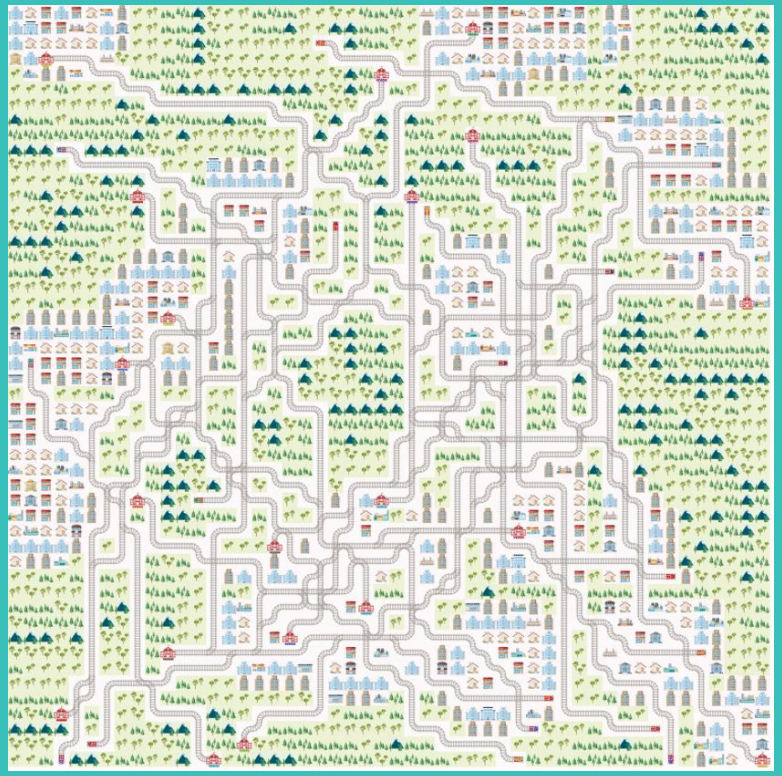
Railway routing (pt. 2)

One challenge, two different ways

Overview of the presentation

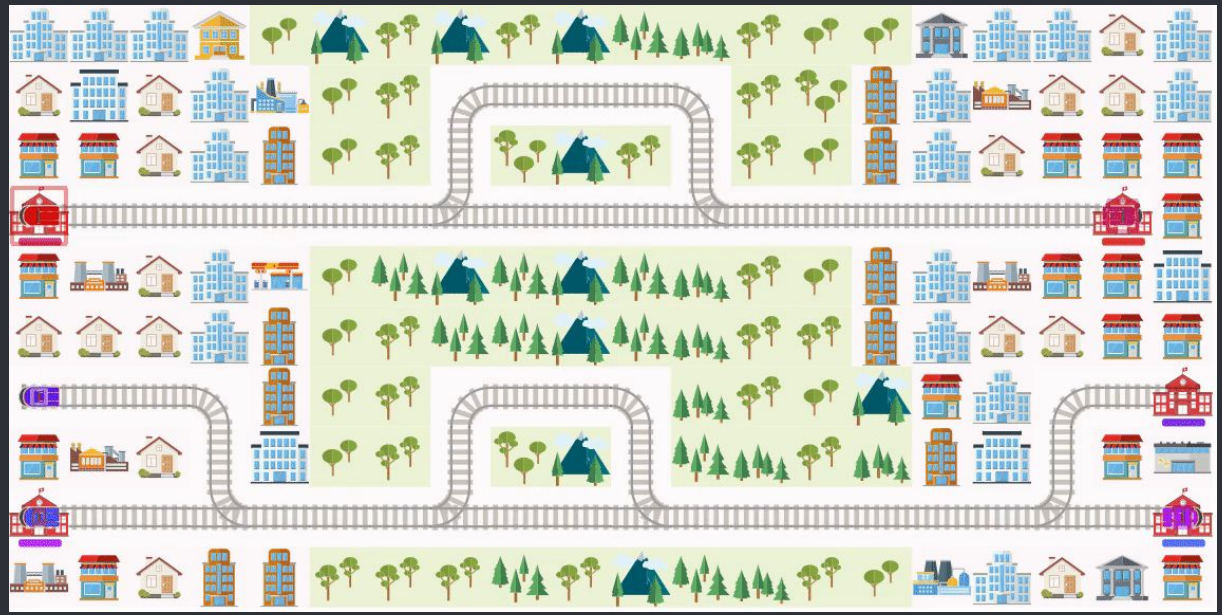
- Quick recap
- Reinforcement learning approach
- Combinatorial optimization approach

Flatland



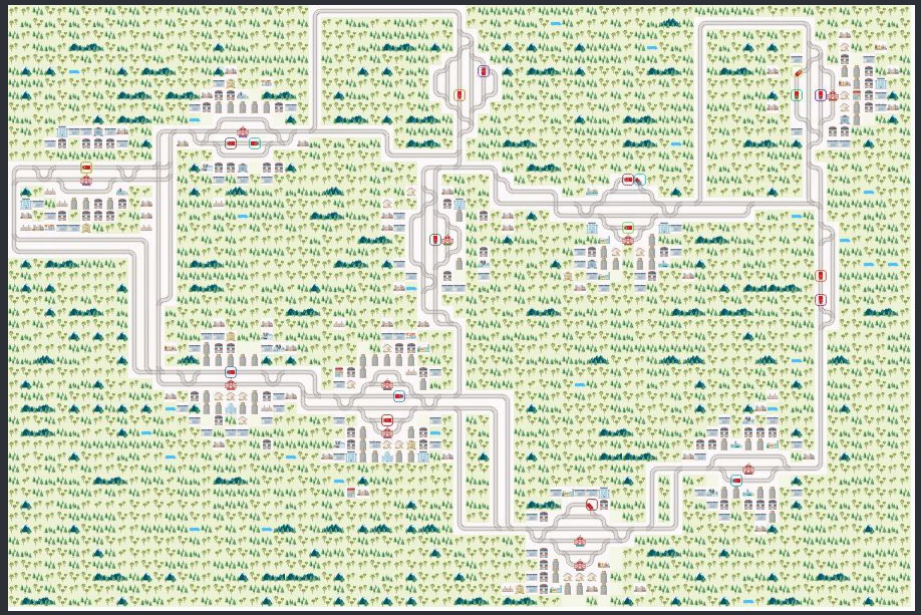
Online competition by SBB

“How can trains learn to automatically coordinate among themselves, so that there are minimal delays in large train networks ?”



Recall the environment

- Recall:
  - Each agent has its own target.
  - Collisions lead to deadlocks.
  - Trains have different speeds
  - Trains may malfunction for a variable amount of time-steps.



Reinforcement learning approach

What is Reinforcement Learning ?

Active Learning

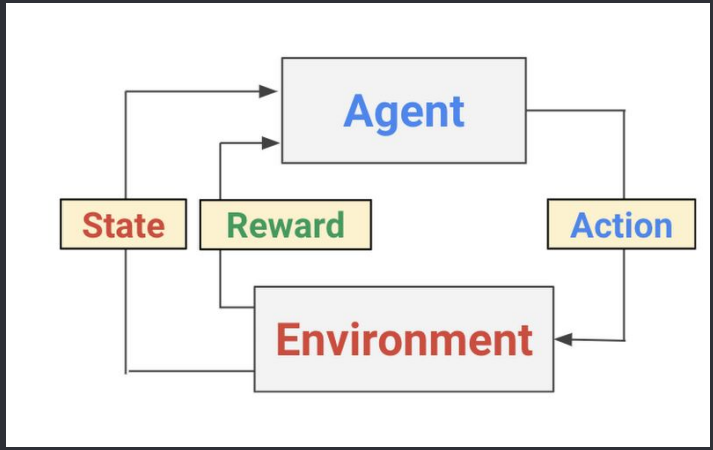
We learn by interacting with our environment.

