Neural Prioritized Planning: Flatland

Task 2.1 Beta release - UvA

AI4REALNET has received funding from European Union's Horizon Europe Research and Innovation programme under the Grant Agreement No 101119527

Context

Definition

Multi-Agent Path Finding (MAPF) involves finding collision-free paths for multiple agents in a shared environment.

Motivation

Efficiently scheduling agents in congested networks (e.g., railway systems) is crucial for minimizing delays and optimizing flow. While optimal solvers do exist, they **fail to scale** to problems with more than a few dozen agents. More scalable solvers are needed.

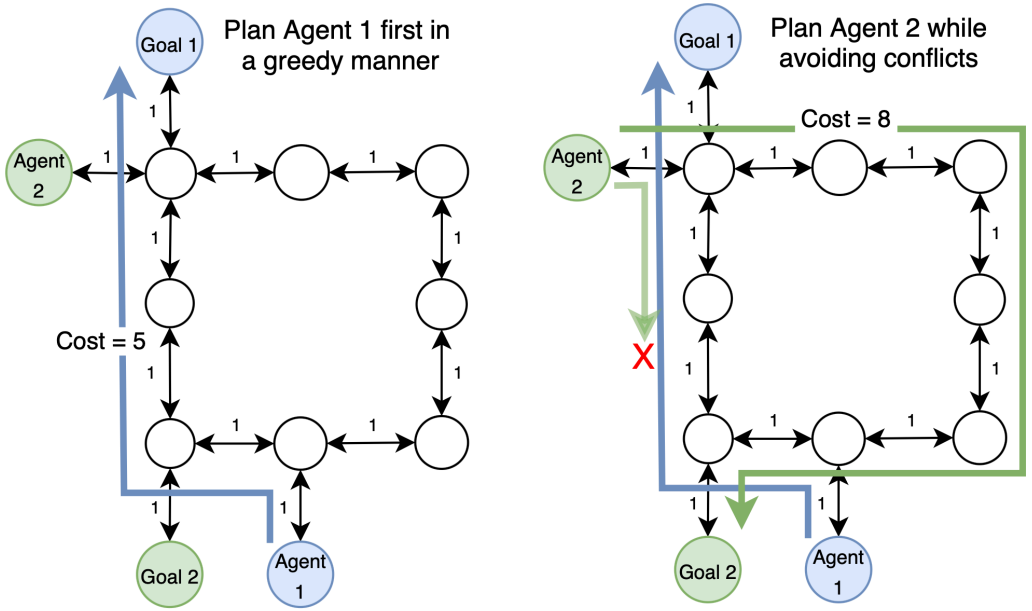
Use cases

Railway networks: Scheduling and routing trains. Ability to quickly replan on case of train breakdowns.

Motivation: small scale example

- Plan Agent 1 **greedily**, by using shortest distance to goal
- Agent 2 has to **avoid collision** with Agent 1

