Multi-Agent Reinforcement Learning on Trains using Classical AI Search Technique

Anthony Leamer[1], Rajat Sharma[1], Mohammed Shafakhatullah Khan[1]
Charles University[1]
anthonyleamer@gamil.com, rajatsharma3200@gmail.com, mdshafakat91@gmail.com

Abstract

The goals of Flatland Challenge are resolving problems of train scheduling and rescheduling. To address these issues we used AI search and planning techniques. In this report we produce the classical AI search and planning techniques to review transport planning problems with the help of the framework provided by NeurIPS flatland Challenge. Multi-Agent Path Finding (MAPF) is a problem of finding paths for multiple agents and those paths must be collision free. To solve the scheduling and rescheduling issues of dense railway network we are using an optimized technique of MAPF. The techniques used will resolve collision and deadlock problems and provide a smooth transportation environment.

1 Introduction

NeurIPS 2020 Flatland Challenge is a railway scheduling competition which was held in partnership with German, Swiss, and French railway companies. This research challenge deals with the real key problem in the transportation world. The Flatland Challenge (Mohanty et al. 2020) is a research competition designed to come up with solutions addressing transportation issues not only in railway but also in other areas of transportation and logistics "How to efficiently manage dense traffic on complex rail networks?" this competition is organized by AIcrowd, and this edition of the challenge is affiliated with the AMLD2021 and ICAPS 2021 conferences. This is a real-world problem faced by many transportation and logistics companies around the world such as the Swiss Federal Railways, Deutsche Bahn and SNCF (Li. J, et al. 2021).

The Flatland challenge was initiated in the year 2019, the key concept of it is to answer "How can trains learn to automatically coordinate among themselves, so that there are minimal delays in large train networks?", at the core of this challenge lies the general vehicle rescheduling problem (VRSP) (Li et al. 2007). In 2020, the organizers came up with new issues regarding transportation and added them onto the 2019 version. The Flatland Challenge is a train planning problem. The task of this competition is to design a plan such that the trains reach their goal / destination position within a time limit without colliding with each other.

Multi-Agent Path Finding (MAPF) deals with multi- agent path finding problems, how to move agents from start to target locations on a graph without vertex and edge collisions (Stern R et al. 2019).

2 Flatland Challenge Environment

2.1 Problem Definition

Flatland environment is the core concept for a simulation which contains all of the concepts like the railway network itself (turns, one-ways, crossroads, turnouts, etc.) and its agents (trains). The railway networks comprise of a 2D rectangular grid with width and height, and number of cities and stations. Each city contains multiple parallel rail tracks, and each rail track in a city contains one or more stations. Let's assume that there are k trains t_1, t_2, $t_3 \dots t_k$ each of them has its starting point and a destination point, here we discretize time into timesteps from 0 to T_{max}. To maximize the reward, we give commands to the trains at every timestep so that we move as many trains as possible to their goal cells as soon as possible. We are interested in each agent reaching its destination but we also strive to reach a global goal that all trains reach their destinations.

Each grid is of size 1 x 1 and contains one of the seven base types of tracks as shown the figure below Figure 1, that determines how the train can move through the cells. In addition to that, some of these base track types can be rotated in up to four directions. This creates up to 27 different rail types.

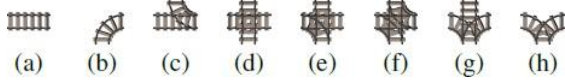

Figure 1: Eight rail types: (a) straight, (b) curve, (c) simple switch, (d) diamond crossing, (e) single slip switch, (f) double slip switch, (g) tri-symmetrical switch, and (h) symmetrical switch (Li. J, et al. 2021).

The Flatland grid contains sparse railways that essentially create limitations on the movement of the agents. Trains must legally move around the grid through these various rail types without colliding. Another rather important conflict that must be solved in the challenge are deadlocks where trains get stuck in a position and cannot – without interruption – continue to their final destinations. The basic conflicts are when two or more trains occupy the same cell or when they exchange positions at same timestep – that is forbidden and must be solved at the beginning.

