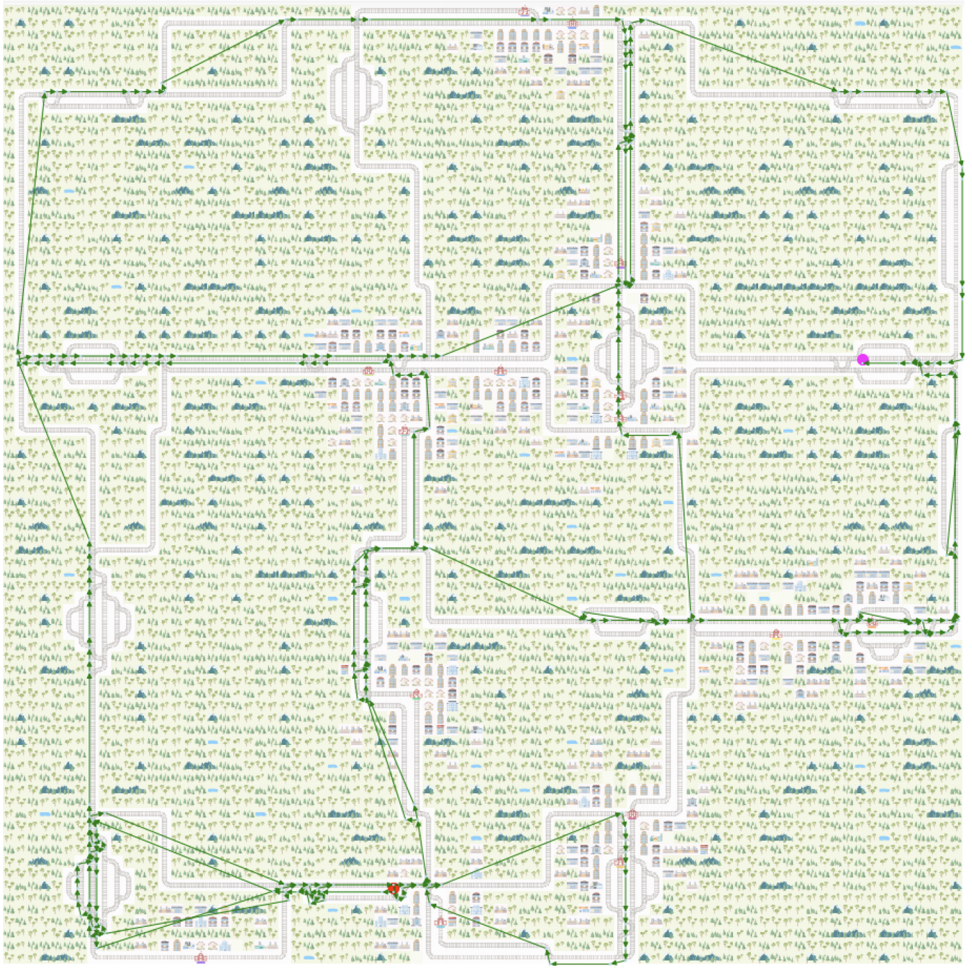


Deviation Paths

As before - at each node in the shortest path generate a new route by forcing the agent to take **the longer route.** (After that we apply no further deviations.)

This results in more “exploratory” - and longer - routes.

(Could refine to detect shorter alternatives.)

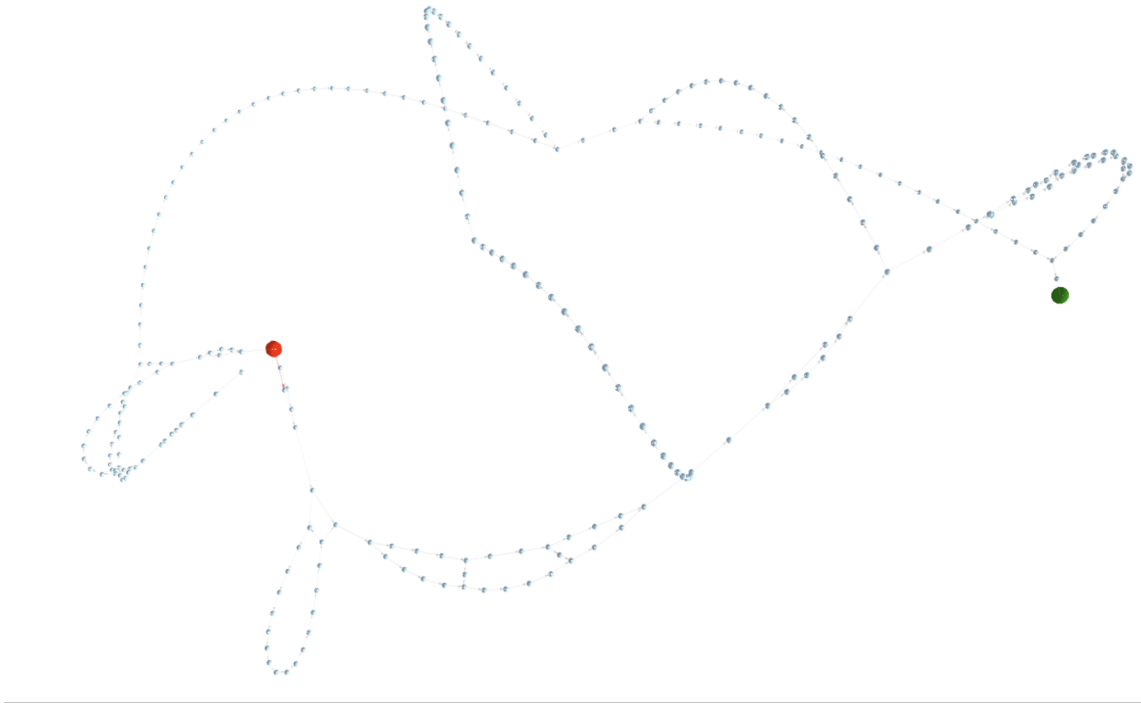


Deviation Paths - Force Layout

Here’s another illustration of the paths generated by forced deviations.

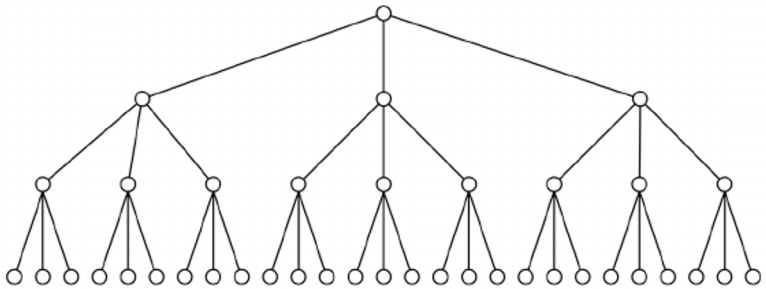
(Red = start, green = target)

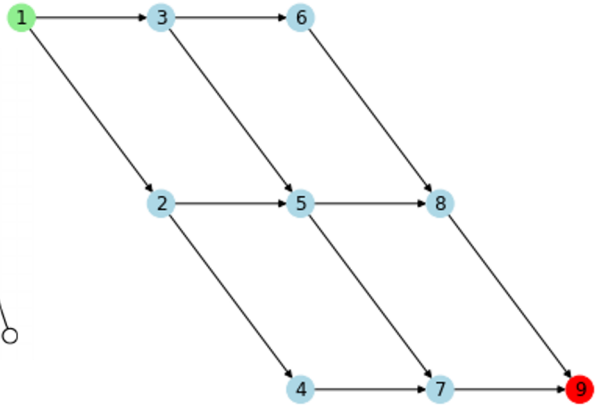
(** need to contract the linear paths!)



Graphs and RL observations

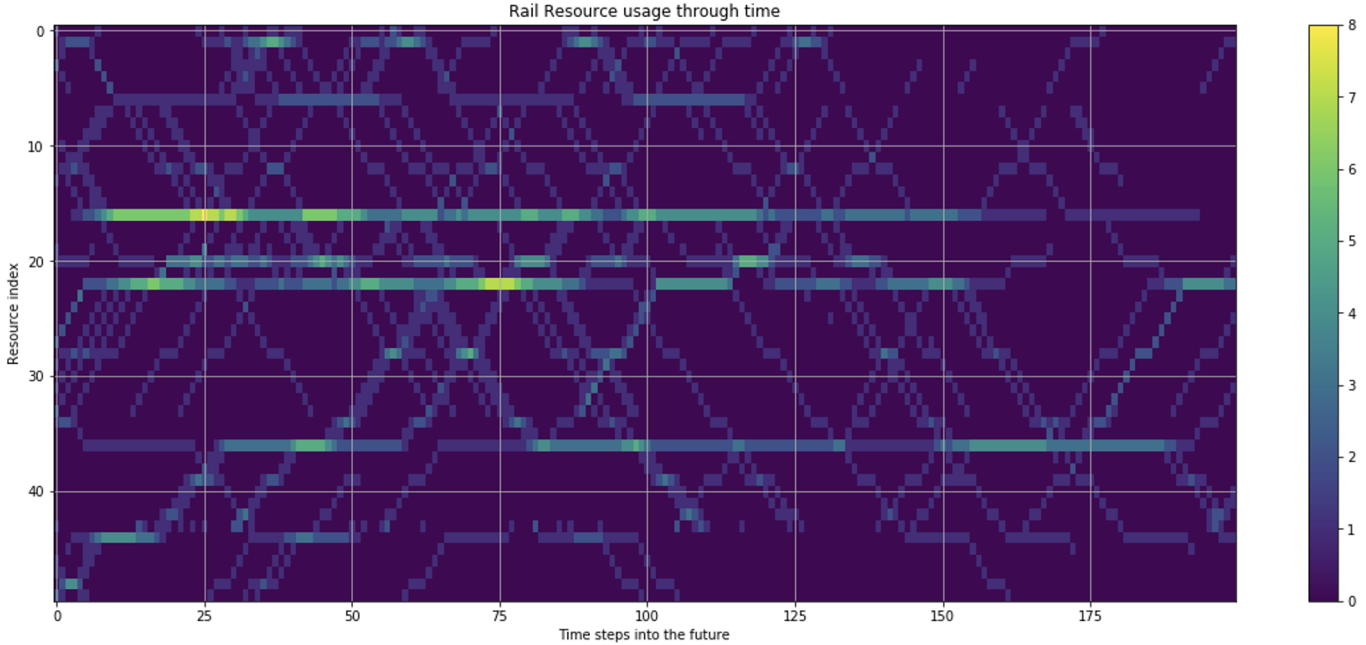
- How can we use graphs to inspire new Observations for RL?
- A Tree-based observation grows exponentially (branching factor ^ depth.)
- Deviation paths more like a “**hammock**” - since all routes end up at the target.
- (OK the recombination in a hammock / lattice is not that representative ;)