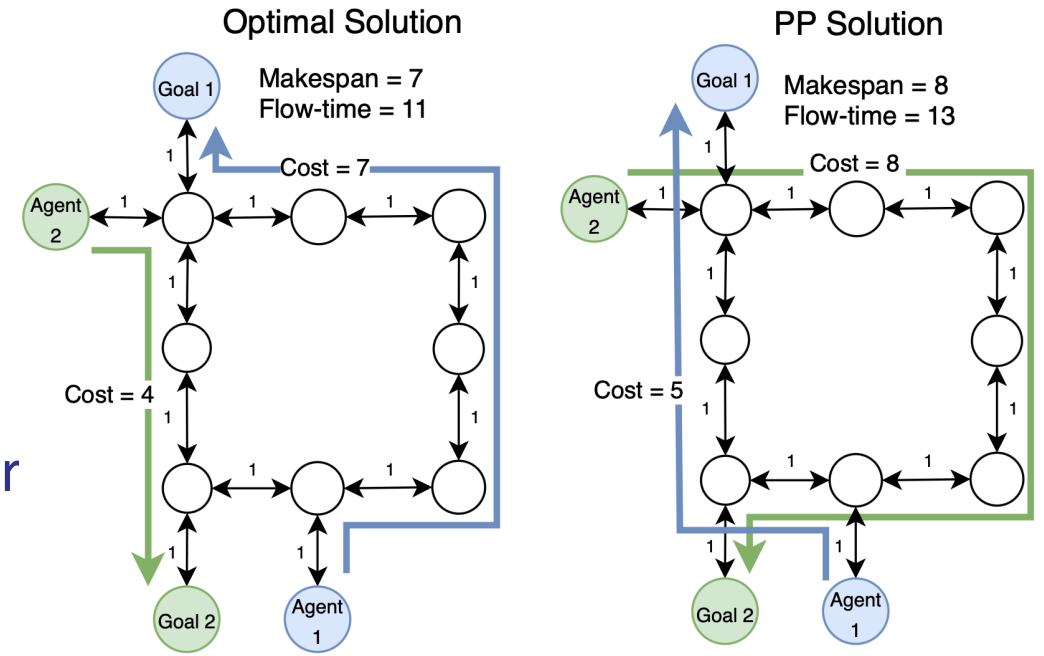
- **Prioritized Planning** (PP) is such an algorithm, that plans agents in sequence, based on the assigned priorities.
- Often times, PP it is **not** optimal.



Motivation: small scale example

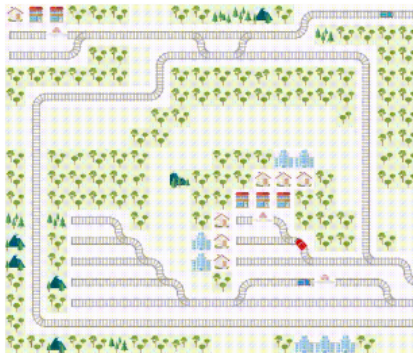
- In this example the optimal path is for Agent 1 to take the slightly longer route, while Agent 2 takes the much shorter one.
- **Makespan** = max length of all paths
- **Flow-time** = sum of length of all paths

- However, computing such optimal solutions for large scale environments is **not scalable**.



Methodology

- **Main idea** = learn **graph edge weights** representation such that Prioritized Planning (PP) finds solutions that are closer to optimal, while still planning in a greedy way.



Flatland input instance.
Graph weights set to **1** at the beginning.

Run PP w/ input graph weights

Proposed solution (PP)

Differentiable loss calculation between the 2 solutions

Expert (optimal) solution

Back-propagate loss to update the graph **edge weights**, making PP paths more similar to optimal.

Set **updated weights** for PP

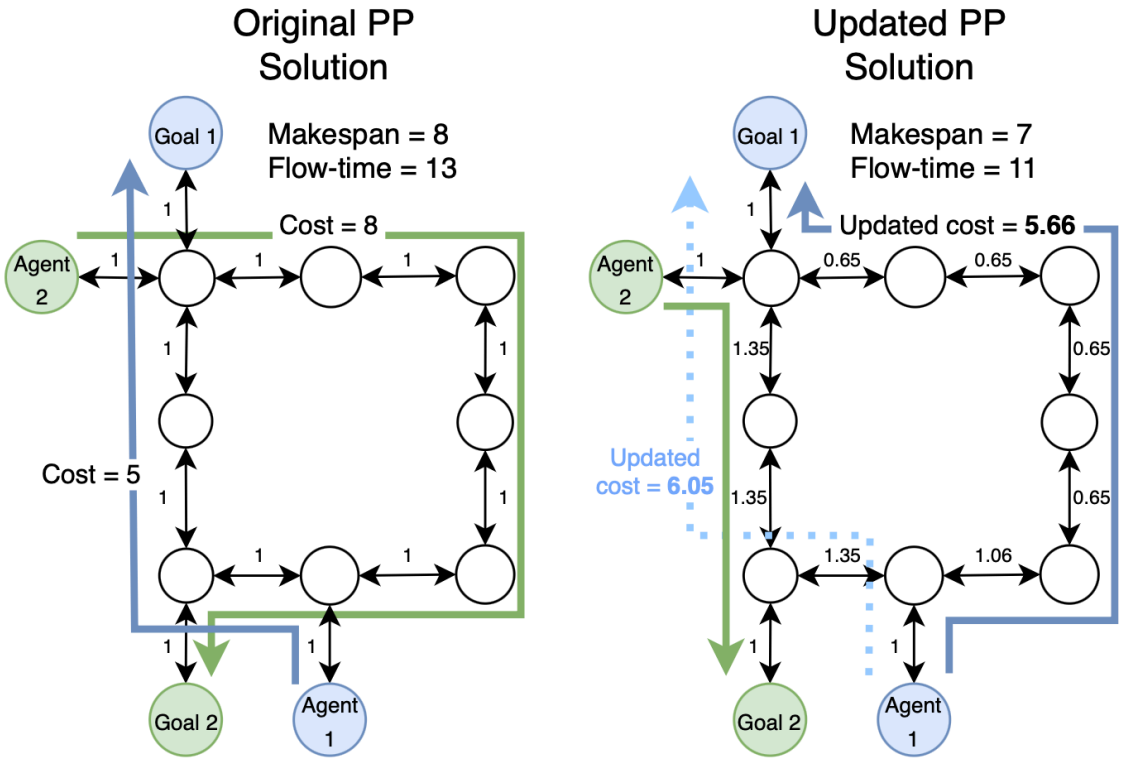
Use **Conflict Based Search** (CBS) to generate optimal paths.