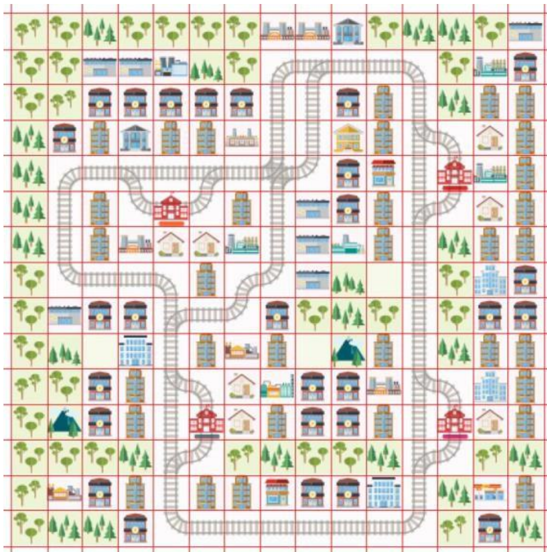

Figure 2: Flatland map with a grid before the execution of any plan. (Source: AIcrowd [2020])

Each Flatland object has an assigned timestep limit and all trains are required to reach their target within that limit. The train which does not reach its goal position within that timestep limit will be kept on an incomplete trains list. For each train that is on the incomplete list there is a penalty subtracted from the reward function. Time is discretized into timesteps from 0 to T_{max}, where

$$T_max = 8 * (w + h + \lceil |A|/|C| \rceil). \qquad \textit{equation 1}$$

In equation 1, w is the grid width, h is the grid height, A is the number of trains in the problem and C is the number of cities in the grid. The task is to command the trains in such a way that as many as possible get to their target stations without collisions in the shortest time. With these individual timestep costs, we will also analyze the total time (make-span) until the last agent arrives at its goal point. This gives birth to a reward function that is used and evaluates the efficacy of various algorithms on a local and global scale.

Timestep 0 means there are no trains in the environment. To make the train appear in the environment with its initial orientation, which occupies one cell, we provide the departure time of the train and pass a command at that time. During the execution, the train makes only one action and occupies a single cell at each timestep. As per the commands passed, the train behaves accordingly and leaves the environment as soon as it reaches the destination position if and only if there are no conflicts during the transition and it doesn't suffer a malfunction. If a conflict occurs or a malfunction happens then the train becomes still for a given number of timesteps and it cannot make any sort of transition. This helps raise issues in the execution as the penalties inquired by these delays are visible in the final score.

The train has to perform a particular action at every timestep from the start to the target position, either it has to wait or move forward, turn left or right. Collision of trains occurs only when two trains enter the same cell at the same timestep (or pass through each other). The Flatland Challenge considers the solution as valid if the trains reach their targets within a reasonable timeframe. In the environment we will come across four types of conflict situations a) tile conflict (collision) b) following conflict (can be eliminated by introducing elementary timesteps) c) cycle conflict (form of deadlock) and d) swapping conflict, illustrated in Figure 3 below.

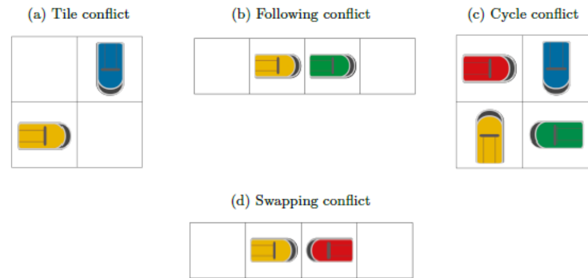

Figure 3: Conflict Situations. (Source: AIcrowd [2020])

2.2 Examples

The following Figures 4 and 5 are visualizations of the Flatland problem with 4 and 32 agents respectively. Results obtained after running these simulations provided by AIcrowd can be seen in Figures 6 and 7, where the individual agents moved around the grid randomly. The five scores in each figure are reflecting five separate runtimes of the algorithm with again 4 and 32 agents respectively. The reward function takes into account runtimes and the number of trains that successfully got to their target locations on time. The precise calculation of the reward function can be found in the Flatland source code documentation.