Interaction between predicted paths

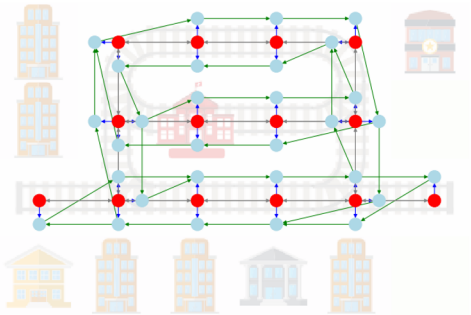
- In a larger env with lots of agents, we can see the “busiest” tracks.
- This was generated from the 80x80 env with “real” agent paths

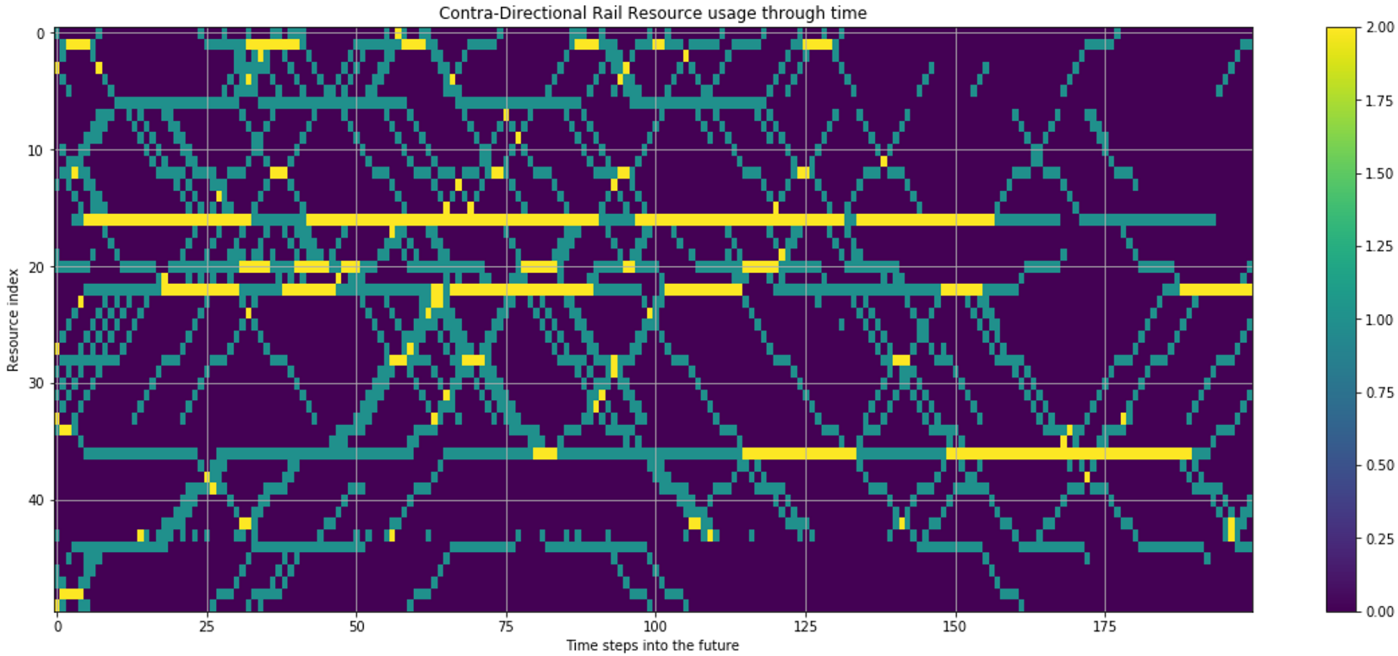


Interactions - direction-aware

By looking at which direction nodes are occupied, we can spot the deadlocks

Perhaps count numbers in each direction?





How can we apply this to a RL solution?

- No solutions here - just ideas!
- We presented a “visual” way of thinking of the Flatland Challenge as a graph / network + resource problem
- If we accept this problem structure,
  - Can a deep learning approach infer the structure by itself from pixels?
  - Or should we present it with this structure directly?
  - Can we help it learn it?
  - Eg
    - Give the agent the resource “model”
    - Shape the reward based on anticipated conflicts
    - Present an agent with a “conflict score” for each alternative path
    - Agents’ preferred paths go into a global observation
    - Iteration over planning / communication actions before execution of movement

Libraries & Code (again)

- Python - **NetworkX** - Graph library.
- Javascript
  - **Vasco Asturiano** - https://github.com/vasturiano/3d-force-graph
    - **@vastur (Twitter)**
  - **Three.js, d3js**

Notebooks & html/js wrappers in **flatland repo** under `flatland/notebooks`

Currently in **branch** `223_UpdateEditor_55_notebooks`

(Will probably create a `visualisations` folder or repo)

Working with Javascript and Python

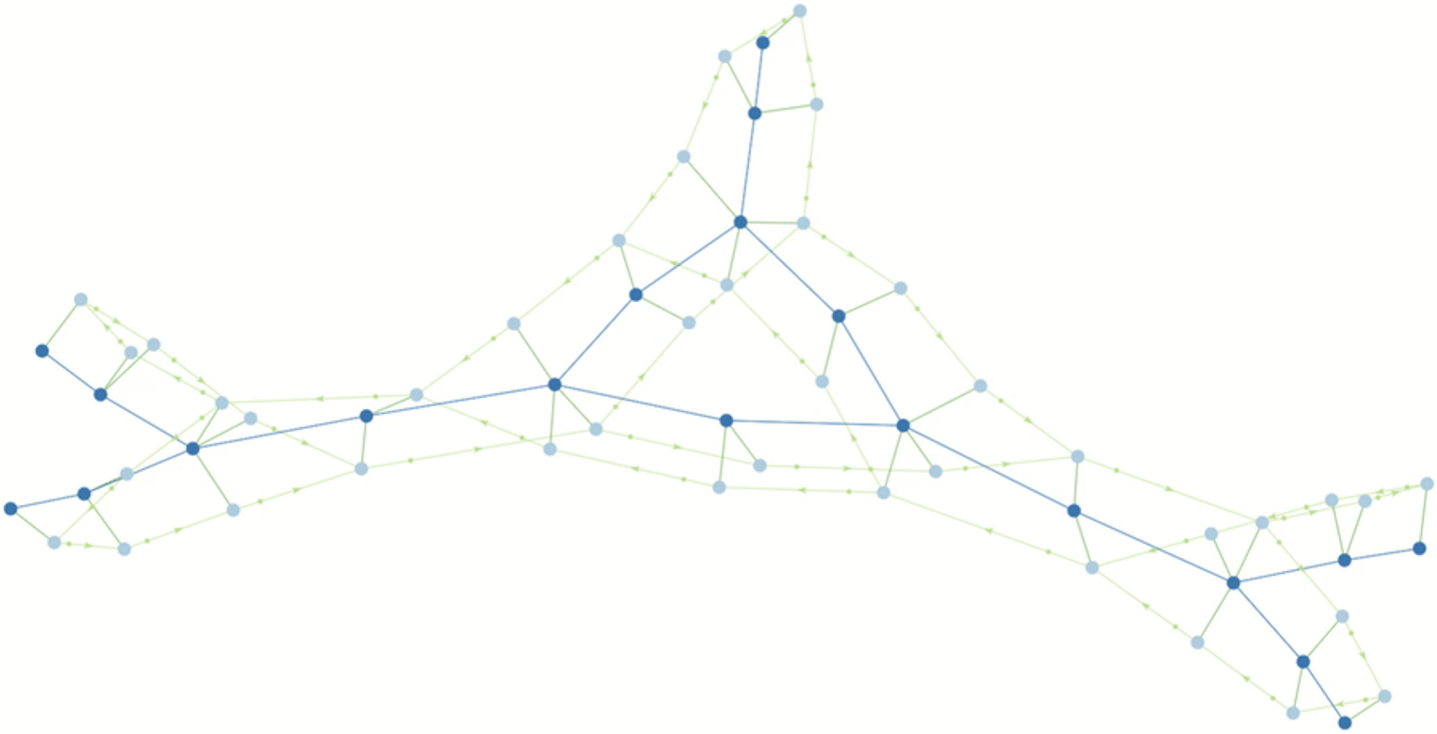
(for the python programmer…)