Methodology

- **Conflict Based Search** (CBS) (Sharon, Guni, et al. 2015) is used to generate optimal paths.
- In order to get a meaningful gradient based on the computed loss, we use the method proposed in Vlastelica et al., 2019 which allows for the **Differentiation of Blackbox Combinatorial Solvers**.
- The main training loop is as follows:
    1. Generate optimal CBS paths.
    2. Compute initial PP paths on cost 1 graph.
    3. Calculate usage discrepancy using Hamming distance or similar metric.
    4. Update edge weights using the differentiable solver.
    5. Re-run PP with updated weights.
    6. Iterate until convergence or a fixed number of iterations.

Methodology: back to the small scale example

- Using the proposed methodology, we can learn to assign updated weights to edges in our first example.
- This way the **updated input graph** allows PP to find the optimal path, given this fixed set of priorities.
- PP still plans **greedily** in a fast way, since the input is the only change.



Preliminary experiments on Flatland

- On 30x30 Flatland maps, overall **mean flow-times** over 100 seeds for each configuration:

	Mean Flow-time		
Number of agents	CBS	PP	Trained PP
3	**59.75**	60.74	60.38
7	**140.86**	144.39	143.03
11	**224.33**	230.14	227.67

Preliminary experiments on Flatland

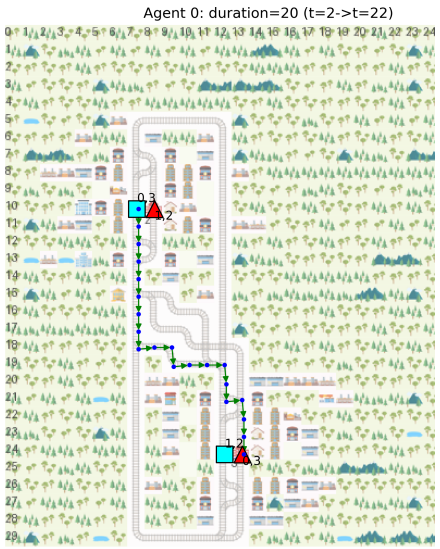
- Filtered mean flow-times (**only** cases when flow-time of **PP > CBS**):

	Mean Flow-time		
Number of agents	CBS	PP	Trained PP
3	**56.65**	60.83	59.30
7	**137.90**	145.42	142.52
11	**214.81**	224.76	220.52