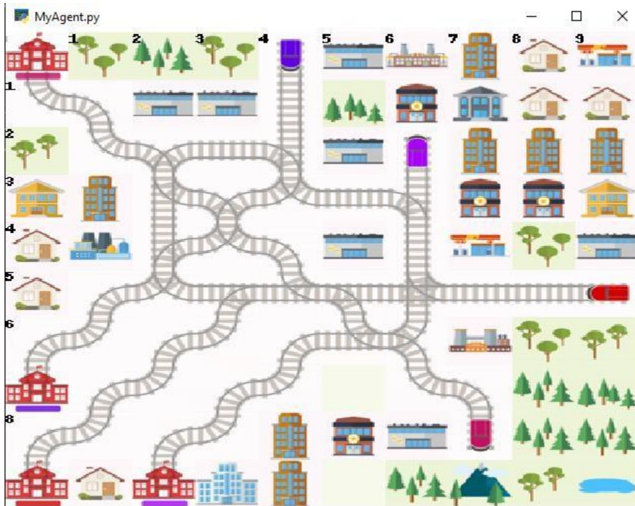

Figure 4: 10 x 10 grid environment with 4 trains.

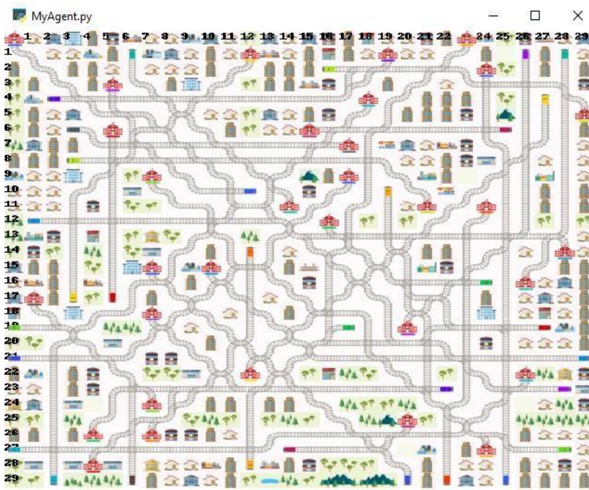

Figure 5: 30x30 grid environment with 32 agents

```
open_window - pyglet
Episode Nr. 1    Score = -180.0000000000011
Episode Nr. 2    Score = -129.2999999999992
Episode Nr. 3    Score = -149.09999999999994
Episode Nr. 4    Score = -116.99999999999925
Episode Nr. 5    Score = -148.79999999999993
```

Figure 6: Scores of 5 Episodes for 4 trains.

```
open_window - pyglet
Episode Nr. 1    Score = -4526.700000001284
Episode Nr. 2    Score = -3453.000000000633
Episode Nr. 3    Score = -3737.1000000008053
Episode Nr. 4    Score = -4181.1000000010745
Episode Nr. 5    Score = -4723.8000000014035
```

Figure 7: Scores of 5 Episodes for 32 trains

3 Literature Review

3.1 Multi-Agent Path Finding

In the AI community, MAPF is one of the most researched domains. Finding an optimal path for all the agents from their initial state to the goal state without causing collision during their transition is NP-hard (G. Sharon et al. 2015). And while the problem is NP-hard, modern multi-agent pathfinding algorithms are able to find optimal paths for more than 100 agents in reasonable time. In most cases there will be an additional goal to reduce the sum of timesteps required for each agent to reach its goal state. Hence, lot of research in this area is done on finding the appropriate heuristics to solve the problem as quickly as possible. In recent days this research has practical applications in airplane taxiway scheduling (Li. J et al. 2019), robot routing (W. Honig et al. 2018, 2019), traffic control, robotics, aviation and video games, etc. (G. Sharon et al. 2015).

To solve the MAPF problem, the algorithms used are categorized into two classes: optimal and sub-optimal solvers. We usually apply optimal solvers when the number of agents is relatively small and the task is to find an optimal and minimal-cost solution. Whereas suboptimal solvers are used to find paths when the number of agents is high and finding the optimal solution is NP-hard.

