Using RL to perform Vehicle Re-Scheduling with Efficiency and Scalability

Son Pham

Johns Hopkins University, Baltimore, Maryland
spham9@jh.edu

Abstract. Scheduling trains to arrive at their destination on time with minimal delays in a shared network is an enormously valuable problem but is mathematically NP-hard. Traditionally, this problem is solved through tools and methods in the field of Operation Research such as CPLEX, SAV, PA, CFN, etc. These tools scale non-linearly as the problem gets larger and, while sufficient for initial route planning, often struggle when rescheduling routes under time limit when the network has disruption. In this paper, we propose solving this problem under a Reinforcement Learning approach. We believe that Reinforcement Learning architecture and methodology offer unique advantages in terms of scaling and allow for solutions that are both efficient, and easily adaptable to network disruption and additions of new tracks and trains.

Keywords: Reinforcement Learning · Vehicle Re-Scheduling Problem

1 Introduction

Train transportation is a vital piece of infrastructure in any major city in the world. They often act as an affordable mobility option for the middle class, helping millions of people move around and facilitating countless economic activities within the cities. Yet, this vital economic and community-building engine often came at an incredible initial investment from the government. According to an estimate, the cost-per-mile for ground and underground subways in the developed world is estimated to be north of $350 million per mile and even reaches $1 billion per mile in some major cities such as New York or Los Angeles.

There is also another problem with train traffic management. Sometimes, the traffic density in certain sections of the rail network is so high that it requires many long trains moving back and forth in the same section to satisfy the traffic demands. Facing this problem, the city has two options:

1. Build parallel lanes of tracks for those specific sections, which would cost multiples of the aforementioned cost.
2. Build single lanes and rely on intelligent planning to avoid trains overlapping each other.

Cities obviously favor the second choice whenever possible to save cost, but the intelligent planning of trains to reach destinations with minimal time and

avoid other trains is known as the Vehicle-Rescheduling Problem in the domain of Operation Research (OR). This problem has been proven to be NP-hard [3], as the problem gets combinatorially more difficult as more trains and rails get added to the network. This difficulty is further compounded by the fact that trains can break down or move slower than expected, meaning that the entire schedule has to be recomputed, often under a time limit of just about 10 to 30 minutes. Many current OR algorithms such as CPLEX, SAV, PA, CFN simply cannot compute a solution in time once the size of the problem gets too large [2].

2 Our proposal

In this paper, we propose solving the Vehicle Re-scheduling Problem using the Multi-agent Reinforcement Learning (MARL) framework. Our training framework results in a fixed size neural network acting as an agent that makes decisions on the fly on whether the trains should break, reverse, turn left or turn right at each intersection. Framing the problem as a MARL problem provides several advantages:

1. Each agent is controlled by a fixed-sized neural network. Therefore, introducing a new agent will only add a linear complexity to the problem. In fact, it is possible to use the very same network for the new agent. The new agent decision making will cost the same amount of time as any other agents. Given the modern GPU technology, the cost of inference in a neural network is often very small, even in extremely large networks.
2. Reinforcement Learning method has been shown to adapt gracefully to change in situations since these changes are already accounted for during the training process. When one train broke down and stopped in the middle of the track, this only resulted in a changed input to the network. The agent should readily adapt without triggering total path recalculation as in the OR methods [1].
3. If the agent is already trained in a variety of rail network environments, then it can readily adapt when a new path is introduced to the network. The agent should be able to readily operate with the newly introduced path to the network.

Thus, providing that the agents are trained in a sufficiently large and diverse set of rail network environments, we potentially transform all the expensive re-scheduling costs of OR methods into upfront training cost of Machine Learning. The recalculation time limit merely becomes the inference time limit for the neural network model. With the recent advances in GPU computing, this is a non-issue; it has been shown time and time again that several minutes are more than enough even for extremely large neural network models with billions of parameters.

3 Problem formulation

3.1 Rail network

Any rail network can be reformulated as a graph of nodes and edges, where each edge represents a consecutive uninterrupted track of rail, and each node represents a transition between a track. In railway standards, there are 7 different types of transitions, as shown in figure 1 below.