- We found it hard to integrate JS animations in notebooks
  - Sometimes possible, but fiddly and unstable
  - The JS visualisations are already quite heavy, as is the JS in Jupyter Notebooks
- Best to just serve the JS from a simple http server:
  - `python -m http.server`
  - Browse to http://localhost:8000
- Then fairly easy to cut & paste JS examples from Vasco, Three.JS etc
- Can't serve from Colab, Binder etc because need a second port :(
- Easy: Flatland -> NetworkX Graph -> JSON -> JS
- (Just for fun & temporary - http://estelle.nsupdate.info:38471/graph1.html?file=test-large-red.json)

Summary

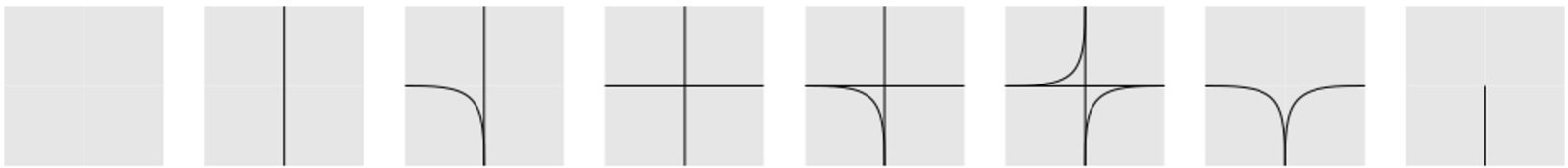
- A way of representing Flatland envs with a Directed Graph
  - Correctly capturing junction / slip constraints
  - Extra “grid” nodes represent resources
  - Simplifying (sparse) envs by removing blank cells and contracting long paths
  - Use of standard representation admits integration with existing libs, algos, & visuals
- Generating alternative paths
  - “Hammock” rather than tree, avoid exponential growth
  - Use deviations + shortest path mechanism
- Planning / Predicting resource conflicts
  - Quantify resource contention (same direction)
  - Predict deadlocks (opposite direction)
  - Simplifying 2d env -> list of nodes allows 1d + time = 2d chart

Thanks for watching!



Welcome to

FLATLAND


empty
straight
simple switch
diamond crossing
single slip switch
double slip switch
symmetrical switch
dead end

Action

Observation

Reward

Done

Action
Action
Action
...

Observation
Observation
Observation
...

Reward
Reward
Reward
...

Done
Done
Done
...

```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```

```
observations = {
    0: obs_0,
    1: obs_1,
    2: obs_2
}

rewards = {
    0: reward_1,
    1: reward_2,
    2: reward_3
}

done = {
    0: done_1,
    1: done_2,
    2: done_3
}
```

```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```

```
observations = {
    0: obs_0,
    1: obs_1,
    2: obs_2
}

rewards = {
    0: reward_1,
    1: reward_2,
    2: reward_3
}

done = {
    0: done_1,
    1: done_2,
    2: done_3
}
```

```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```

```
observations = {
    0: obs_0,
    1: obs_1,
    2: obs_2
}

rewards = {
    0: reward_1,
    1: reward_2,
    2: reward_3
}

done = {
    0: done_1,
    1: done_2,
    2: done_3
}
```

```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```

DO_NOTHING = 0

(or turn around)

```
observations = {
    0: obs_0,
    1: obs_1,
    2: obs_2
}

rewards = {
    0: reward_1,
    1: reward_2,
    2: reward_3
}

done = {
    0: done_1,
    1: done_2,
    2: done_3
}
```

```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```

MOVE_LEFT = 1

DO_NOTHING = 0

```
observations = {
    0: obs_0,
    1: obs_1,
    2: obs_2
}

rewards = {
    0: reward_1,
    1: reward_2,
    2: reward_3
}

done = {
    0: done_1,
    1: done_2,
    2: done_3
}
```

```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```

MOVE_FORWARD = 2

MOVE_LEFT = 1

DO_NOTHING = 0

```
observations = {
    0: obs_0,
    1: obs_1,
    2: obs_2
}

rewards = {
    0: reward_1,
    1: reward_2,
    2: reward_3
}

done = {
    0: done_1,
    1: done_2,
    2: done_3
}
```

```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```

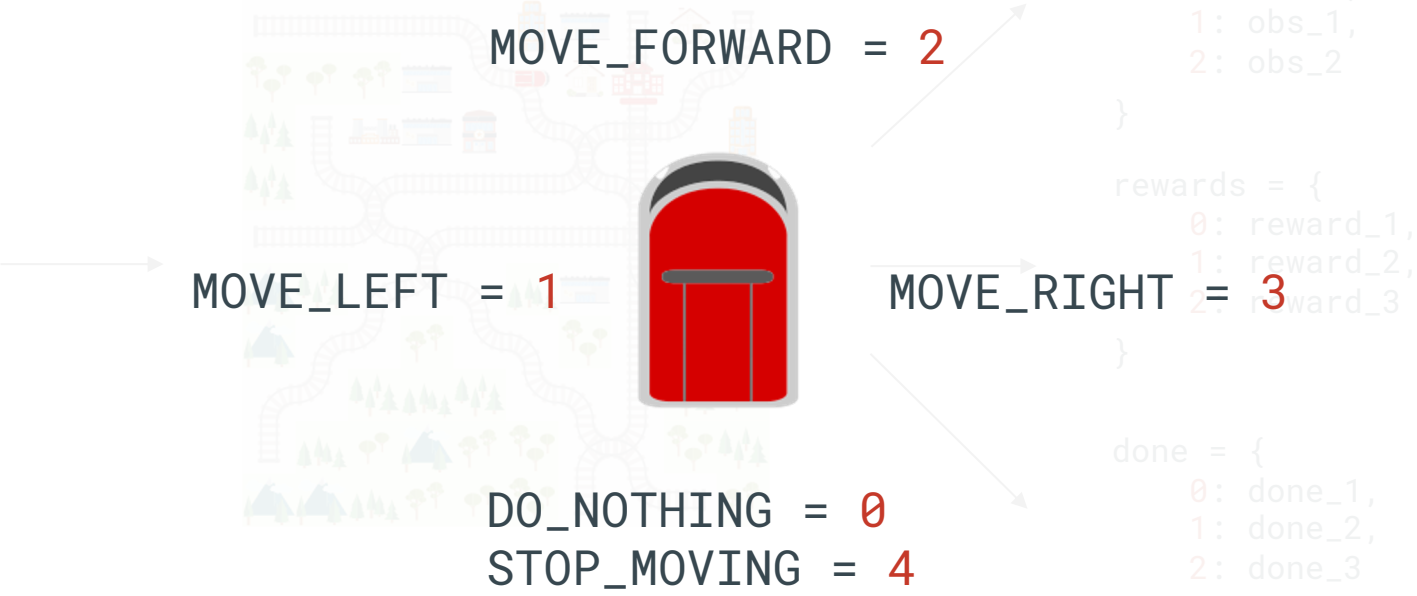
MOVE_FORWARD = 2

MOVE_LEFT = 1

MOVE_RIGHT = 3

DO_NOTHING = 0

```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```



```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```

```
observations = {
    0: obs_0,
    1: obs_1,
    2: obs_2
}

rewards = {
    0: reward_1,
    1: reward_2,
    2: reward_3
}

done = {
    0: done_1,
    1: done_2,
    2: done_3
}
```

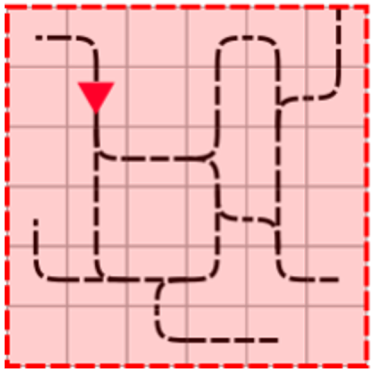
```
actions = {
    0: action_0,
    1: action_1,
    2: action_2
}
```

```
observations = {
    0: obs_0,
    1: obs_1,
    2: obs_2
}
```

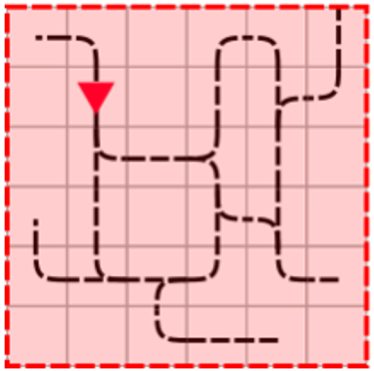
```
rewards = {
    0: reward_1,
    1: reward_2,
    2: reward_3
}
```