3.2 A* Search

The A* search is an informed, best-first search algorithm. It is an extension of the Dijkstra algorithm by a heuristic function which makes the search more efficient – especially useful when working with large state spaces. It can be simply modified to be relevant for prioritized planning as well. The A* builds a tree of possible paths originating from the start node and always extends these paths one edge at a time until a certain termination criterion is met. The path extension order is determined by expanding nodes that minimize the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ represents the cost of travelling to the node n from the agent's start node and $h(n)$ represents the estimated cost of the cheapest path from node n to the target node based on the heuristic function. The A* algorithm terminates when the target node is expanded or when there are no more nodes to expand.

The A* algorithm is complete when used on finite graphs with non-negative edge weights (Russell & Norvig [2009]) and optimal if the heuristic function is admissible. An admissible heuristic function is a distance function h that never overestimates the cost of getting to the target node from the current node. However, if the heuristic function is only admissible, it does not guarantee that $g(n)$ obtained upon first expansion of the node is optimal. In order to guarantee that the $g(n)$ of a node is optimal upon its first expansion, we must require that the heuristic function is also consistent.

3.3 Conflict Based Search

Conflict Based Search (CBS) - as summarized by (Sharon et al. 2015) - is a tree-based search algorithm used to find the optimal solution for multiple agents by decomposing MAPF into a number of constrained single-agent pathfinding CB Searches. It is a two-level algorithm that is complete and

optimal and is able to solve large instances of MAPF problems. All these problems are resolved in time proportional to the size of the map and length of the solution, but there is a possibility of having an exponential number of such single-agent problems.

At the beginning of the CBS the search tree contains only one node i.e., the root node without any constraints. Using this node, CBS – on its low level - finds the shortest path for the agents using one of the shortest path algorithms, mostly A* search algorithm is used.

Conflict tree (CT) is a binary tree that guides the overall search for the solution. Each node contains constraints for each agent's paths and the cost of the solution based on the objective. The solution is in the node with the lowest cost, where there are the fewest constraints, and all agents are satisfying them and there are no conflicts between their paths.

If nodes in the CT contain conflicts, we initially select the one with the lower cost and secondarily the one with fewer constraints (if they have the same cost). If they have the same cost and same number of constraints, one of them is arbitrarily selected and solved by splitting the node into two child nodes where each of them prohibits one of the conflicting agents from entering the conflicting cell. The CBS always splits the conflicting node into two, meaning it constrains only two agents in each step, even when there are more agents in conflict in the node in question. Before any implementation, this fact seems likely to play a negative role in the Flatland environment because the railways are sparse and contain a very high number of agents.

3.4 Prioritized Planning

Prioritized planning is a decoupled MAPF approach as it plans agent paths individually. It is sub-optimal because it doesn't look at a global picture and does not enforce any cooperation between agents that could be obtained through coupled approaches which plan all agents together.

Alike other successful participants using local search techniques we decided to solve a simplified problem by omitting potential breakdowns and deadlock situations and use a prioritized path planning algorithm to break down the number of trains into smaller subgroups. This algorithm is able to solve the Challenge in its first stages with few agents but with variable speeds (no tricky deadlock/breakdown situations). Based on the previous winners of the competition that used prioritized planning to reduce the number of agents that the CBS has to sift through, it is obviously a very effective approach. The question for testing is the group size.

The prioritized planning algorithm is simple and fast but is not generally complete nor optimal. In our Flatland setting where the grid starts empty and allows the placement of agents in any order as well as time-step changes and also the agents disappear immediately from the grid after reaching their destination makes the prioritized planning algorithm complete.

The prioritized planning approach is sensitive to the priority assignments to agents and is very sensitive to the correct assignment. There are various ordering heuristics that can be used to boost the prioritized planning algorithm. From research conducted on other Flatland participants we've noticed that Fast-First ordering was the most effective and decided to include it in our own implementation.