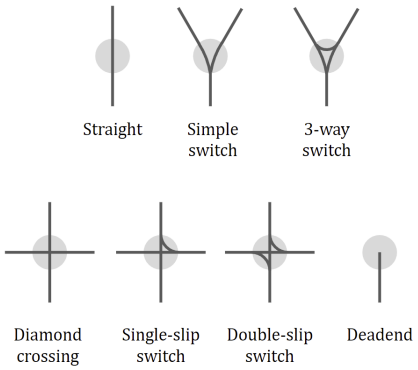

Fig. 1: 7 different types of switches in a standard rail network

Using these 7 types of transitions, we can model any rail network in the world [4]. Note that in some cases, not every edge is accessible. For example, in a simple switch transition, a train can switch from a root track to either a left track or a right track, but a train coming from a left track cannot switch to a right track, and vice versa.

A train is defined as occupying parts of the network equaling the length of the train. It can move back and forth along the edge of the network. and no two trains can occupy the same part of the edges. The head of the train can make a turn in the transition, and the train will occupy the transition until its entire length has moved off that transition. No other train can occupy the same transition during this occupation. Transitions are where the main trade-off of the Vehicle Rescheduling Problem happens:

- A train must decide whether to make the turn and block the other train or wait until the other train passed.
- A train must decide whether to take the shorter but busier route that has a higher chance of collision, or take a longer but emptier route with a lower chance of collision.

Each train also has a breakdown probability per unit length, which we used to simulate the occasional situations where trains break down. When it does, the train is stopped for a certain amount of time until being functional again.

3.2 Problem definition

A network can have as many trains as its tracks can support. A train has a starting location and a destination, defined as a point arbitrarily placed in the middle of the edge. The goal is to make all trains reach their respective target in the least total amount of time possible.

To transform this objective into the reward for our RL agent, we set a generous maximum time window M in which train must complete their trips. All n trains must reach their destination in the least amount of time possible. We will divide this total time by M * n to normalize the time to a number between 0 and 1 and treat this as a normalized loss. Taking 1 minus this loss, we form a normalized reward value, allowing the agent to be compared across different maps that might have different number trains, map lengths, map formations, and time limits. In mathematical terms, this reward value can be formulated as the following:

$$R = 1 - \frac{\sum_{i=1}^{n} t_i}{Mn} \tag{1}$$

If $R = 0$, this means that none of the trains reached their destination before the time limit. If $R = 1$, this means that all trains reached their destination immediately, which is an unlikely scenario. We want our agent to have a completion rate of close to 100%, while the reward to be as close to 1 as possible.

4 Methodology

4.1 State observation

To create our RL agent, we first need to convert the local observation of each train into some form of a fixed vector of values. We exploit the fact that the rail network is a graph and the fact that trains can only turn left or right to build observation along with the allowed transitions in the graph. The observation is generated by spanning 2-branched trees from the current position, each representing the possible path the train can take if turning left or right. Since the train can perform a u-turn (by switching head and tail), we also branch the tree from the tail of the train as well to make sure the train is aware of what is behind it. The following figures describe how this branching mechanism works when we branch 3-deep.

For each selected branch, we will collect the following features about that branch:

1. Remaining distance to complete the branch (normalized by the branch distance).
2. Whether the train's destination lies within the branch.
3. Remaining distance to target if the train's destination lies in the branch (normalized by the branch distance), false otherwise.
4. Whether there is another train in the branch.