```
done = {
    0: done_1,
    1: done_2,
    2: done_3
}
```

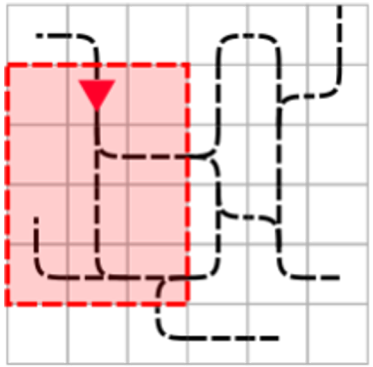
Global



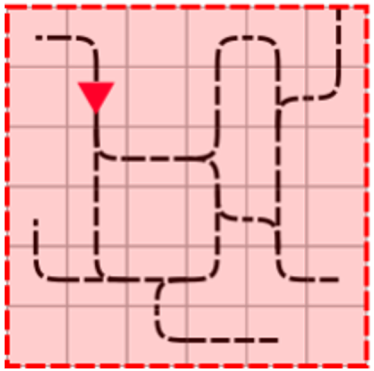
Global



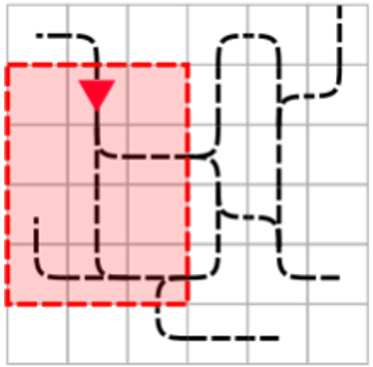
Local



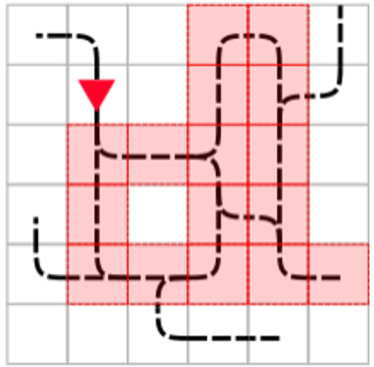
Global

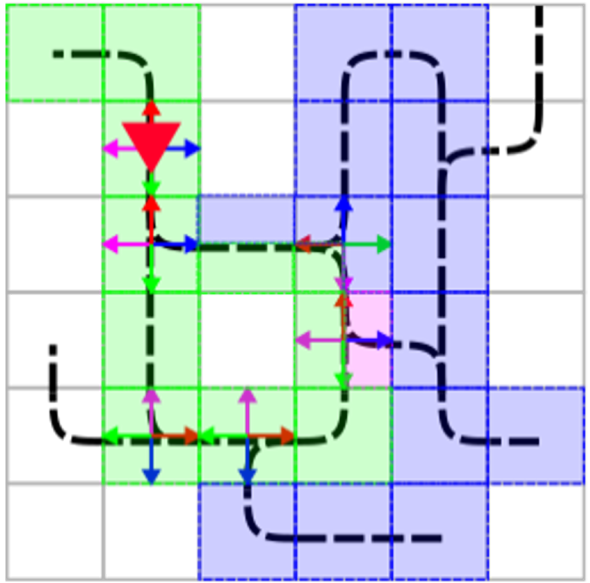


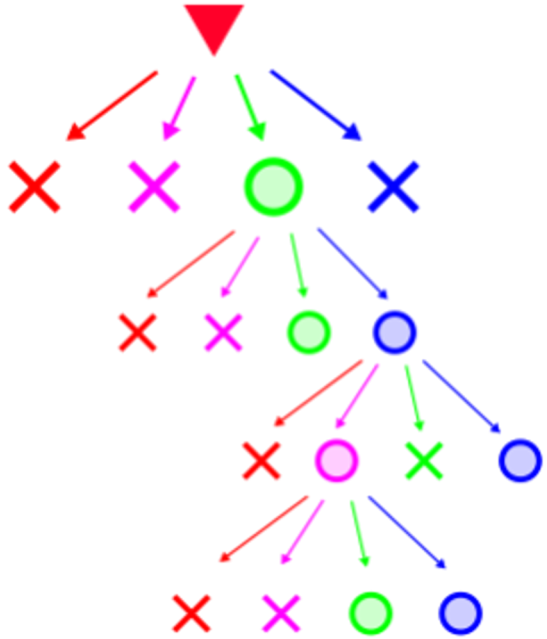
Local

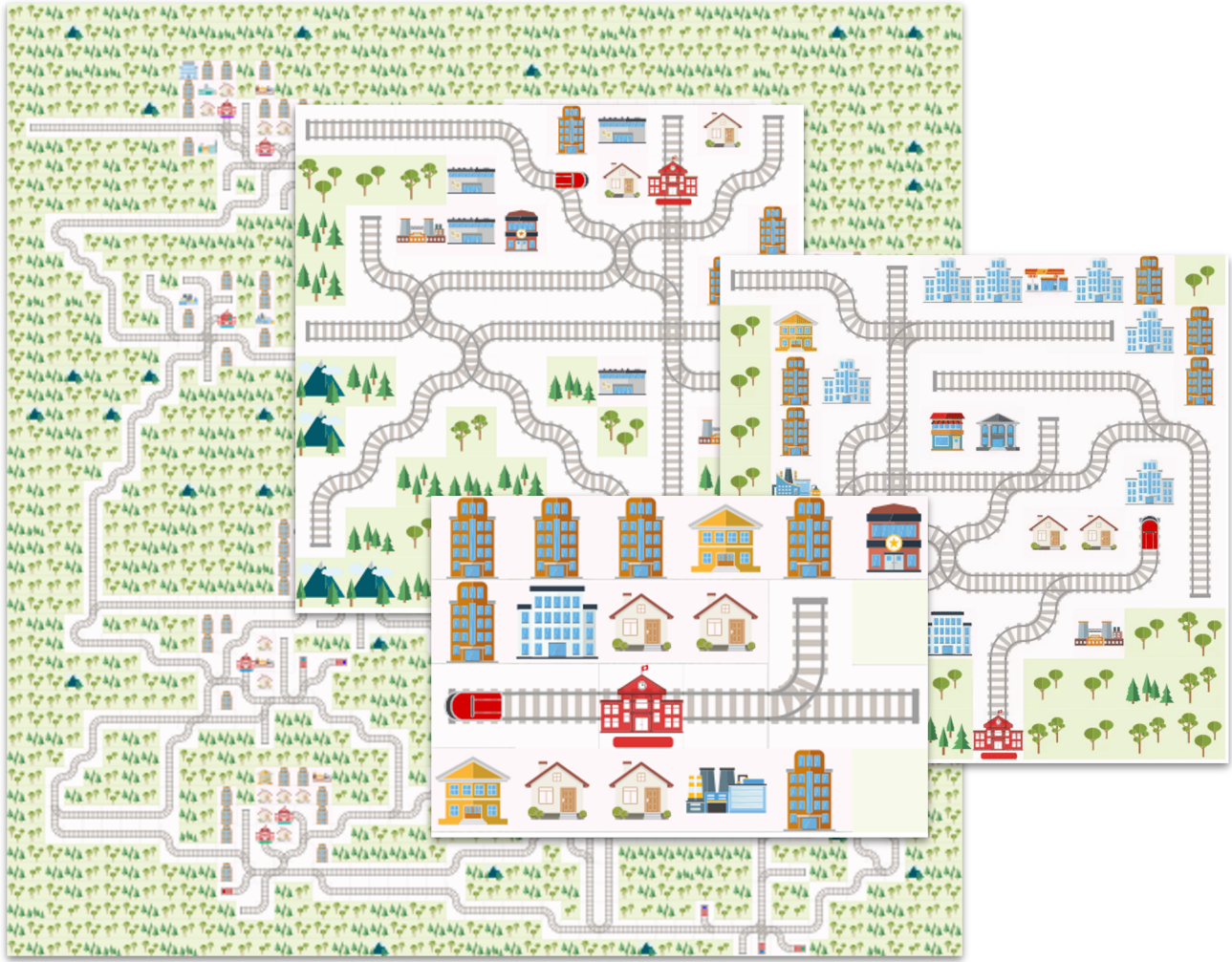


Tree



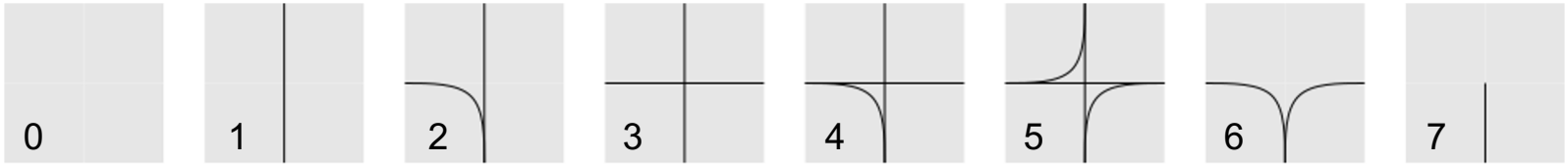




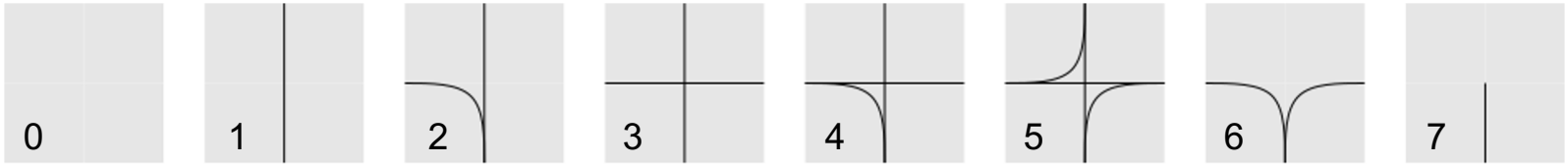