Sequential Interactions

Future interactions can depend on earlier ones.

Independent Learning

We can learn without examples of optimal behaviour.



Markov Decision processes (MDP)

- A set of agent states : $\mathcal{S}$. → $(x, y,\ direction)$
- A set of actions $\mathcal{A}$ of the agent. → $\{stop,\ go\ forward,\ turn\}$

- The immediate reward after transition from state s to s' under action a:

$$R_a(s, s')$$

- The MDP and agent together give rise to a trajectory :

$$S_0, A_0, R_1, S_1, A_1, R_2 \ldots$$

Goals and Rewards

- The agent's job is to maximise its cumulative reward or *return*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots,$$
$$\gamma \in [0, 1]$$

- State-Value function:

$$V_\pi(s) = E_\pi[G_t | S_t = s]$$

- Action-Value function:

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

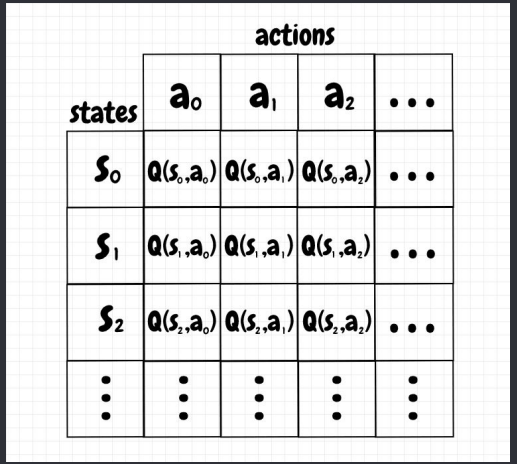
Previous approach

Q-Learning Algorithm : Learning the Q-table, using an *ε-greedy policy*.

- Initialize the Q-Table with 0.
  - Use an *ε-greedy policy* :

    With probability ε, pick an action at random, otherwise take $\arg\max_a Q(s,a)$.

    → *Exploration vs Exploitation trade-off*

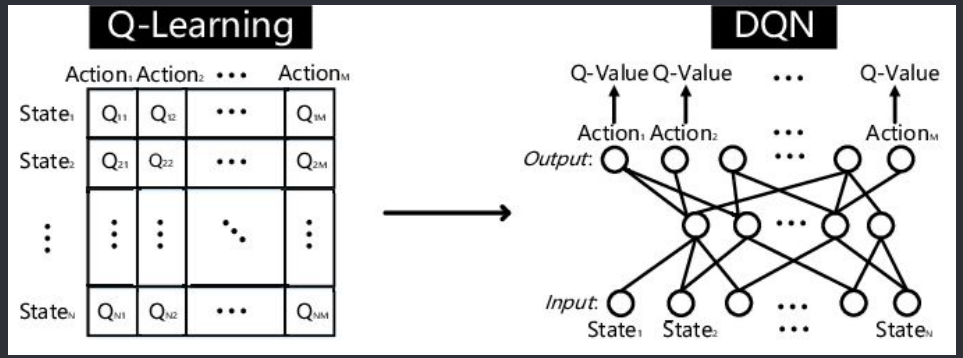


- For each step taken, we update the table with the formula:

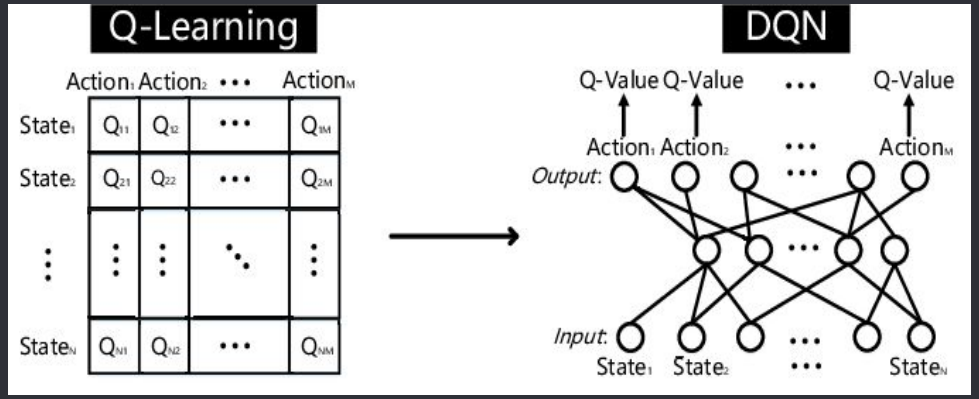
$$Q^{new}(s_t,a_t) \leftarrow (1-\alpha)\cdot \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Deep RL vs ‘tabular’ RL

- What if the states become too complex for the this approach ?
  - This is where a neural network can assist through function approximation.



Deep RL advantages



- **<u>Change of paradigm</u>**:
  - We do not store every state-action combination anymore.
  - Instead we approximate the value of a state-action combination with states as inputs.

→ Not only does function approximation eliminates the need to store everything, it can also allow the agent to ***generalize*** to the value of states it has never seen before.

High level idea of Deep Q-networks:

- Make Q-learning look like supervised learning:
  - We want the inputs (states) to be **independent and identically distributed.**
    - If not, the model may overfit for some class of inputs.
  - In addition, for a given input, we would like the target value to not change over time.

→ However, in Reinforcement Learning, both the input and the target change constantly during the learning process, making the training ***unstable.***

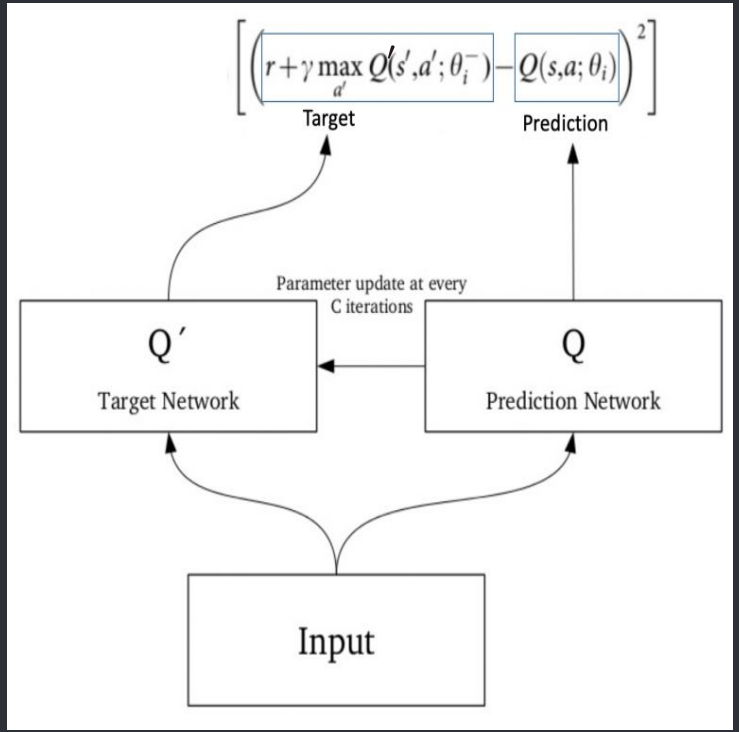
Double Q-network (DQN)

2. Feed the state s to our DQN, which returns the Q-values of all the possible actions
3. With probability ε, pick an action at random, otherwise take $\arg\max_a Q(s,a)$.

4. Perform this action a, observe new state s' and reward r and store this transition in a replay buffer as (s, a, r, s').
5. Sample random mini-batch from the replay buffer and calculate the loss:

$$Loss = (r + \gamma \cdot \max_{a'} Q'(s', a'; \theta') - Q(s, a, ; \theta))$$

6. Perform gradient descent in the Prediction Network to minimize the loss.
7. After C iterations, copy the weights of the Prediction Network to the Target Network



Deep Q-Networks adjustments.