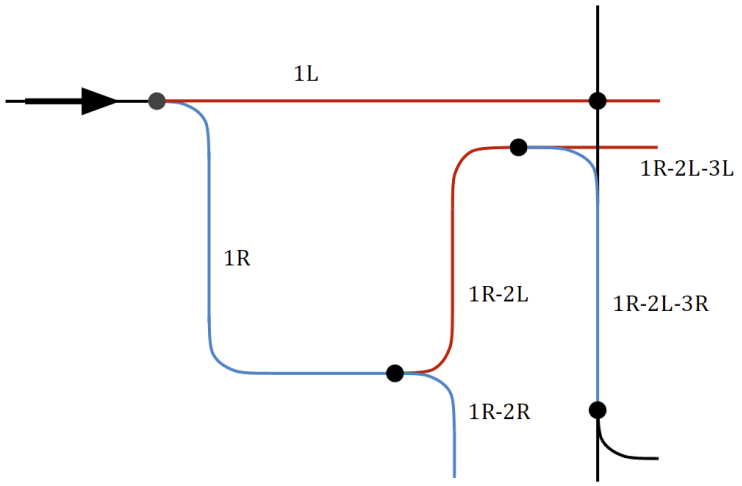

Fig. 2: An example of branches selected by branching 3-deep

5. Remaining distance to that train (normalized by the branch distance).
6. Whether there is a possible conflict with another train running across.
7. Remaining distance to the conflict point (normalized by the branch distance).
8. Whether there is an unusable switch in the branch.
9. Distance to the unusable switch (normalized by the branch distance).
10. Minimum distance to target if the branch is taken (normalized by total map length).
11. Number of trains that are moving in the same direction (normalized by the total number of trains).
12. Speed ratio of the slowest train in the same direction.
13. Speed ratio of the fastest train in the same direction.
14. Number of trains that are moving in the opposite direction (normalized by the total number of trains).
15. Speed ratio of the slowest train in the opposite direction.
16. Speed ratio of the fastest train in the opposite direction.

Each of these branches will contain these 16 numerical values. If there isn't a branch, all 16 values will be filled with -1. We will then stack all these values on top of each other into a long vector, as demonstrated by the following figure. This gives us a fixed-length observation based on which an agent can be trained to make a decision.

4.2 Model

We used a Dueling Double DQN model to train the agent [5]. The model contains two hidden dense ReLU layers of size 128. It takes the full-size observation vector and goes through the two hidden layers to produce a compressed observation vector. This compressed vector will then be used to predict two values:

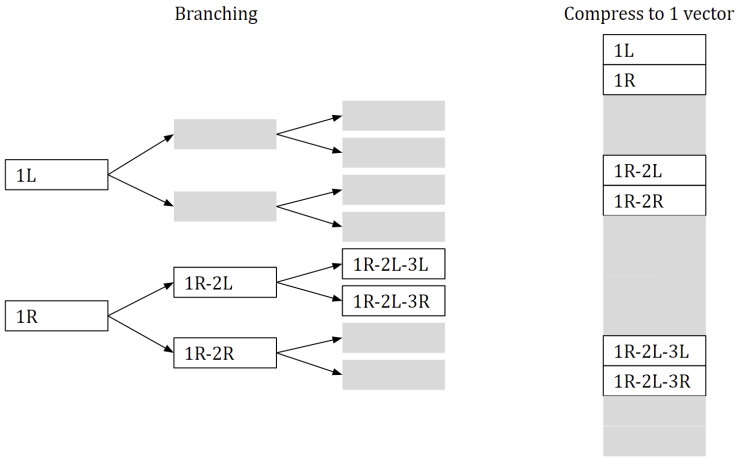

Fig. 3: An example of branches selected by branching 3-deep

- V(s) value representing the value of the state that the train is currently in.
- A(s, a) value representing advantage value of the state action pair, where the train will take one of 4 actions $BREAK$, $U-TURN$, $TURN-LEFT$ or $TURN-RIGHT$.

Based on these two pieces of information, we can calculate the following Q-value.

$$Q(s,a) = V(s) + A(s,a) - \max_{a' \in |A|} A(s,a) \tag{2}$$

Then for each step, we presume that the optimal action is one would maximize the Q-value.

$$a* = arg \max_{a' \in |A|} Q(s,a) \tag{3}$$

To ensure that our trains properly explore the space during the training phase, we will perform the classic epsilon-greedy decision making, where for ϵ amount of time, the train will make a random decision. ϵ will slowly decay over time after each episode to slowly shift the training processing from exploration to optimization. The ϵ function by the episode number is calculated as the following:

$$\epsilon(t) = \min(0.99^t, 0.01) \tag{4}$$

For each episode, each train will continuously use the above equations to make decisions that control the trajectory of the train and collect tuples of the train state, action, and reward (s, a, r). All these tuples from all trains will be stored in a replay buffer. After each episode, we will sample randomly from this replay buffer and use them to update the network each episode.