3.5 Large Neighborhood Search (LNS)

The LNS metaheuristic was proposed by Shaw. In LNS the neighborhood is defined implicitly by a destroy and a repair method. The destroy method destructs part of the current solution while a repair method rebuilds the destroyed solution. The destroy method typically contains an element of stochasticity such that different parts of the solution are destroyed in every invocation of the method. The neighborhood $L(s)$ of a solution s is then defined as the set of solutions that can be reached by first applying the destroy method and then the repair method. The main idea behind the metaheuristic is that the large neighborhood allows the heuristic to navigate in the solution space easily, even if the instance is tightly constrained. This is to be opposed to a small neighborhood which can make the navigation in the solution space much harder. The destroy method is an important part of LNS. The most important choice when implementing the destroy method is the degree of destruction: if only a small part of the solution is destroyed then the heuristic may have trouble exploring the search space, as the effect of a large neighborhood is lost. If a very large part of the solution is destroyed then the LNS heuristic almost degrades into repeated re-optimization. This can be time-consuming or yield poor-quality solutions dependent on how the partial solution is repaired. This is why an educated neighborhood selection strategy is essential.

3.6 Minimal Communication Policies (MCP)

(Ma, Kumar, and Koenig 2017) MCP's are decentralized robust plan-execution policies that can prevent collisions and deadlocks during plan execution for valid MAPF plans. Usually, they stop some trains so that the original plan maintains the ordering with which each train visits each cell. This ensures all trains reach their destinations within a finite number of timesteps.

4 Methodology

We have somewhat laid out our methodology higher, but to sum up:

On the lower level of our Conflict Based Search we use the A* algorithm which can quickly find paths for individual agents and is subject to restrictions imposed by the corresponding node in the CT. We adopted the space-time A* algorithm to find the shortest path for every agent that avoids collisions with the given paths of all other trains. Each state of space-time A* is a pair of a cell and a

timestep. We reviewed previous heuristics used for the A* algorithm in the Challenge and chose to use the *Distance Map* heuristic. It is consistent; hence the algorithm is admissible. It also showed more promise in the Challenge than the *Manhattan Distance* heuristic. For every agent, the heuristic calculates the distance to the agent's target for every reachable cell by the agent. The heuristic function is formulated as follows

$$h_{i,j} = min(argmin\ (c(\pi^0_{i,j}),\ c(\pi^1_{i,j}\ ),\ .\ .\ .),\ \infty),$$

we are searching for the minimal $\pi^k_{i,j}$ – the k-th possible path from node i to node j. function c takes a path and gives its length (cost).

We apply the high level of the CBS algorithm to subgroups of trains in the grid. We use Prioritized Planning to do so. The groups differ in size, we didn't restrict all groups to have a certain number of trains in each group. In different steps of the Flatland Challenge there are different numbers of agents in the grid, so we firstly use PP to break the agents into as many groups as there are different speeds of the trains. Then, if the number of agents in a group is higher than 8, we break them up into two either arbitrarily or preferably based on the distances from each of theirs targets. We assign higher priority to the ones that are faster and then also to the ones that are closer to their destinations. Individual path planning is done for the agents with higher priority first.

In our implementation we took care of agent collisions. We did so by creating conditions in the run of the CBS which then fed these constraints to the A* algorithm to find collision-free paths. However, deadlocks are a separate problem that require significant attention in the Challenge in order to produce high ranking results. We addressed them by introducing a Communication Policy (CP) that recorded the intersections that a malfunctioned train was supposed to pass through. All trains that were supposed to pass through each of these intersections are then stopped in such a cell, so they do not block any additional trains and before passing any of these intersections in question. We use Prioritized Planning to prioritize the list of potentially affected trains based on how close they are to any of these intersections in question. We then run a partial replanning subroutine on every train in this list in the order of its priority.

Firstly:

Step 1: create a queue of trains passing through the cell occupied by the malfunctioned train.

Step 2: sort the queue based on the number of timesteps that it takes each train to reach this occupied cell.

Step 3: pop the train with the highest priority and perform replanning on its route.

Step 4: repeat steps 1-3 until the queue is empty.