Unstable target:

→ We build a deep network to learn the Q-values but the target values keep changing as we keep refining them when *exploring*.

<u>Solution</u>:

Use two deep networks with the same architecture but different weights.

→ A target network with weights θ'. $Target = r + \gamma \cdot \max_{a'} Q'(s', a'; \theta')$

→ A prediction network with weights θ. $Estimate = Q(s, a, ; \theta)$

→ After a many of updates, *replace* the weights θ' with θ.

<u>Idea</u>: prevent instabilities to propagate quickly and reduce the risk of divergence by having fixed target values for many iterations.

Deep Q-Networks adjustments.

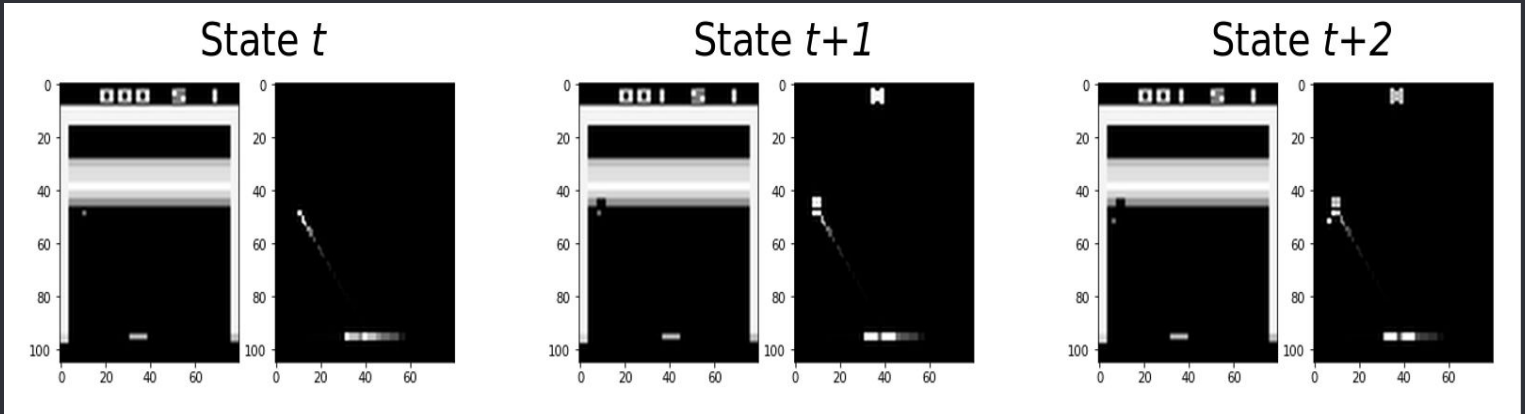
Trajectory correlations:

Say we just scored a good reward and we make a new action. Then the new state may look similar to the previous one and our new target for Q will be higher regardless of the merit of the new action.

When we pull up a Q-value, the Q-values in the surrounding state space will also get pulled-up like lifting a net.

Deep Q-Networks adjustments.

Trajectory correlations - example:



Deep Q-Networks Potential Problems

Trajectory correlations:

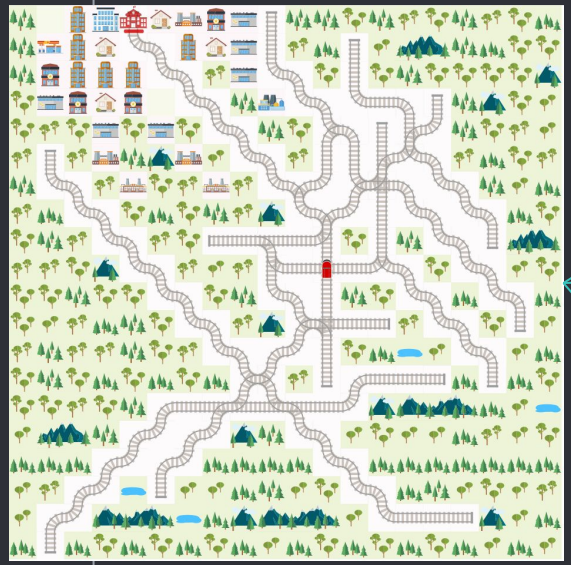
Solution:

→ Store experience tuples (s, a, r, s') in a *replay buffer*, and randomly sample mini-batches from this buffer when training.

Idea: To decorrelate the data: As we randomly sample from the *replay buffer*, the data is closer to being **i.i.d**.

1 Onto the flatland problem.

State description



```
[ 0, 10,  9,  0,  0,  8,  9,  8,  9, 10,  0]
[ 0,  0,  8,  7,  0,  7, 10,  7,  8,  0,  0]
[ 0,  0,  0,  6,  5,  6,  5,  6,  7,  0,  0]
[ 0,  0,  0,  0,  4,  3,  4,  0,  8,  9,  0]
[ 0,  0,  0,  0,  0,  2,  0,  0,  0, 10, 11]
[ 0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
```

```
[ 0, 10,  9,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  8,  7,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  6,  5,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  4,  3,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  2,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0]
```

\+ Node informations:

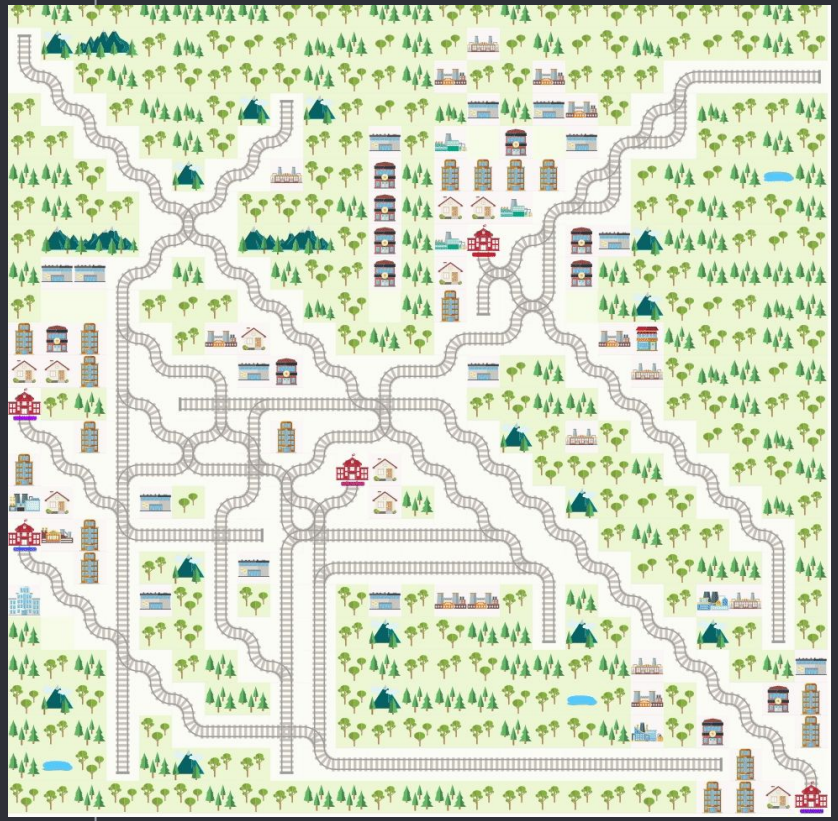
-distance to target

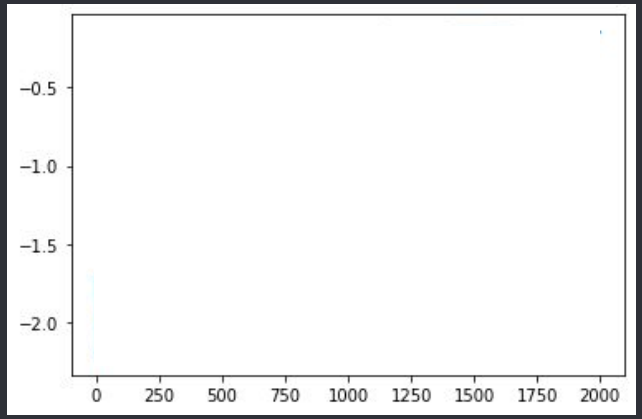
-breakdown informations

-information about other agents . . . .