```
specs = [[(0, 0),   (0, 0),     (0, 0),     (0, 0),     (0, 0),     (0, 0)],
         [(0, 0),   (0, 0),     (0, 0),     (0, 0),     (7, 0),     (0, 0)],
         [(7, 270), (1, 90),    (1, 90),    (1, 90),    (2, 90),    (7, 90)],
         [(0, 0),   (0, 0),     (0, 0),     (0, 0),     (0, 0),     (0, 0)]]

rail_generator = rail_from_manual_specifications_generator(specs)

fixed_env = RailEnv(width=6, height=4,
                    rail_generator=rail_generator,
                    number_of_agents=1)
```

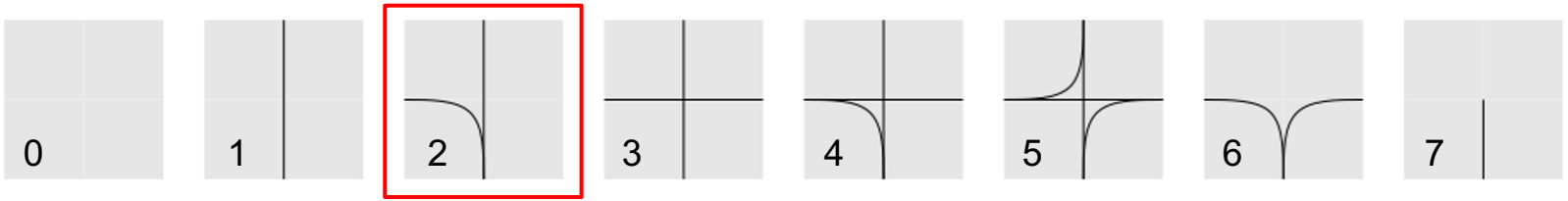
```
specs = [[(0, 0),   (0, 0),     (0, 0),     (0, 0),     (0, 0),     (0, 0)],
         [(0, 0),   (0, 0),     (0, 0),     (0, 0),     (7, 0),     (0, 0)],
         [(7, 270), (1, 90),    (1, 90),    (1, 90),    (2, 90),    (7, 90)],
         [(0, 0),   (0, 0),     (0, 0),     (0, 0),     (0, 0),     (0, 0)]]

rail_generator = rail_from_manual_specifications_generator(specs)

fixed_env = RailEnv(width=6, height=4,
                    rail_generator=rail_generator,
                    number_of_agents=1)
```

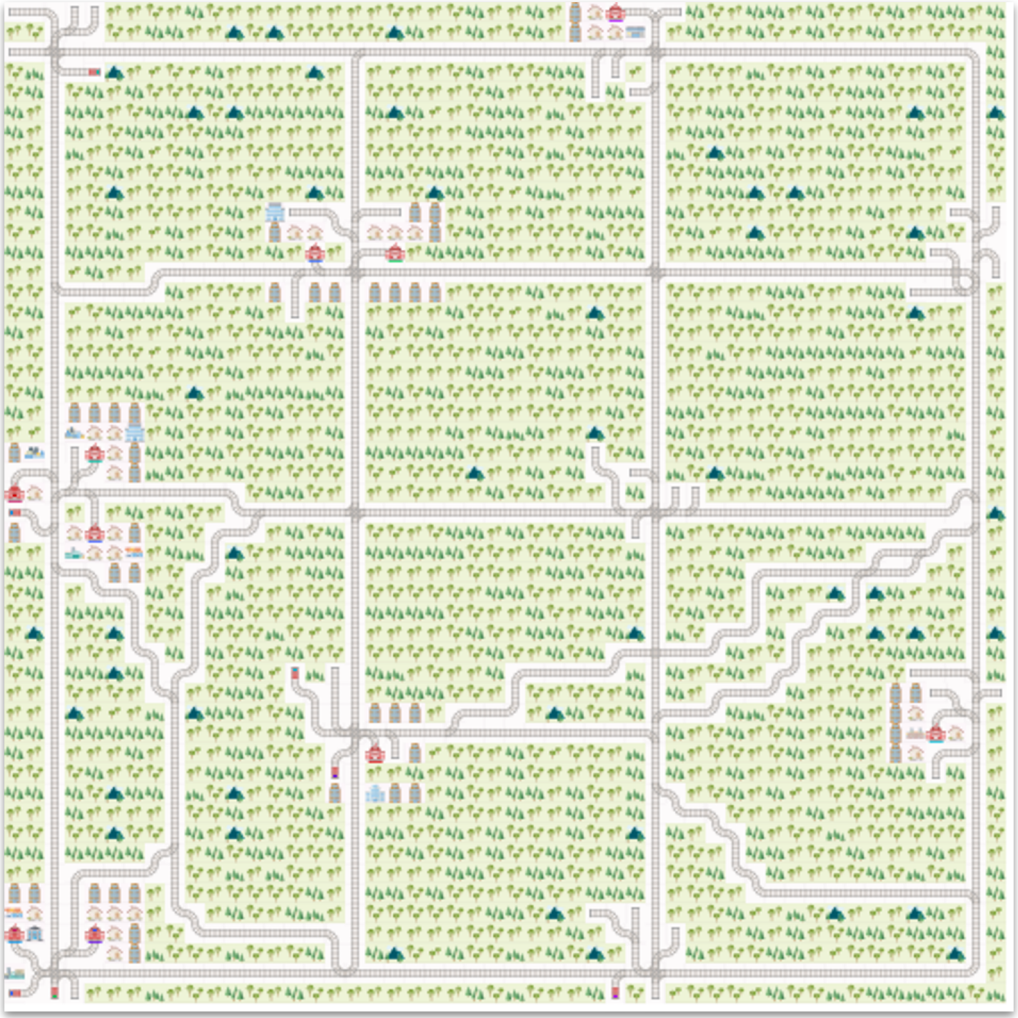


```
specs = [[(0, 0),   (0, 0),   (0, 0),   (0, 0),   (0, 0),   (0, 0)],
         [(0, 0),   (0, 0),   (0, 0),   (0, 0),   (7, 0),   (0, 0)],
         [(7, 270), (1, 90),  (1, 90),  (1, 90),  (2, 90),  (7, 90)],
         [(0, 0),   (0, 0),   (0, 0),   (0, 0),   (0, 0),   (0, 0)]]
```