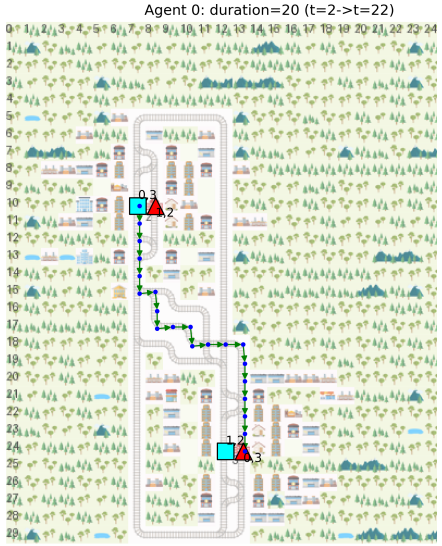
Example on 30x30 Flatland map with 4 agents

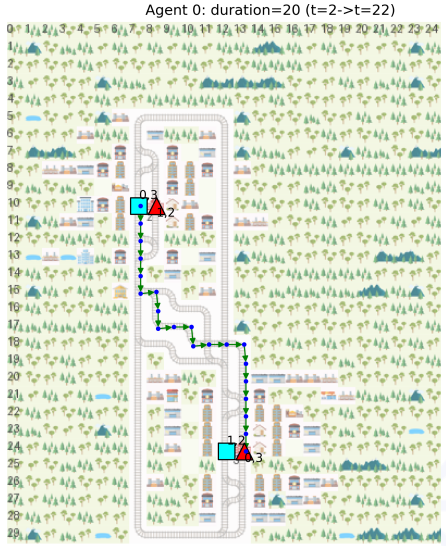
- End Stations are represented with a ▲, the numbers represents which agents has to finish there.
- Start Stations are represented with a ■, the numbers represents which agents start there.
- PP with the updated weights re-routes **Agent 0** through the middle rail, instead of the bottom one.
- **Agent 0 paths:**

Original PP → CBS → PP Trained (matches CBS Agent 0 path)



Vs

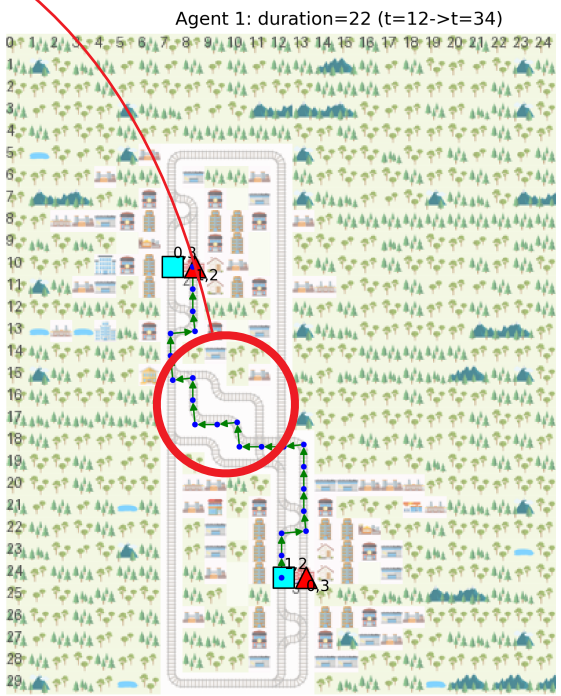




Preliminary experiments on Flatland

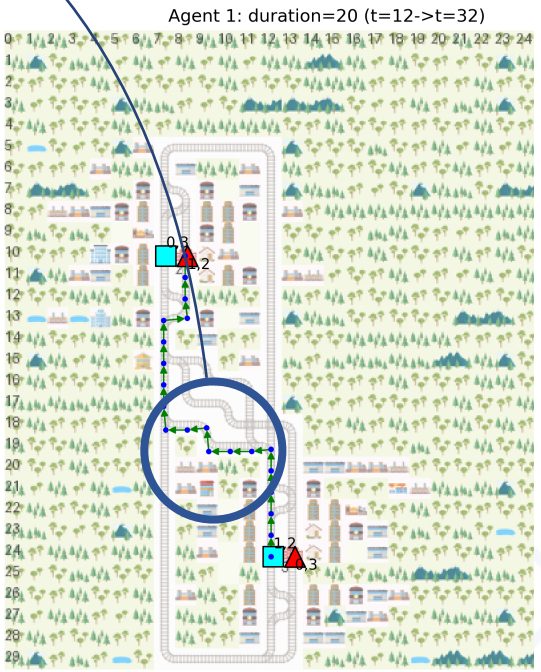
- So that Agent 1 can take the shorter route (**20 length**) using the **updated weights**, instead of the old PP route (**22 length**) to arrive at the goal, avoiding the conflict with Agent 0:

Original PP



Vs

PP Trained



ai4realnet.eu

Overview of code structure

- **`main.py`** is the entry point for running experiments
- CBS and PP implementation can be found under **`/solvers`**
- **`utils.py`** provides utility functions related to graph and instance processing
- **`/outputs`** stores outputs:
  - For single experiments: as **images** of the agent's paths overlaid over the flatland environment
  - For multiple experiments: **3 csv files** with path length statistics for each seeded run
- **`/tests`** includes all tests which can all be run using:
  - `python –m pytest`

```
.
├── README.md
├── flatland_blackbox
│   ├── __init__.py
│   ├── compute_results.py
│   ├── models.py
│   ├── run_experiments.py
│   ├── solvers
│   │   ├── cbs.py
│   │   └── pp.py
│   ├── train.py
│   └── utils.py
├── main.py
├── outputs
├── pyproject.toml
├── requirements.txt
├── run_configs.json
└── tests
    ├── conftest.py
    ├── test_solver.py
    ├── test_train.py
    └── test_utils.py
```

Installation and running experiments

- The installation instructions can be found on the `README.md`
- Run single experiments using command line arguments:
  - `python main.py –mode –solver [pp or cbs]`
- To run a single training instance :
  - `python main.py –mode train`
- To run a set of experiments across multiple instances, over a set amount of seeds:
  - `python main.py –mode experiments`
  - The experiments parameters are stored in **`run_configs.json`**
- **`python main.py –help`** to get a list of all possible commands

```
.
├── README.md
├── flatland_blackbox
│   ├── __init__.py
│   ├── compute_results.py
│   ├── models.py
│   ├── run_experiments.py
│   ├── solvers
│   │   ├── cbs.py
│   │   └── pp.py
│   ├── train.py
│   └── utils.py
├── main.py
├── outputs
├── pyproject.toml
├── requirements.txt
├── run_configs.json
└── tests
    ├── conftest.py
    ├── test_solver.py
    ├── test_train.py
    └── test_utils.py
```

Experiments input and output

- **Input**: Flatland instances, defined by the underlying graph structure and the collection of various agent's start and goal positions.
- For the experiments the specific input is given by the parameters found inside **`run_configs.json`** `on the right ->`
- **Output**: A **Plan**. A plan is a dictionary of **Agent_ID: Path** entries. A path is a list of tuples ((row, col), timestep), representing the positions of an agent at each distinct timestep.

{..} **run_configs.json** 229 B

```
{
  "iters": 300,
  "lr": 0.01,
  "lam": 3.0,
  "num_seeds": 100,
  "start_seed": 0,
  "num_agents_list": [
    3,
    7,
    11
  ],
  "max_cities_list": [
    2
  ],
  "width_list": [
    30
  ],
  "height_list": [
    30
  ]
}
```

Perspectives

- Next steps:
  - Working in Flatland simulation:
    - Learn based on **dynamic start/current positions** of agents.
      - Edge weights as a function of the current agent positions
    - If breakdowns happen, **replan** using updated positions/weights.
    - Learned weights should reflect possibility of **breakdowns**.
  - Learn to assign **priorities** (learn to rank approach).
- Future work:
  - Optimal solvers such as CBS dont scale into more realistic scenarios with tens/hundreds of agents at the same time.
  - Have an **Reinforcement Learning loss** instead of relying on expert optimal trajectories.

Authors

- Marius Captari (Universiteit van Amsterdam)
- Herke van Hoof (Universiteit van Amsterdam)

ai4realnet.eu

AI4REALNET

AI4REALNET has received funding from European Union's Horizon Europe Research and Innovation programme under the Grant Agreement No 101119527

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.