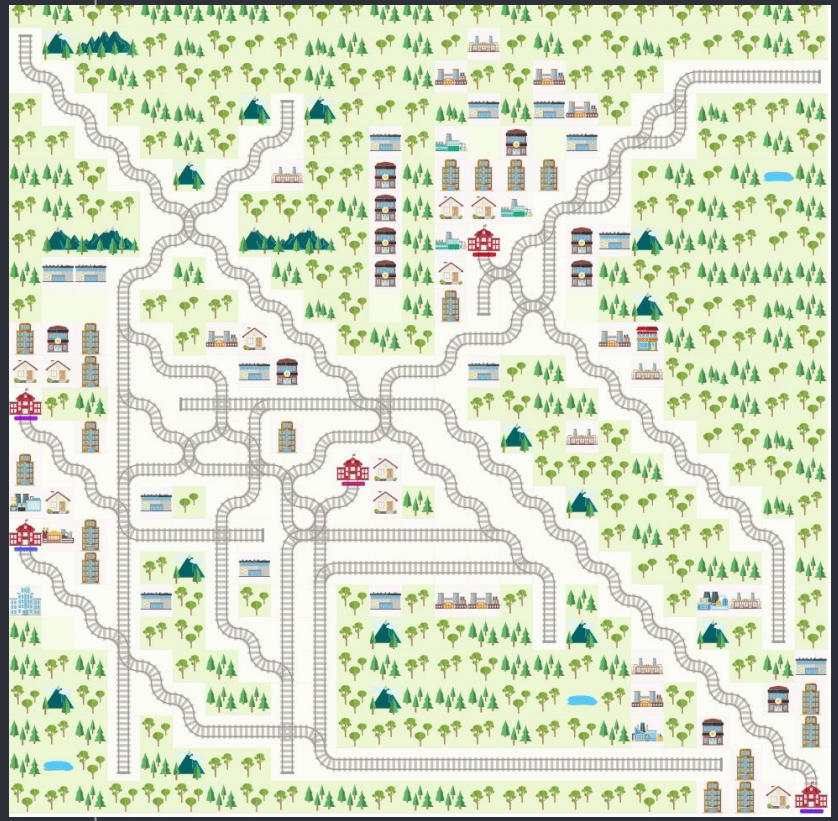
- . . .