Secondly*:*

Step 5: create a queue of trains passing through any of the intersections that the malfunctioned train was supposed to pass through on its way to its destination.

Step 6: sort the queue based on the arrival times to any of these intersections in question (lowest time first).

Step 7: pop the train with the highest priority and perform replanning on its route.

As defined in the Flatland Challenge, the trains appear on the map at the start and disappear when they reach the goal state which makes our prioritized planning approach complete as it guarantees to find the optimal solution. At the beginning of the simulation, priorities are given to all the trains and are used throughout the simulation. Fast-First is the first ordering, if some trains have identical speeds, we apply the Near-First ordering that prioritizes the trains closer to their final destinations. In order to avoid the conflicts and make the trains reach their destination state safely, the highest priority should be given to the fast trains rather than the slower ones – this is a Fast-First heuristic that turned out to be more efficient considered to others (Rýzner 2020).

To farther improve our algorithm, we used the LNS to replan possible train paths for a given group of trains (a certain neighborhood of trains). The selection of neighborhoods we mostly overtook from last year's Flatland Challenge winners only with minor changes, mostly to the parameters. Our initial neighborhoods were always the size of 4 (unless there are fever trains left in the grid). The first two neighborhood selection methods are from (Li et al. 2021a): (1) the train-based strategy, which selects a train a_i with the largest delay and 3 trains that prevent train a_i from reaching its target cell earlier; (2) the intersection-based strategy, which selects *4* trains that visit the same intersection (i.e., cell of rail types (c) to (g) in Figure 1); (3) the start-based strategy, which selects *4* trains with the same start cell; and (4) the destination-based strategy, which selects 4 trains with the same target cell.

5 Experimental Results

5.1 Collisions

We got our implementation to work very successfully on levels with a maximum of 32 agents and no malfunctions. The CBS was very efficient in finding the collision bound trains and finding an optimal solution. We used Prioritized Planning for breaking down the overall number of trains that the CBS had to run on. The most effective approach was when we limited the largest possible group of agents entering a CBS run to 8. The smallest could be as small as 1 agent - based on its subgroup chosen by the prioritization.

In Figures 8 through 10 we can see three different settings with one smaller group limitation (maximum of 8 agents entering a CBS) and one larger group limitation (maximum of 16 agents).

In these figures runtime was the only factor we analyzed in percentages, with a runtime of 7865ms being 100%. It is a

limitation set for good representativity in our figures. It was produced as the hardest problem where we broke 32 agents down only into 2 groups of 16 agents for a CBS. For example in Figure 8, for 16 agents the runtime was 81% of the 7865ms cap, therefore 6370ms.
It was based on research we have conducted in Figures 8-10 (varying the maximal group sizes entering a CBS) that we decided to cap the group sizes entering a CBS at 8.

Note: To more accurately test the optimal maximal group for a CBS, we would have to conduct more expansive research and would need more information about the evaluation itself.