```
specs = [[(0, 0),   (0, 0),   (0, 0),   (0, 0),   (0, 0),   (0, 0)],
         [(0, 0),   (0, 0),   (0, 0),   (0, 0),   (7, 0),   (0, 0)],
         [(7, 270), (1, 90),  (1, 90),  (1, 90),  (2, 90),  (7, 90)],
         [(0, 0),   (0, 0),   (0, 0),   (0, 0),   (0, 0),   (0, 0)]]
```

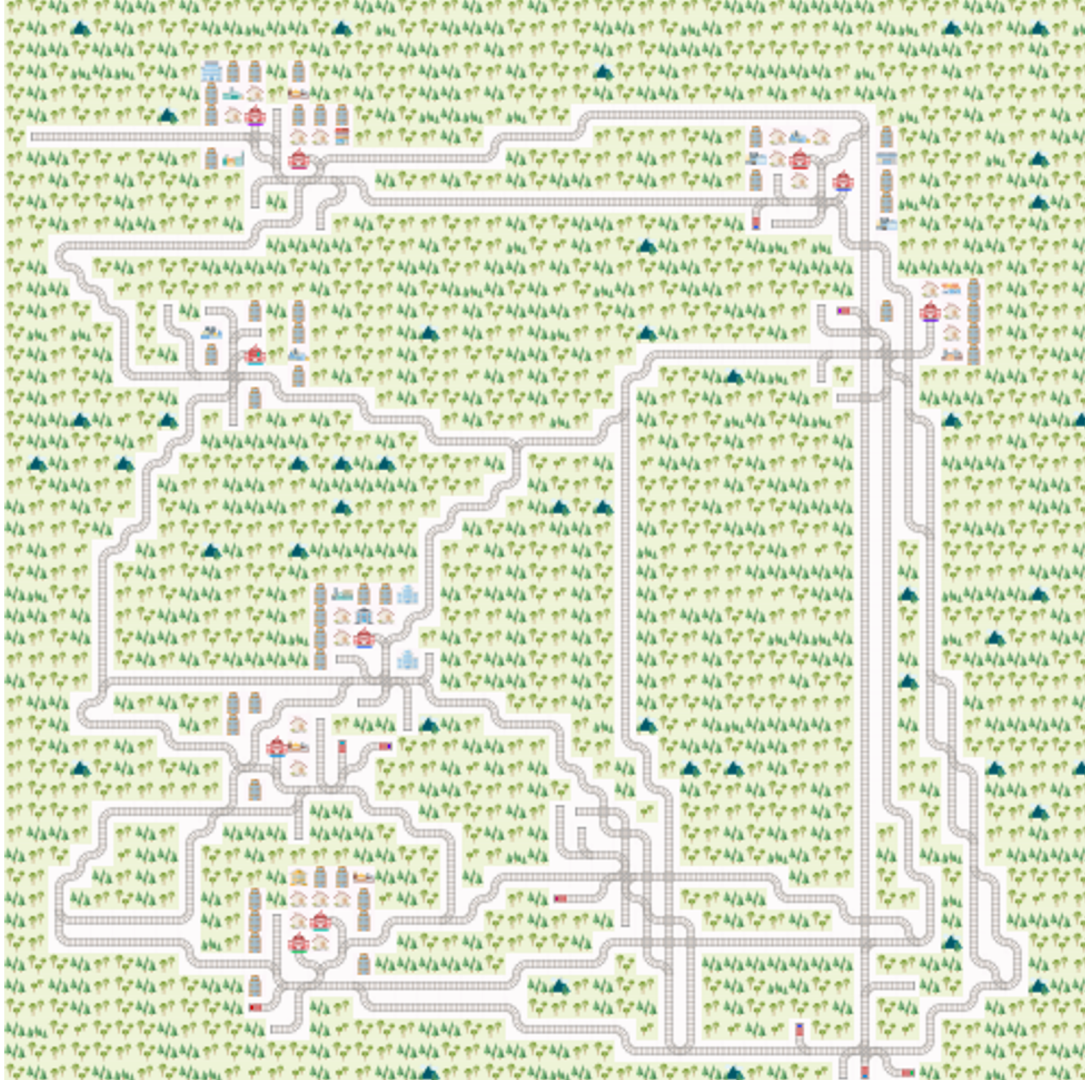




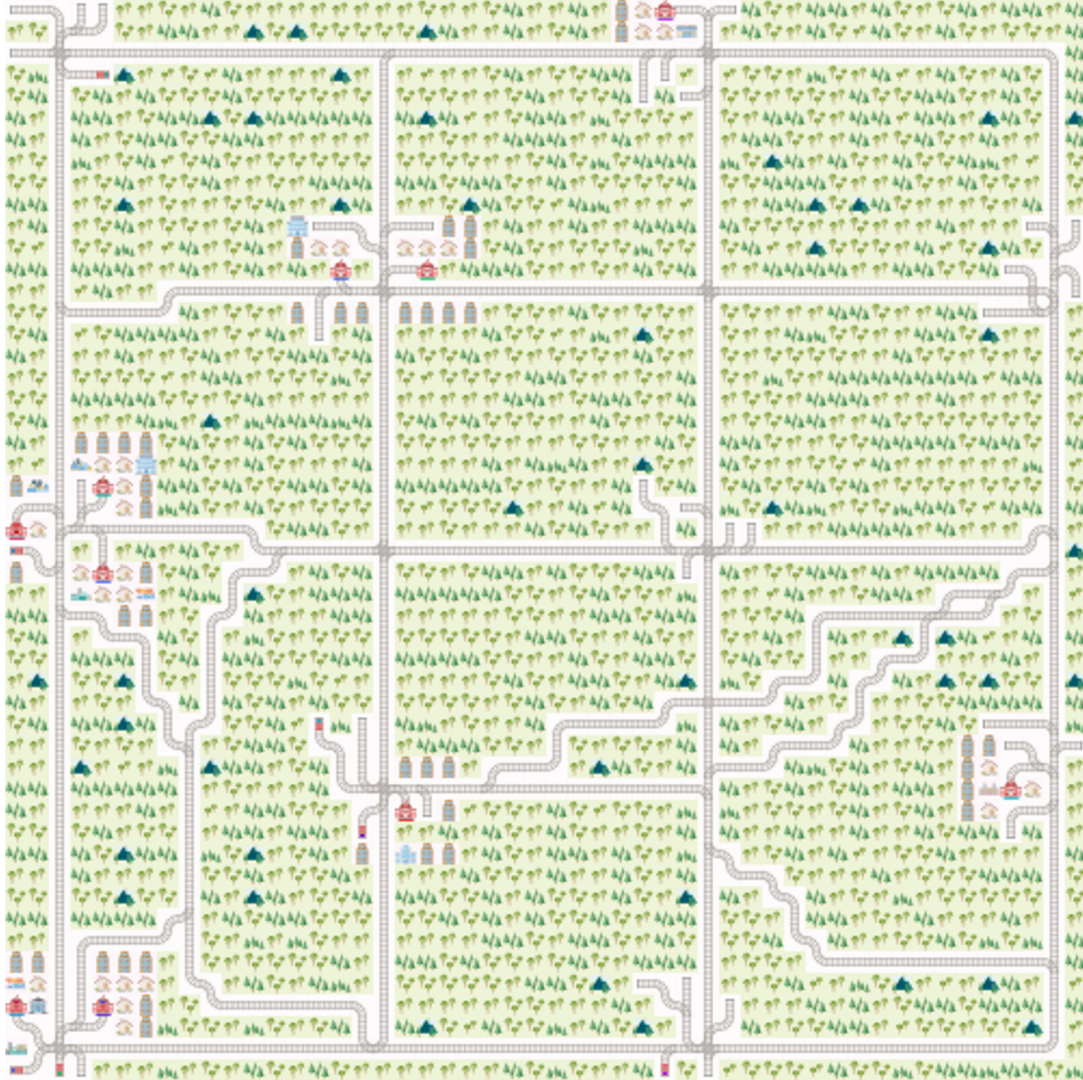
```
def rail_from_manual_specifications_generator(rail_spec)

def random_rail_generator(cell_type_relative_proportion, seed)

def complex_rail_generator(nr_start_goal, nr_extra, min_dist, max_dist, seed)

def sparse_rail_generator(max_num_cities, grid_mode,
max_rails_between_cities,
                          max_rails_in_city, seed)
```

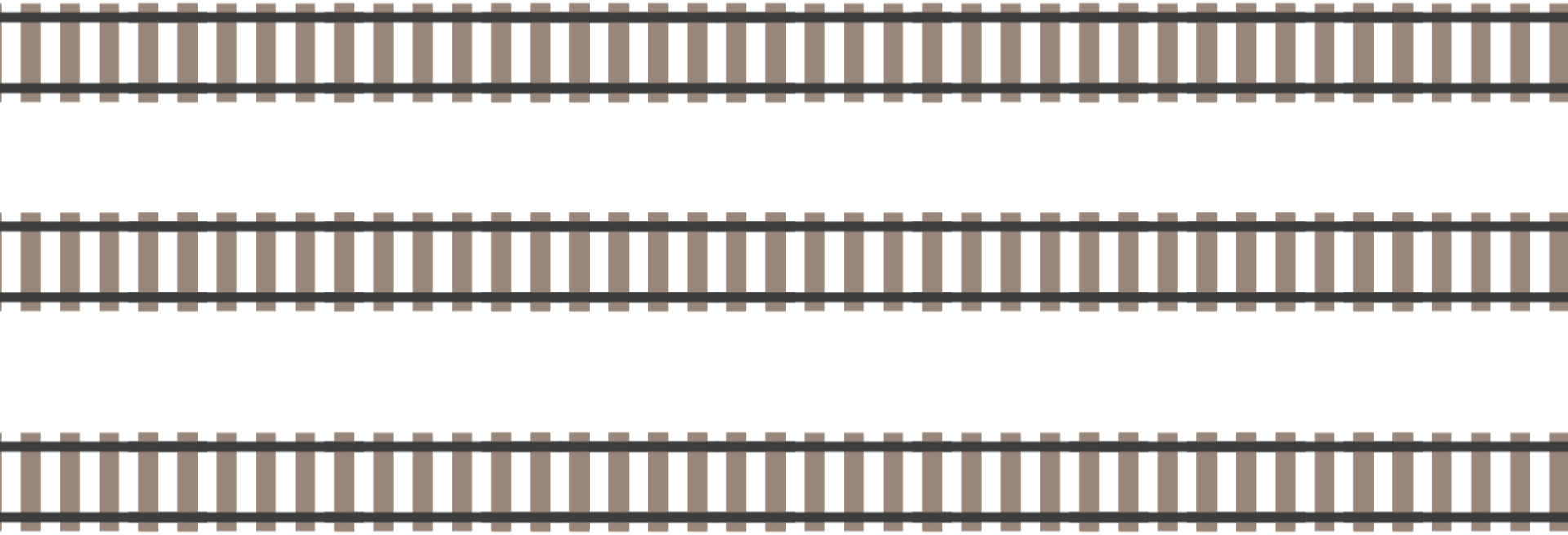
```
def rail_from_manual_specifications_generator(rail_spec)

def random_rail_generator(cell_type_relative_proportion, seed)

def complex_rail_generator(nr_start_goal, nr_extra, min_dist, max_dist, seed)