4.3 Data generation

As mentioned, a diverse set of rail networks is critical to the training of the agent. We want the agent to be exposed to as diverse a set of the network as possible. Ideally, we should sample all the rail networks in the real world. Unfortunately, this will likely require an amount of labor too large for this project. We settle for procedural content generation of random rail networks.

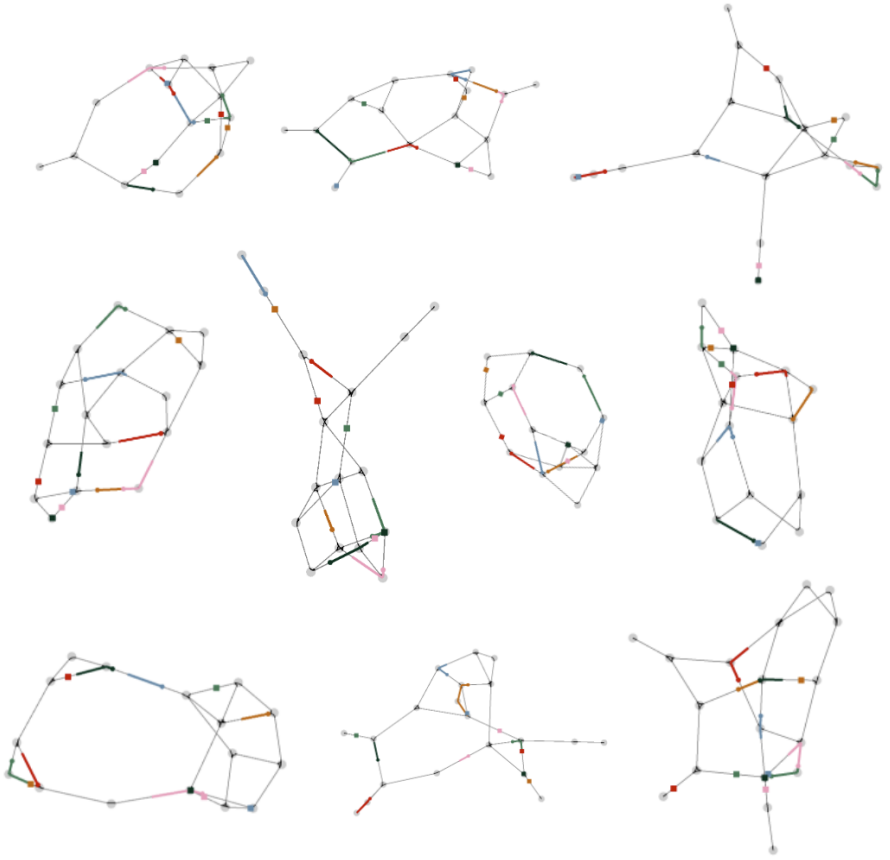

Fig. 4: Examples of generated networks and train placement

Since any rail network is a graph, the problem boils down to generating a random graph for every episode. The process can be generalized as the following algorithm:

1. Randomly generate a set of m transitions.
2. Since each transition has a unique degree, we now have a degree sequence. We randomly connect the edges under the degree constraint of each node. These edges form the tracks of the network

3. Given the graph of nodes and connected edge, we used the Fruchterman-Reingold force-directed algorithm to "spring" out the network to make sure that each node is reasonably distanced from other nodes.
4. For each of the n trains, we take a pair of edges, place the train on one edge, and place the destination on another edge.

Since there is randomization in step 2, 3, 4, this results in a very diverse set of networks that will help the agent train in a variety of environments. The following figure shows a sample of 10 networks with train and destination placement that we generated with $m = 15$ and $n = 6$.

All train speed is set to 1000 unit length per second, has a breakdown probability of 10^{-5}, and a breakdown time of 1 second. The positions of all nodes and edges in the railed network are scaled in such a way that the average length of each edge is 400 units. Each scenario is restricted to a maximum time of 20 seconds. Effectively, this means that the trains have to reach the destination in, on average, 50 transitions.

5 Results & Potential Improvements

Fig. 5: Training results

In the beginning, the performance is very poor as trains only move randomly. Over time, we observed that agents started to catch up on basic facts such as trains should not simply move forward most of the time. As time goes on, trains start to learn how to guide itself towards areas of the network that are less dense to avoid collision with other trains. The training performance reached its peak at around 70% completion rate, with a normalized score of 0.50 in episode 500, and stay that way for the rest of the training process.