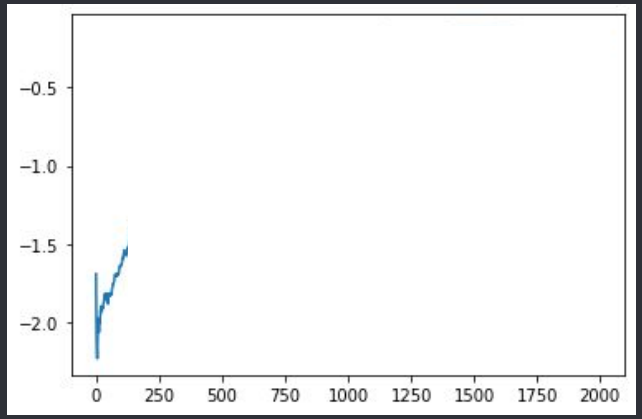
Episode 1



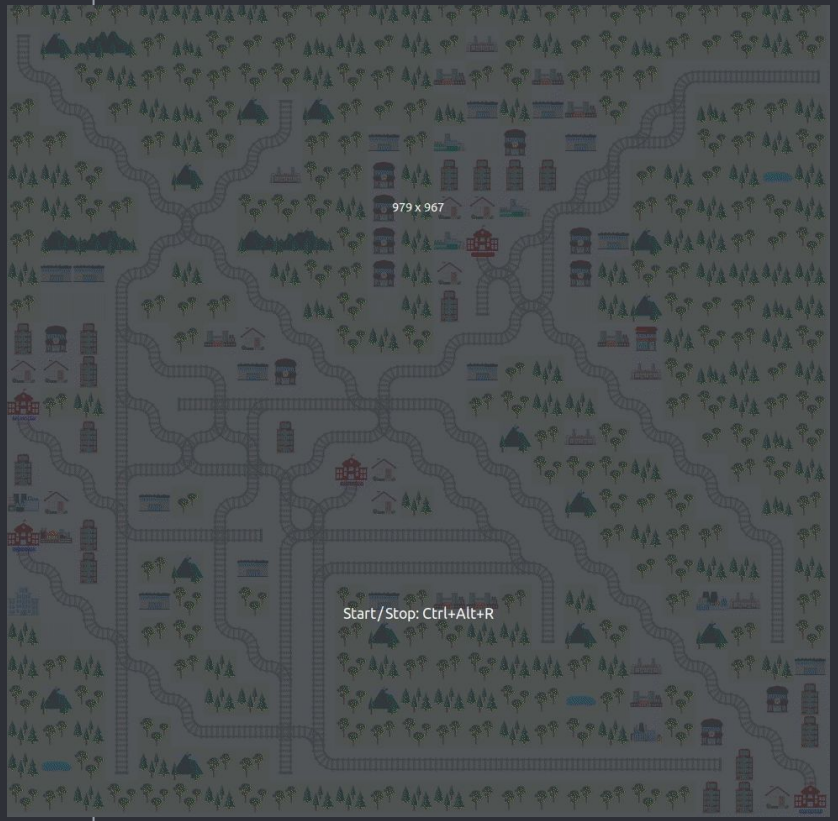


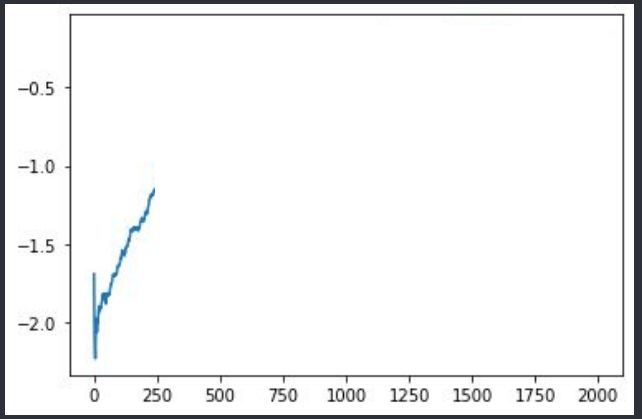
Episode 100



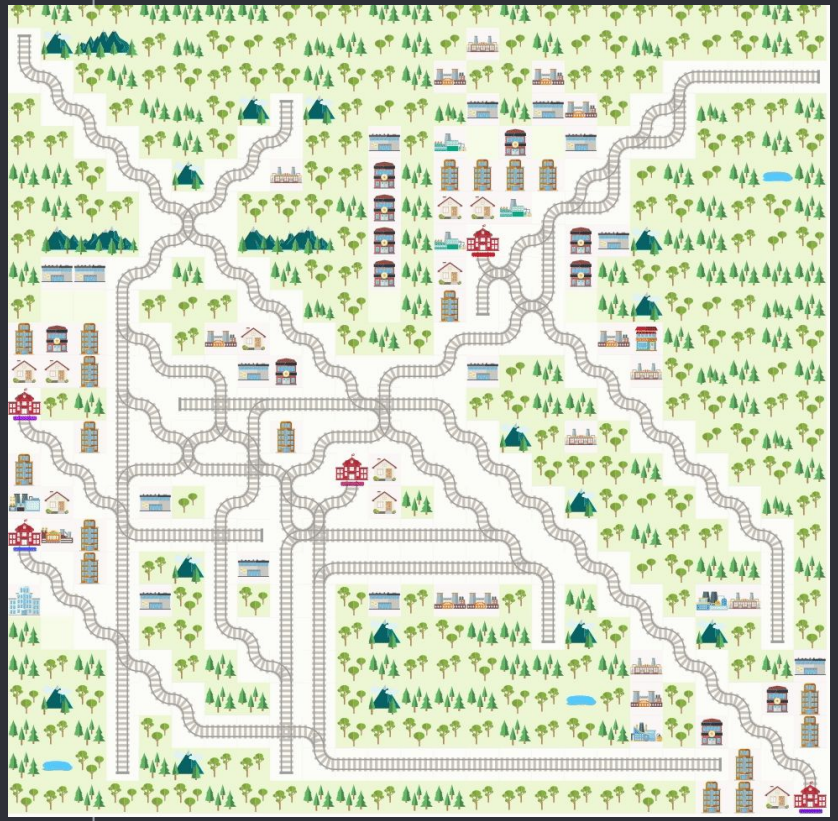


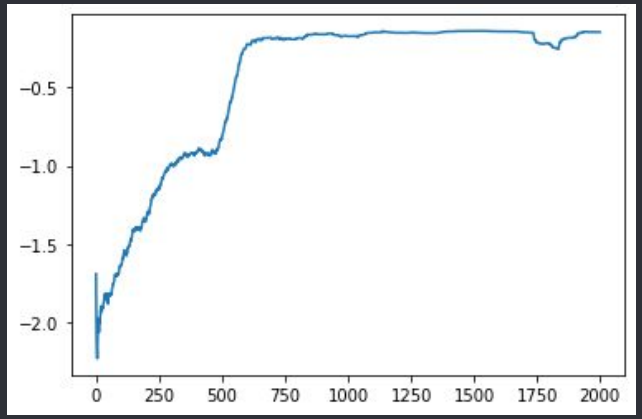
Episode 250


979 x 967
Start/Stop: Ctrl+Alt+R


−0.5
−1.0
−1.5
−2.0
0
250
500
750
1000
1250
1500
1750
2000

Episode 500



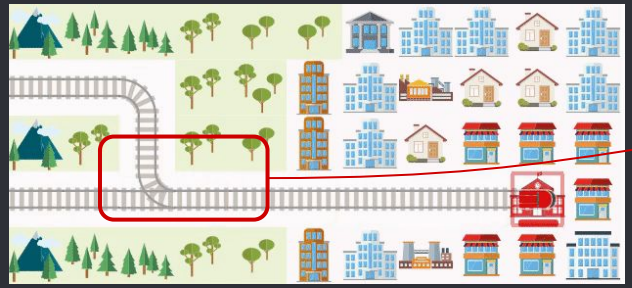


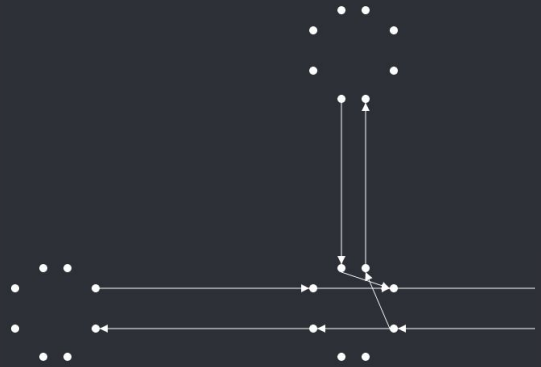
Combinatorial optimization approach

1

What was done before the first presentation

Previous method





$G = (V, A)$ directed graph

$$\text{minimize} \sum_{k=1}^{K} \sum_{(i,j)\in A} x_{ij}^k$$

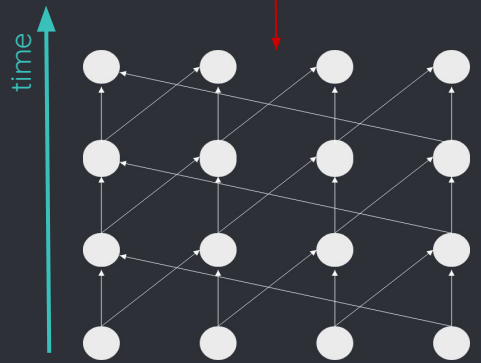
$$\text{subject to} \sum_{k=1}^{K} x_{ij}^k \leq 1, \qquad \forall (i,j) \in A$$

$$\sum_{k=1}^{K} \sum_{(i,j)\in S} x_{ij}^k \leq 1, \qquad \forall S$$

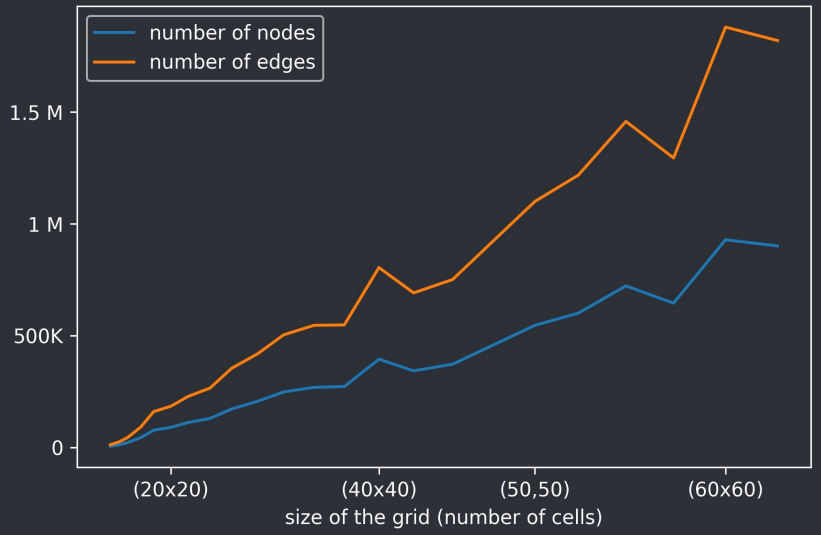
$$\sum_{k=1}^{K} \sum_{(i,j)\in C} x_{ij}^k \leq 1, \qquad \forall C$$

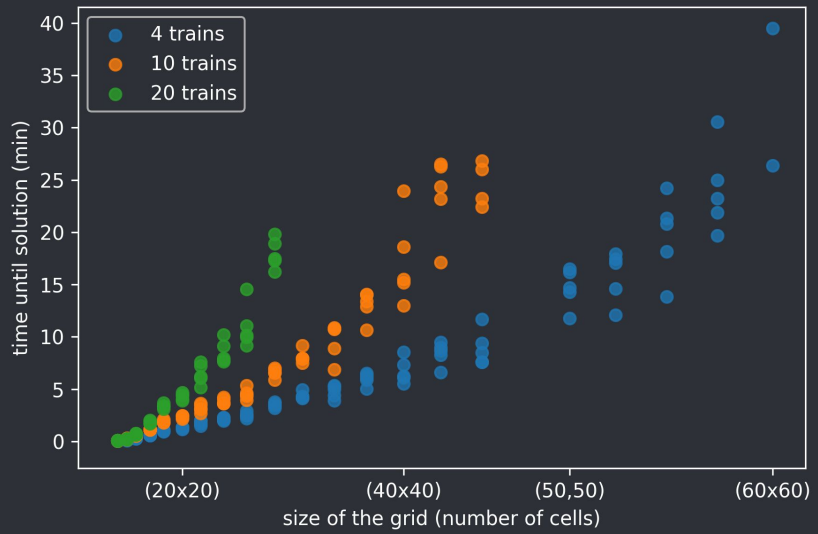
$$\mathcal{N} x^k = b^k \qquad k \in [K]$$

$$x_{ij}^k \in \{0,1\}, \qquad \forall (i,j) \in A, k \in [K]$$



What was the issue ?