def sparse_rail_generator(
    max_num_cities=5,  # Max number of cities to build
    max_rails_between_cities=4,  # Max number of rails connecting to a city
    max_rails_in_city=4,  # Number of parallel tracks in the city
    grid_mode=False,  # Distribution of cities
    seed=5,  # Random seed
)
```

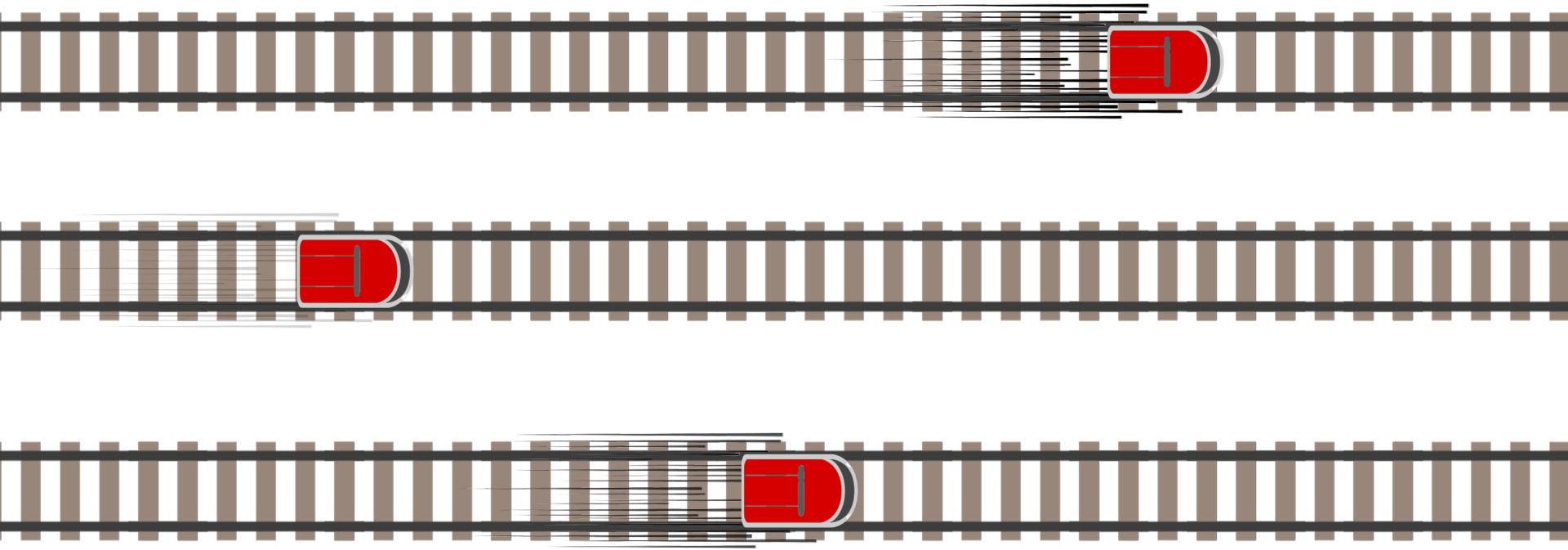
```
sparse_rail_generator(
    max_num_cities=5,
    max_rails_between_cities=4,
    max_rails_in_city=4,
    grid_mode=False
)
```



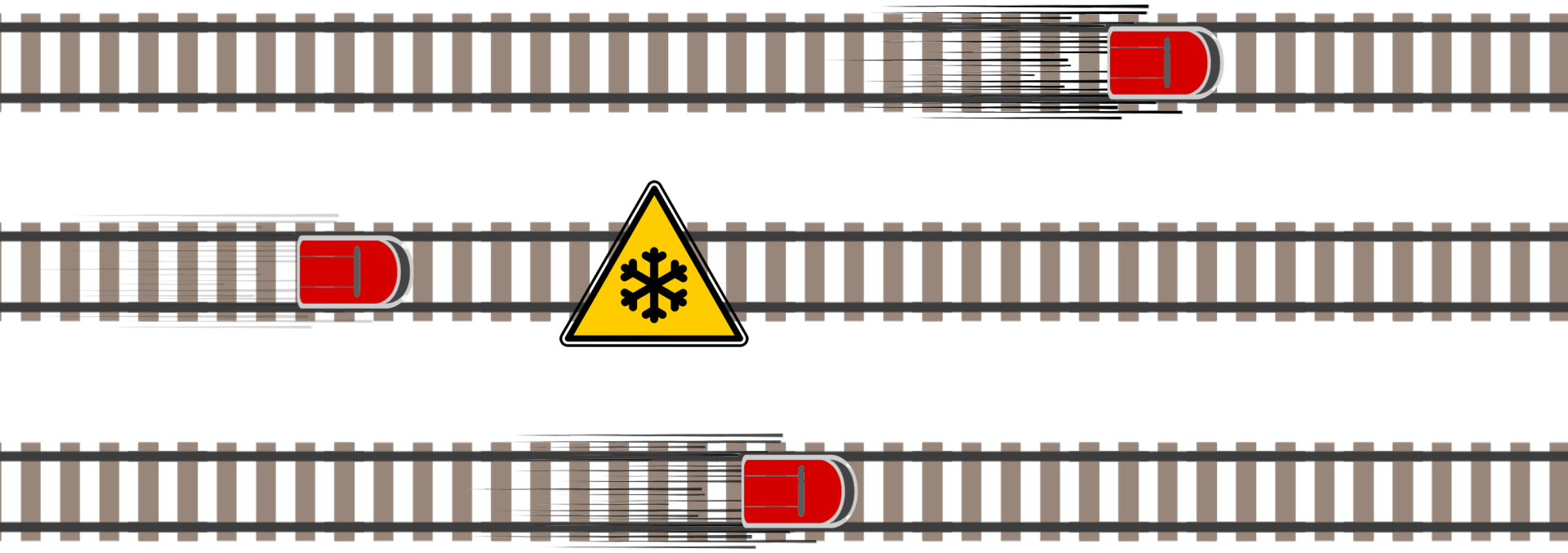
```
sparse_rail_generator(
    max_num_cities=5,
    max_rails_between_cities=4,
    max_rails_in_city=4,
    grid_mode=True
)
```







Different speed profiles!



Different speed profiles!
Stochastic events!

```
sparse_rail_generator = sparse_rail_generator(
    max_num_cities=5,  # Max number of cities to build
    max_rails_between_cities=4,  # Max number of rails connecting to a city
    max_rails_in_city=4,  # Number of parallel tracks in the city
    grid_mode=False,  # Distribution of cities
    seed=5,  # Random seed
)

speed_ratio_map = {
    1.: 0.25,  # Fast passenger train
    1. / 2.: 0.25,  # Fast freight train
    1. / 3.: 0.25,  # Slow commuter train
    1. / 4.: 0.25  # Slow freight train
}