While our results demonstrate that it is possible to train an agent through the MARL framework, we found that our agent still has areas for improvement. Firstly, agents seem to favor moving and rarely use the breakes. When the agent is in the middle of a track and wants to keep staying in the track waiting for another train to pass the transition, we found that trains like to keep performing $U-TURN$ and move back and forth in the middle of the train. We believe this comes from the fact that trains don't have any penalties for moving, and that it is generally better in terms of maximizing time to move as much as possible. This can be solved by encoding a penalty term if the train decided to move, which fits the real-world incentive of saving fuel while operating the trains.

Secondly, we also observed that trains are primarily optimized to perform on their own, but lack coordination. Combined with the fact that all trains are controlled by the same agent, this leads to certain situations where trains perform individually optimal decision that is not necessarily optimal for the whole group. An example can be seen in figure X below. Both trains have to reach the destination on the other side and can choose either the short path or the long path. In this case, the locally optimal schedule is to pick the shorter path, but the globally optimal schedule is for one train to pick the short path and one train to pick the long path. What we have observed with our agent is that when both trains reach the transition where they have to make a decision at the same time, both end up choosing the shorter path, leading to inefficiency, as one train has to turn back to go to the longer path. This could be avoided by using neural network models that allow trains to communicate with each other such as ComNet.

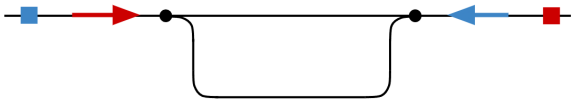

Fig. 6: An example of conflicting situation where trains perform individually optimal, but globally unoptimal actions

Thirdly, we also note that in the real-world, trains tend to be deployed in a fixed rail network. A train only has to perform in one network really well, instead of having to perform well in diverse scenarios that we simulated in our paper. This opens opportunities for fine-tuning the agent for a particular rail network to gain further performance.

Finally, we observed the agent has finished training, the decision inference takes a measly 0.0004 seconds. The time limit to recalculate a decision is often in the range of minutes. As such, wee can still scale the network much further, or add many more features in the network to obtain higher performance without sacrificing the quick inference time advantage that this method has over the OR methods.

6 Conclusion

In this paper, we showed that it is possible to solve the Vehicle-Rescheduling Problem by using the Multi-Agent Reinforcement Learning framework. The framework allows us to train agents that not only perform well, but also can adapt to when there are changes in the network, or when trains are added to the network. Decisions are always made in a very short inference time of the neural network, which provides a crucial advantage compared to existing OR methods where the entire scheduling has to be recalculated once the train breaks down. We believe that this advantage is crucial to managing not only rail networks but any large network of transportation at scale where OR methods simply cannot compute a solution in time. While we still have issues with fuel economy, coordination, and fine-tuning, these are strictly RL modeling issues that can be resolved without taking away the crucial advantage in inference time and are a rich area for improvement for researchers who are interested in this problem in the future.

References

1. Igl, M., Ciosek, K., Li, Y., Tschiatschek, S., Zhang, C., Devlin, S., Hofmann, K.: Generalization in reinforcement learning with selective noise injection and information bottleneck. Advances in neural information processing systems **32** (2019)
2. Lenstra, J.K., Kan, A.R.: Complexity of vehicle routing and scheduling problems. Networks **11**(2), 221–227 (1981)
3. Li, J.Q., Mirchandani, P.B., Borenstein, D.: The vehicle rescheduling problem: Model and algorithms. Networks: An International Journal **50**(3), 211–229 (2007)
4. Mohanty, S., Nygren, E., Laurent, F., Schneider, M., Scheller, C., Bhattacharya, N., Watson, J., Egli, A., Eichenberger, C., Baumberger, C., et al.: Flatland-rl: Multi-agent reinforcement learning on trains. arXiv preprint arXiv:2012.05893 (2020)
5. Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., Freitas, N.: Dueling network architectures for deep reinforcement learning. In: International conference on machine learning. pp. 1995–2003. PMLR (2016)