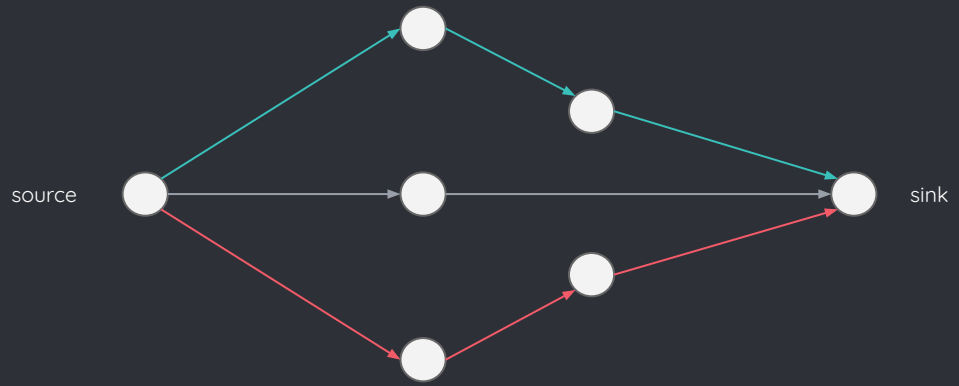




Huge number of variables → explosion of memory and computation time !
But most of the variables are not used in solution

2 What's new?

Path formulation



Variables decide the flow to push on each path from source to sink

Path formulation: master LP

$$
\begin{aligned}
\text{minimize} \sum_{P \in \mathcal{P}} \quad & f(P)|P| \\
\text{subject to} \sum_{P \in \mathcal{P}_R} \quad & f(P) \leq 1, \quad \forall R \in C_R \\
\sum_{P \in P^k} \quad & f(P) = 1, \quad \forall k \in \{1, \ldots, K\} \\
& f(P) \geq 0, \quad P \in \mathcal{P}
\end{aligned}
$$

$\mathcal{P} = \{P^1, \ldots, P^k\}$ P^i : All possible paths for commodity i

C_R : Restriction imposed by the environment

List of paths from source to sink → exponential number of variables!

Column generation method

Main Idea:

Add only useful paths to the LP

Procedure:

1. Start with a feasible solution (set of paths)
2. Solve the restricted LP
3. Find variables unused that could improve the solution
4. Add these variables to the basis and GOTO 2.

Column generation method for multicommodity flows

Finding useful variables

- Similar to simplex method: consider the reduced cost (≥0)
- → minimum weighted path problem

Initial solution

- Any feasible solution

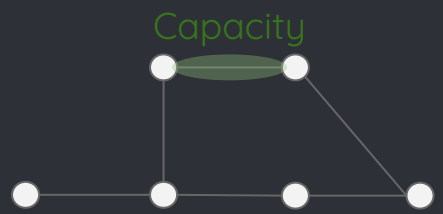
Main issues:

→ Our constraints are different from what is usually found in the litterature

→ Finding a feasible solution is not trivial

Constraints differences

Classic



Ours

Capacity restrictions only
→ one constraint per arc

Swapping, capacity and position restrictions
→ arc will belong to multiple constraints

Dual variables don’t relate directly to edges!

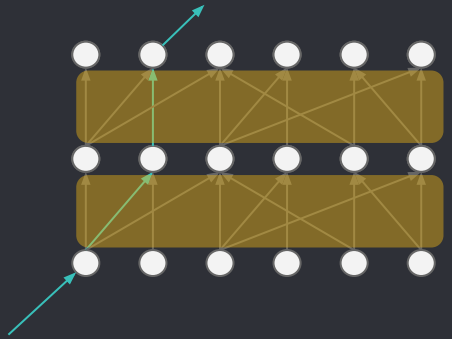
Reduced cost differences

Classic

$$c_P^{\sigma,w} = c^k(P) + \sum_{(i,j)\in P} w_{ij} - \sigma^k$$

Ours

$$c_P^{\sigma,w} = |P| - \sum_{R\in C_R} \delta_P(R)\cdot y_R - \sigma^k$$



Cost of path for commodity k

Weighted path problem

$$\tilde{w}_{ij} = -\sum_{R\in C_R} \delta_R((i,j))\cdot y_R \Rightarrow \sum_{(i,j)\in P} \tilde{w}_{ij} = -\sum_{R\in C_R} \delta_P(R)\cdot y_R$$

Column generation method in our case

Finding useful variables

- Problem structure → relate dual variables to edges

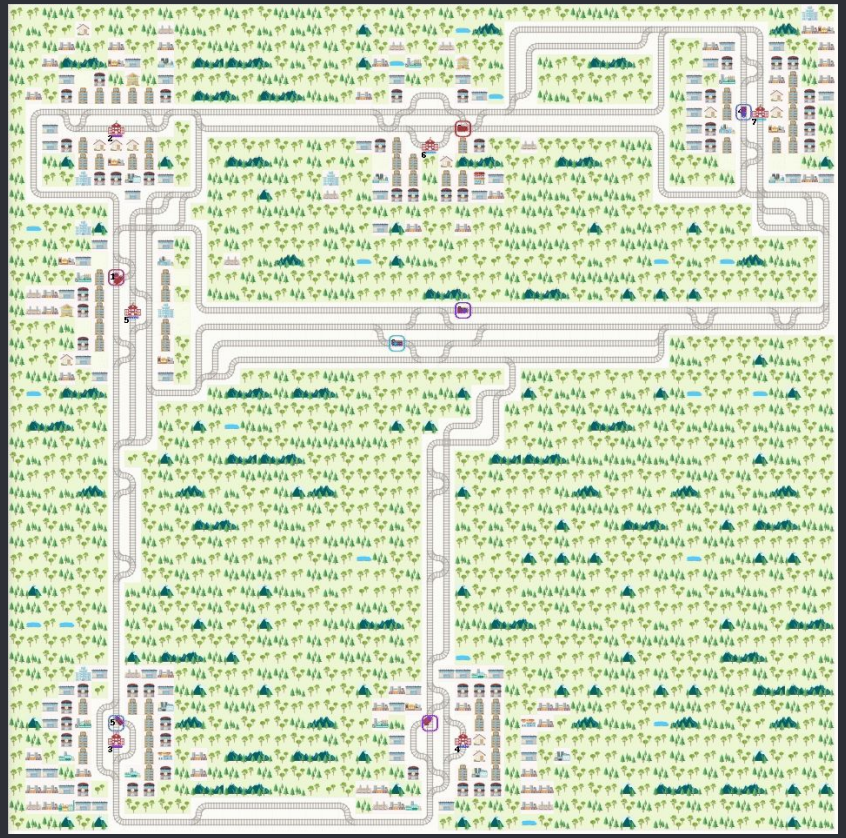
$$w_{ij} = -\sum_{R \in C_R} \delta_R((i,j)) \cdot y_R$$