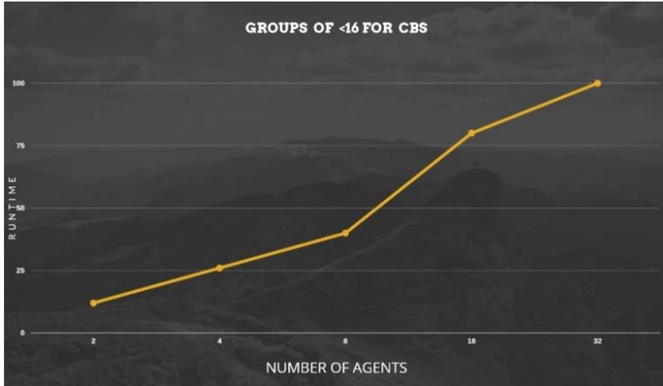

Figure 8: Max group of 16 in one CBS run

Figure 9: Max group of 8 in one CBS run

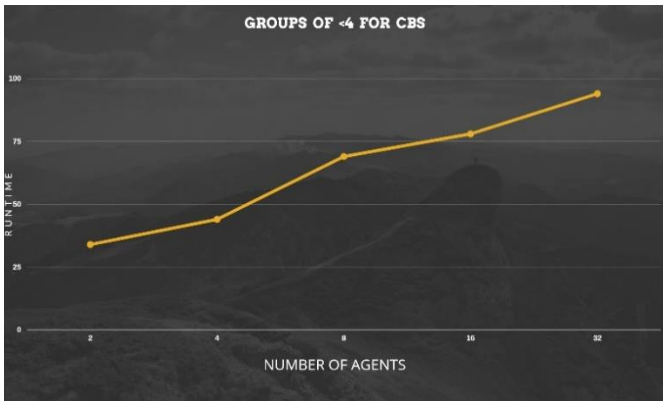

Figure 10: Max group of 4 in one CBS run

5.2 Examples

When we implemented our algorithm with all our heuristics and settings (as stated in Chapter 4) and with the knowledge from Subchapter 5.1 (groups entering a CBS are below 8 trains per group) and kept the default Flatland Challenge reward function, we obtained results that can be seen in Figure 11. The scores in Figure 11 were obtained with 4 and 32 agents in the environment, respectively. In comparison with the random agents that have no logic behind them, we have produced higher ranking scores (compare with Figures 6 and 7).

```
Episode Average     Score = -14.09
Episode Average     Score = -367.12
```

Figure 11: Average scores over 10 episodes for 4 and 32 trains resp.

The default Flatland reward function takes the number of trains that successfully arrived at their target location and their runtimes as arguments. Because both algorithms - default Flatland setting with random agents and our implementation – managed to get all the trains to their locations on time, the only unit the score reflects is runtime. Therefore, our algorithm had on average ten times shorter runtimes than agents moving randomly in the Flatland example.

5.3 Malfunctions

As we didn't manage to submit our techniques addressing malfunctions directly but have already implemented them, we ended up creating our own simulation and testing our replanning approach. We did effectively solve a malfunction problem by marking the intersections in question (as stated in the theoretical part) and running a CBS search with a constraint (blocked cell by malfunction and potential delays at intersections where the malfunctioned train was bound) for trains that had this cell or intersections in their path.
In our simulation – as can be seen in Figure 12 - we had only two trains. Train number 1 malfunctioned and was still supposed to pass through the highlighted intersection. Train 2 stopped a single step in front of the intersection in question and a replanning subroutine was called because train 1 was initially supposed to go through the intersection first. The new paths will have train 2 passing through the intersection first as the previous route is now blocked but there is an alternative route and because train 1 is assigned the lowest priority in the replanning as the malfunction may not be solved quickly.