stochastic_data = {
    'prop_malfunction': 0.5,  # Percentage of defective agents
    'malfunction_rate': 30,  # Rate of malfunction occurrence
    'min_duration': 3,  # Minimal duration of malfunction
    'max_duration': 10  # Max duration of malfunction
}
```

Getting Started with Flatland

> **Flatland** is an environment for developing and comparing multi-agent reinforcement learning algorithms in gridworlds.

This repository contains notebooks to get you started on the right track with the Flatland environment, in order to take part in the Alcrowd Flatland Challenge.

If you want to dive into challenge baselines right away, check out the various approaches below.

Discovering Flatland

Part 1: The Rail Environment

launch binder Open in Colab

- Create a `RailEnv` environment and render it
- Check out the default observations
- "Train" a random agent

Part 2: Observations & Predictions

launch binder Open in Colab

- Finding suitable observations
- Using predictions
- Crafting custom observations and predictions

Part 3: Level Generation

launch binder Open in Colab

- Creating random rail networks
- Creating schedules
- Adjusting size and difficulty

Part 4: Malfunctions

launch binder Open in Colab

- Introducing stochastic malfunctions
- Handling malfunctions

Part 5: Speed Profiles

launch binder Open in Colab

- Handling agent speed
- Handling partial moves

Coming soon!

Your turn!

https://flatlandrl-docs.aicrowd.com/

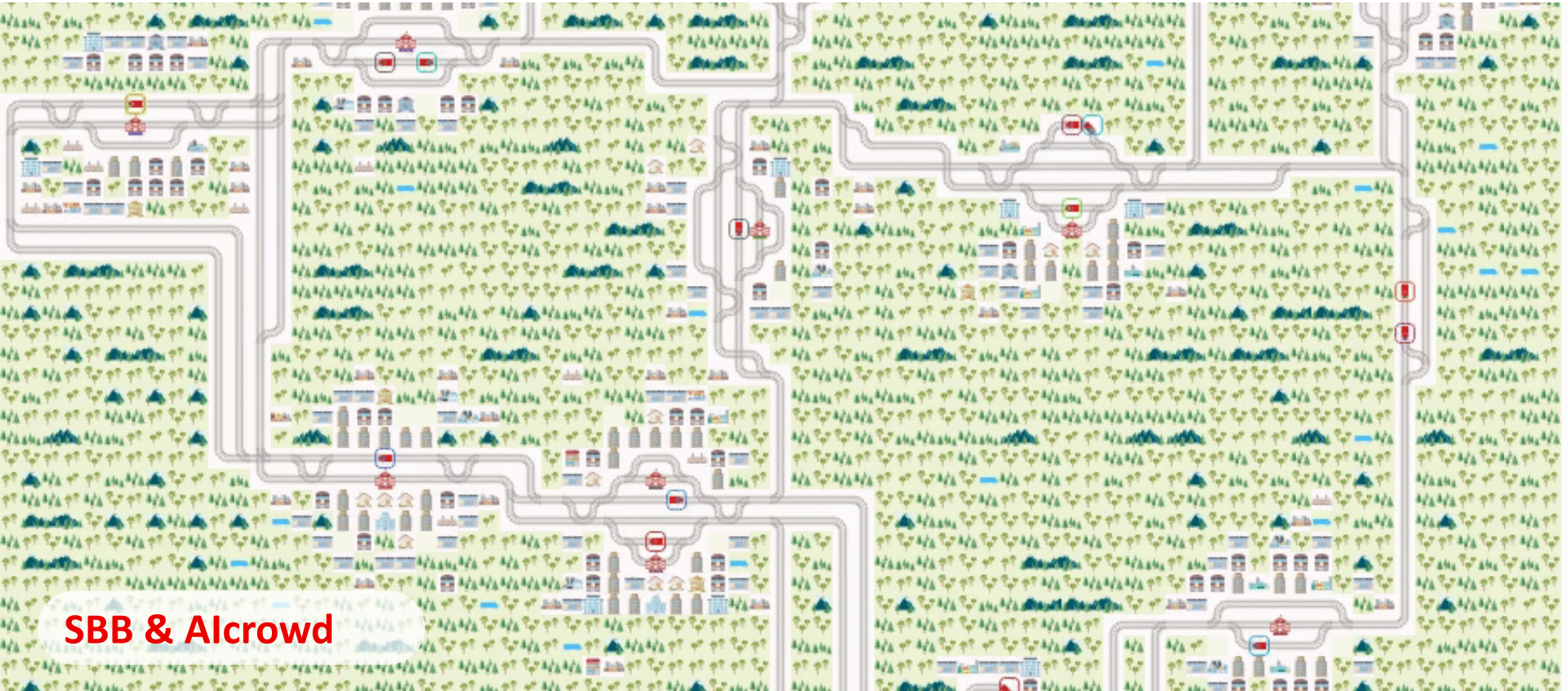
https://www.aicrowd.com/challenges/flatland-challenge

FLATLAND

Challenge:

SBB CFF FFS

What comes next?

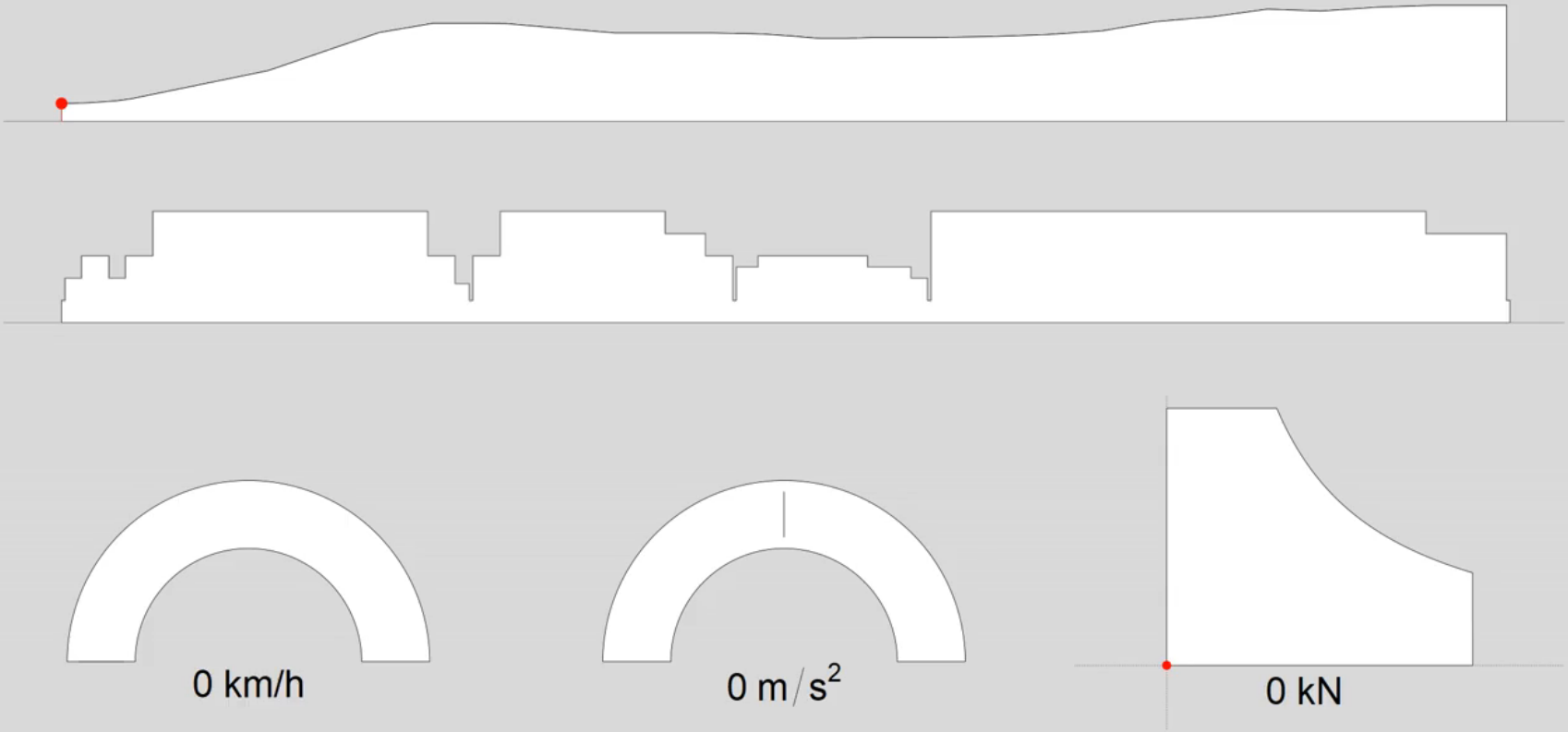


SBB & Alcrowd

Vision und Outlook

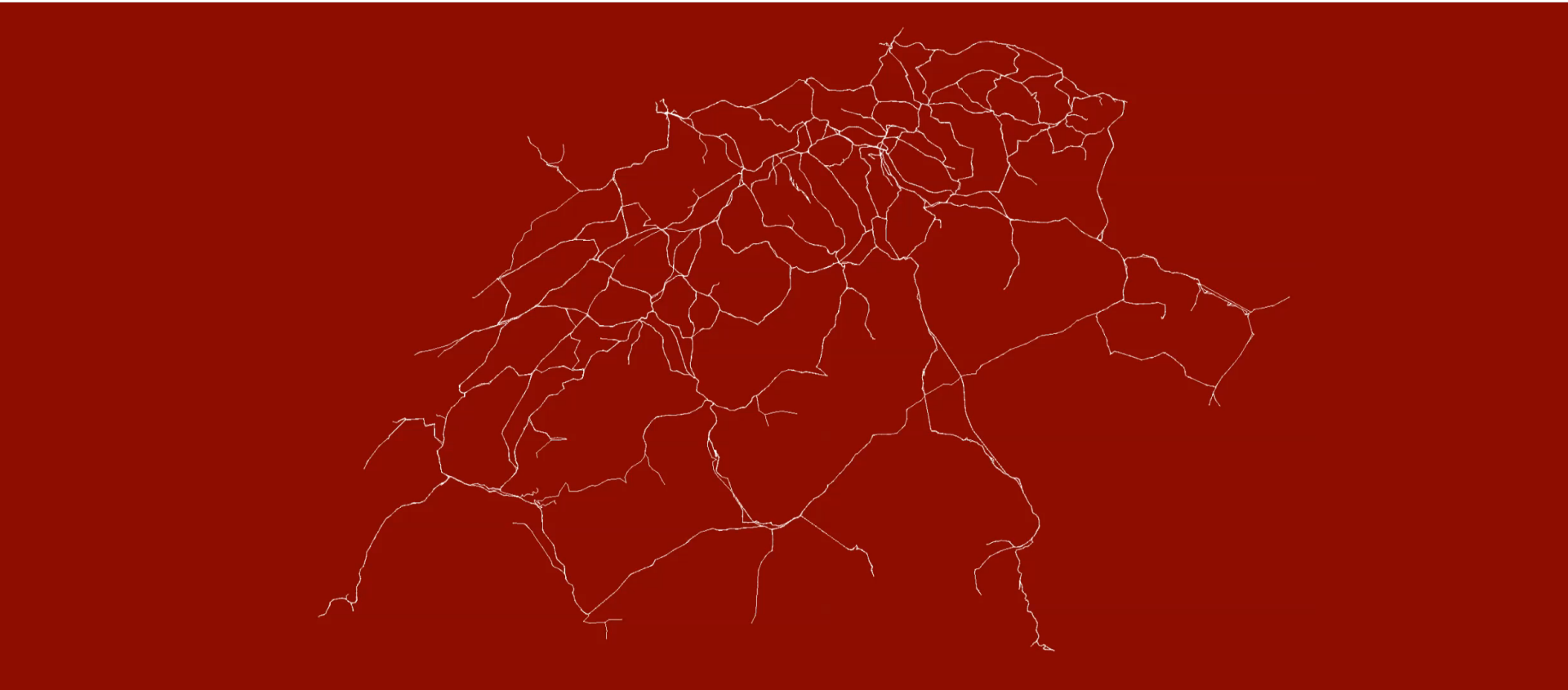
From Toy-Examples towards a Digital Twin.

Railway Simulation.
Physics of individual trains.



High Performance Railway Simulator

28’800 x Realtime



Digital Railway.

Predictions, Analysis and Experiments for the future of railway…