- → minimum weighted path problem

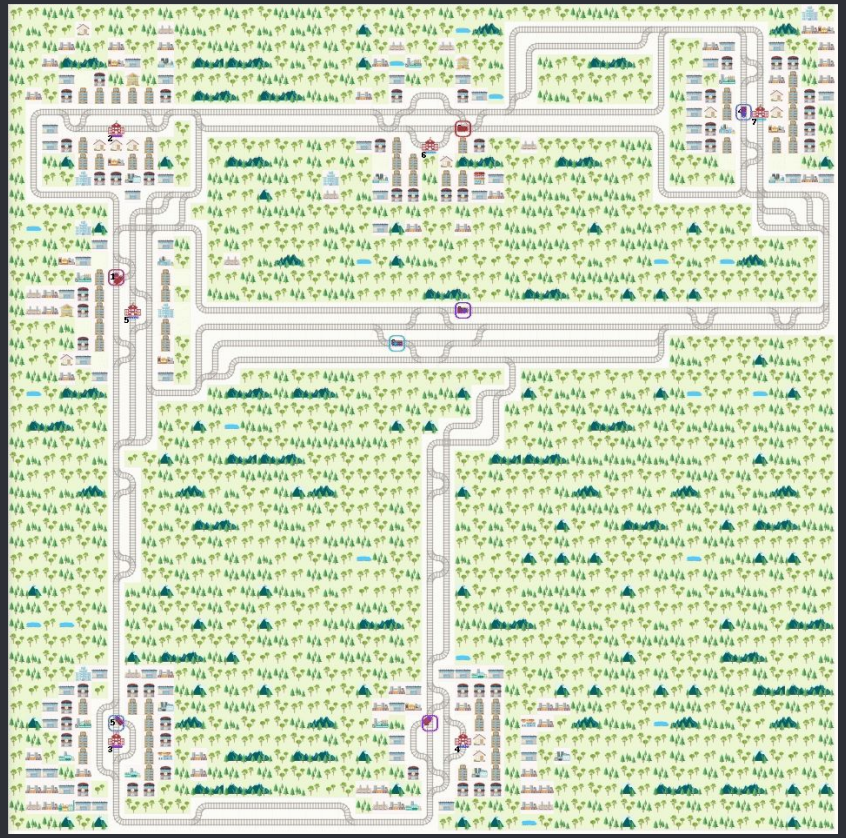
Initial solution

- Greedy sequential approach

Demo

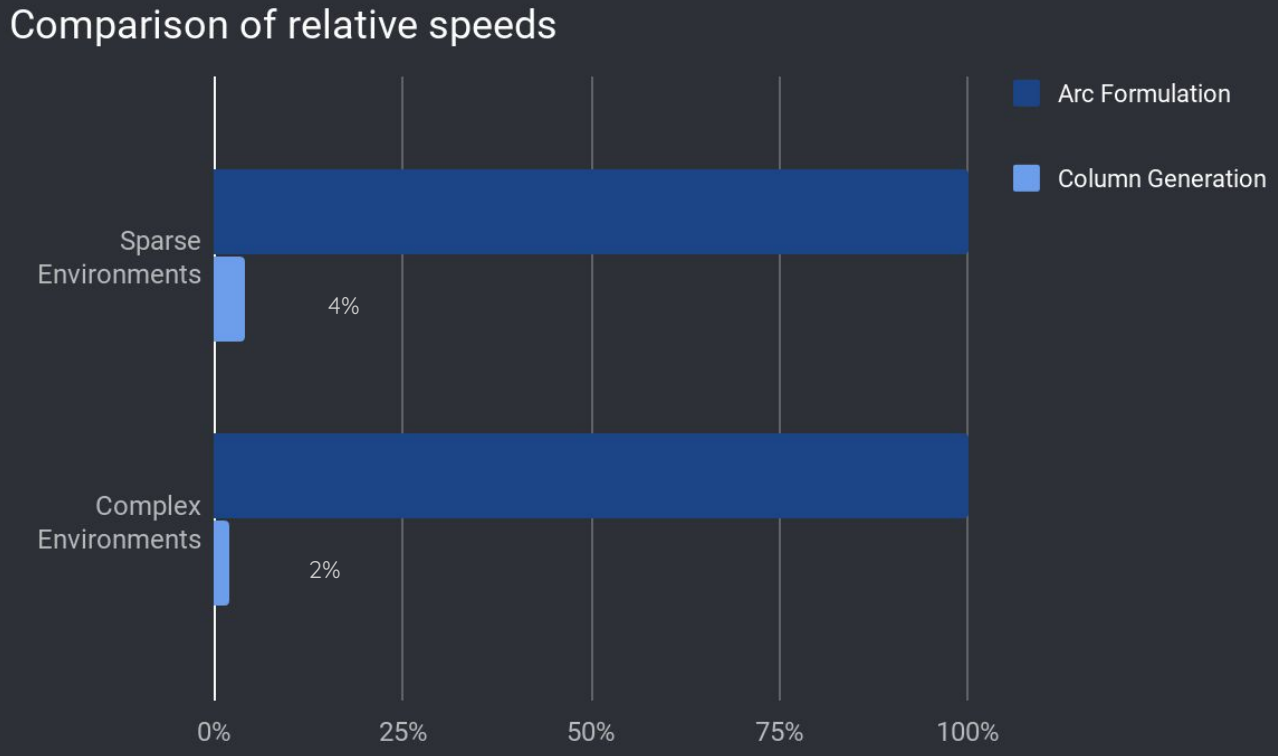


Demo

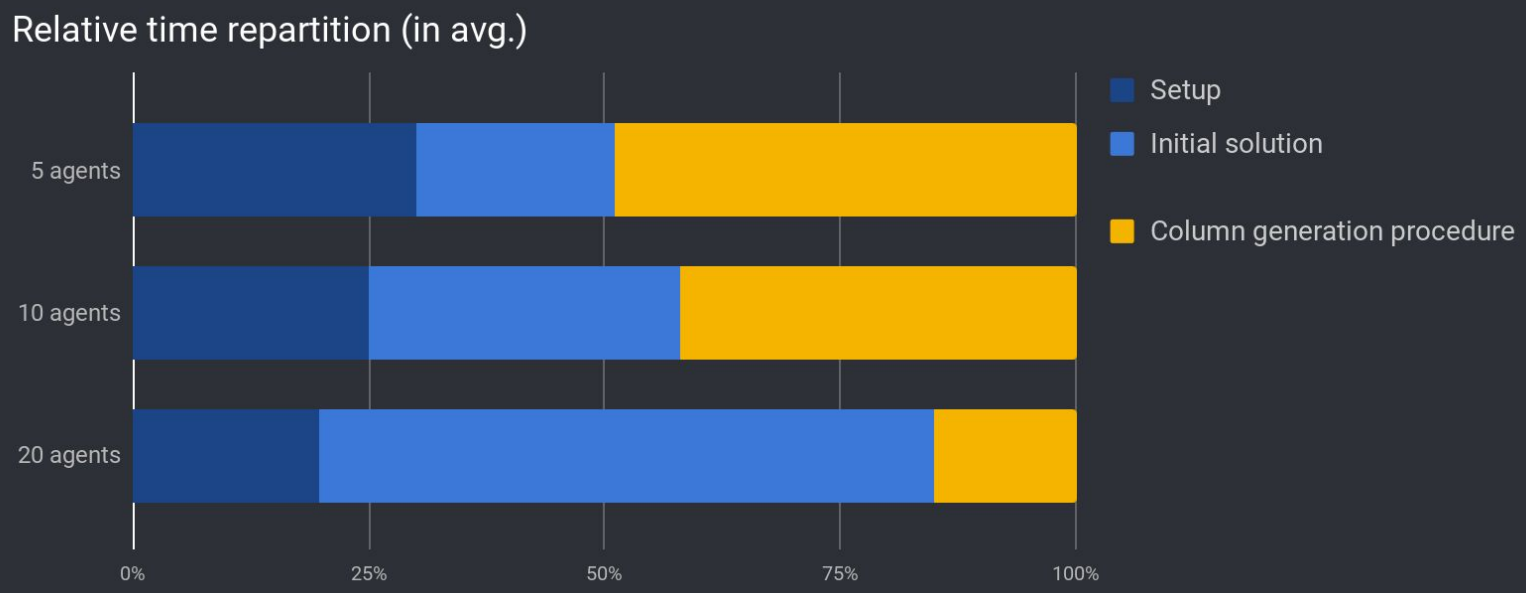


3 Performance analytics

Comparison with previous solution



More analytics for the column generation method



3 Shortcomings & possible improvements

Handling of different cases

Different speeds

- Build one network per speed
- Constraints spanned across networks
- Columns generation method

Network failures

- Delete unavailable cells in time expanded network
- Restart column generation method (either from scratch or only for the broken paths)

What could be next ?

Merging the two approaches (pt. 1)

Combinatorial Optimization → labelled examples

Supervised Learning

Reinforcement Learning (unsupervised)

Merging the two approaches (pt. 2)

Combinatorial Optimization → details

Reinforcement Learning → “landmarks”

Thanks!
ANY QUESTIONS?

Technical annexe

Annexe RL

Return:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} \dots$$

Discounted Return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots$$
$$\gamma \in [0, 1]$$

State Value function:

$$V_\pi(s) = E_\pi[G_t | S_t = s] = E_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots | S_t = s],$$
$$\gamma \in [0, 1], \forall s \in \mathcal{S}$$

Action-State Value function

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

Optimal policies and value functions

For two policies, we have the partial ordering: $\pi' \geq \pi \iff v_{\pi'}(s) \geq v_{\pi}(s), \forall s \in \mathcal{S}$

For finite MDPS, with discrete actions space, discrete state space and bounded rewards, there is always at least one policy *better than or equal* to any other policy. We call these policies *optimal policies.* We denote all the optimal policies by π_*.

These share the same state-value functions as well as action-state value functions, denoted and defined by:

$$v_*(s) = \max_{\pi} v_{\pi}(s), \quad \forall s \in \mathcal{S}$$
$$q_*(s,a) = \max_{\pi} q_{\pi}(s,a), \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$$

With the relationship:

$$q_*(s,a) = E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

Annexe CO

Arc formulation details

$G = (V, A)$ directed graph

$$\text{minimize} \sum_{k=1}^{K} \sum_{(i,j) \in A} x_{ij}^k$$

Sum of time steps for all agents

$$\text{subject to} \sum_{k=1}^{K} x_{ij}^k \leq 1, \qquad \forall (i,j) \in A$$

Capacity constraints

$$\sum_{k=1}^{K} \sum_{(i,j) \in S} x_{ij}^k \leq 1, \qquad \forall S$$

$$\sum_{k=1}^{K} \sum_{(i,j) \in C} x_{ij}^k \leq 1, \qquad \forall C$$

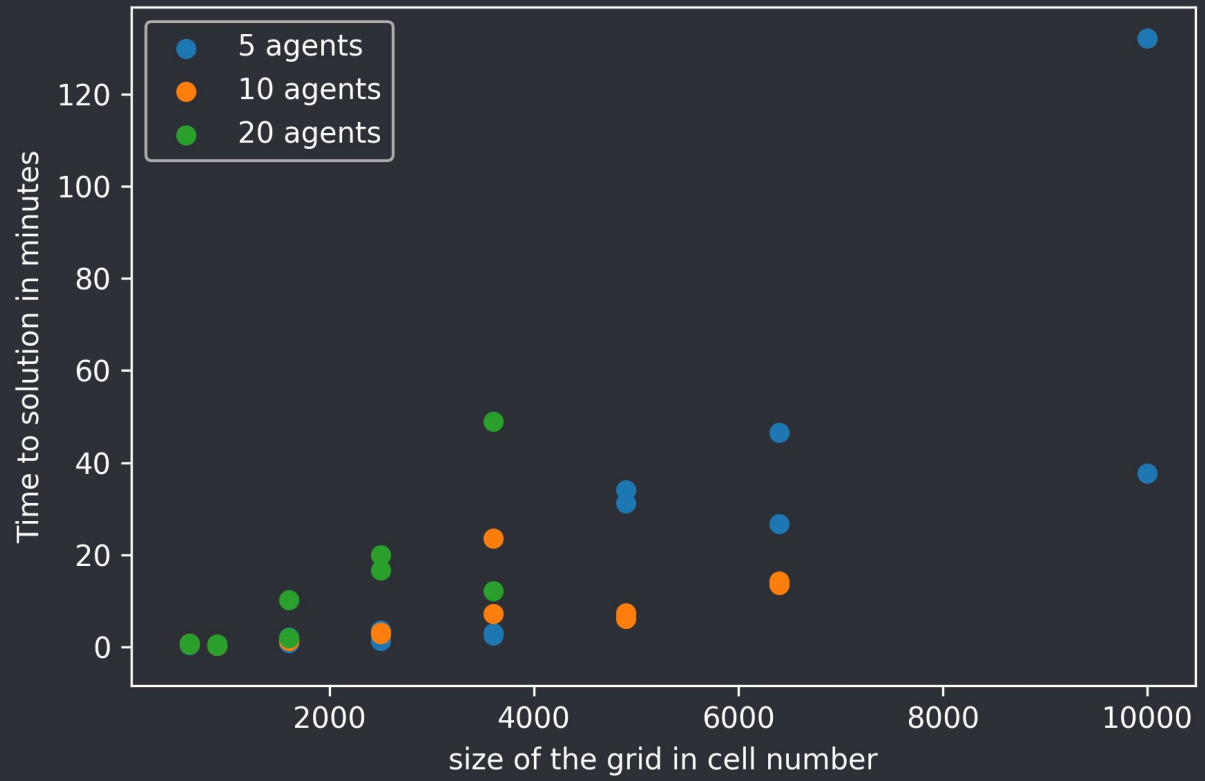
“Position” constraints

$$\mathcal{N} x^k = b^k \qquad k \in [K]$$

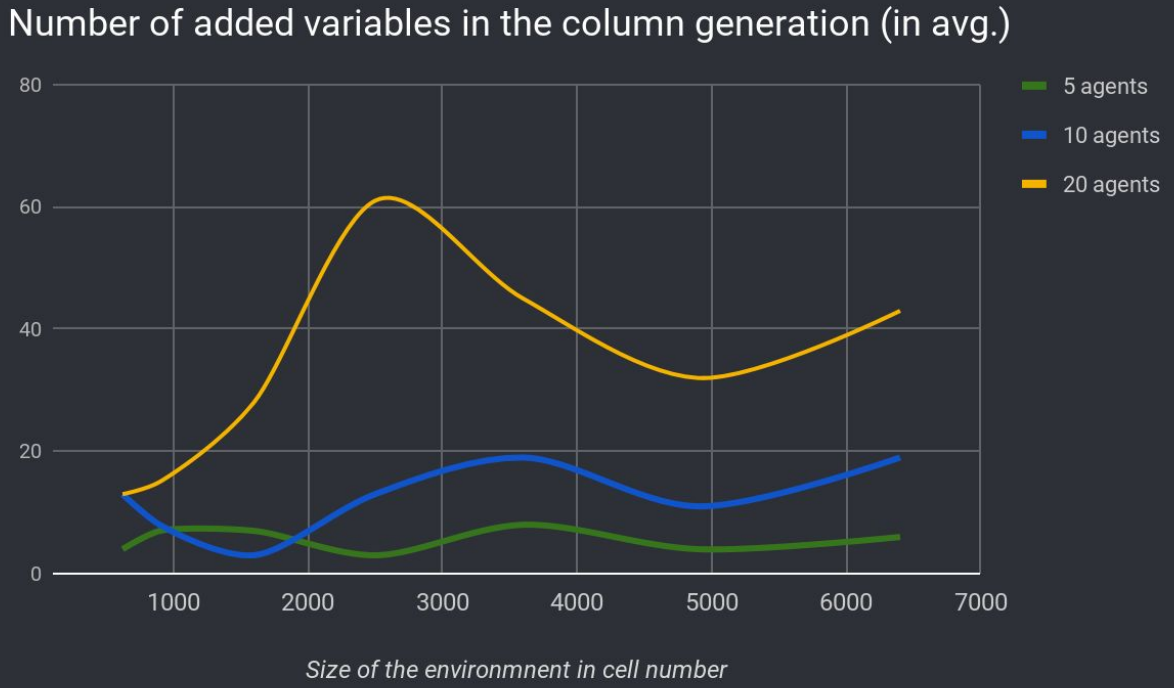
Flow conservation & Supply and demand

$$x_{ij}^k \in \{0,1\}, \qquad \forall (i,j) \in A, k \in [K]$$

Comparison with previous solution



More analytics for the column generation method



Solved examples