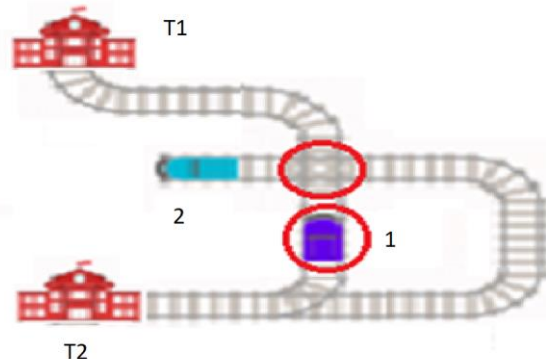

Figure 12: 20x20 grid with 2 agents and 2 Target stations

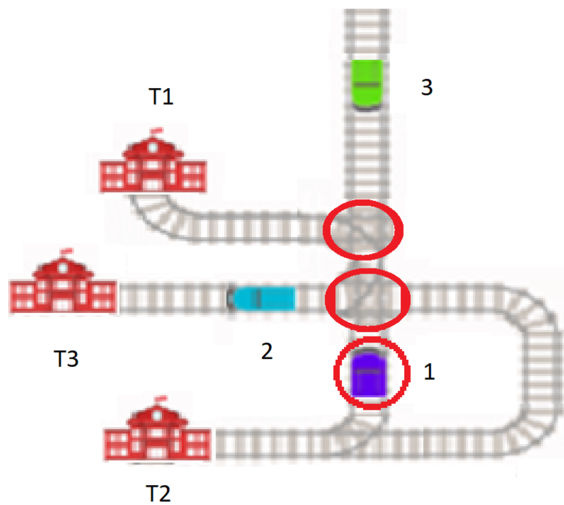

Figure 13: 20x20 grid with 3 agents and 3 Target stations

In Figure 13 we complicated the situation by adding one more train and another intersection influenced by the malfunction of train 1.
Step 1: List all the intersections affected by the malfunction (highlighted).
Step 2: Calculate the times it will take all the trains to reach an affected intersection.
Step3: Prioritize the list of affected trains by the shortest time it will take them to reach the affected intersection.
Step 4: Replan the train on the top of the prioritized list.

In our simulation the first train that needed replanning was train 2. Our algorithm found the alternative around the malfunctioned train 1. Secondly, it replanned train 3.

Possible upgrade to our malfunction problem is to replan the trains before they stop near the affected intersection. We replan at the point when all the trains get to the affected intersections making our prioritizing redundant.

6 Conclusion

Apart from last year's winners of the Challenge, we used a less adaptive LNS, which could have caused lower yielding scores in the more advanced rounds of the Challenge compared to theirs. When we saw on the first levels of the Challenge that the first method of choosing a neighborhood for replanning ((1): picking a train with the longest route and then 3 other trains at random – a slight change from our initial thought of choosing the 3 other trains based on if they play a role in delaying the initially chosen train) was the most effective and we made it our default.

The Flatland Challenge was a challenging and sophisticated task for us to do for the AI 2 seminar. In this report we try to resolve the conflict between the trains to have a smooth railway transportation, using A* search and MAPF solvers. There are many other techniques available which are complete and optimal MAPF algorithms, even the previous competitors of the Challenge have used techniques like CBS, push and rotate, LNS and won the challenge.

7 Contributions

A. Leamer: Extensively researched the entire Flatland Challenge API and figured out where and how to incorporate the proposed classical AI algorithm code in the adopted Flatland Challenge Environment. As well as working hardly on the final report.

R. Sharma: Collection of Data, for approaches and techniques related to Flatland Challenge in order to address the issues of the Challenge and from the AI crowd to prepare the Final Report and Presentation. Even, worked on result simulation.

M. S. K: Implementation of the code viz., creating an agent which uses the search algorithm, developing custom observation, MAPF solver for replanning to resolve the Flatland Challenge issues.

References

Atzmon, D., R. Stern, A. Felner, G. Wagner, R. Barták, & N.-F.Zhou (2018): "Robust multi-agent pathfinding" in "Eleventh Annual Symposium on Combinatorial Search"; pp. 1-9

Atzmon, D., R. Stern, A. Felner, N. R. Sturtevant, & S. Koenig (2020): "Probabilistic robust multi-agent path finding" in "Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)"; pp. 29-37

Barták, R., J. Švancara, & M. Vlk (2018): "A scheduling-based approach to multi-agent path finding with weighted and capacitated arcs" in "Proceedings of the Conference on Autonomous Agents and Multiagent Systems (AAMAS)"; pp. 748–756

Bnaya, Z., R. Stern, A. Felner, R. Zivan, & S. Okamoto (2013): "Multiagent path finding for self-interested agents" in "Sixth Annual Symposium on Combinatorial Search"; pp. 38-46.

Botea, A. & P. Surynek (2015): "Multi agent path finding on strongly biconnected digraphs" in "Twenty-Ninth AAAI Conference on Artificial Intelligence"; pp. 2024-2030.

Boyarski, E., A. Felner, R. Stern, G. Sharon, E. Shimony, O.Bezalel, & D. Tolpin (2015): "Improved conflict-based search for optimal multi-agent pathfinding" in "Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)"; pp. 740-746.

Cohen, L.; Uras, T.; Kumar, T. K. S.; and Koenig, S. (2019): "Optimal and Bounded-Suboptimal Multi-Agent Motion Planning." In "Proceedings of the Twelfth International Symposium on Combinatorial Search (SoCS 2019)"; pp. 44–51.

De Wilde, B., A. W. Ter Mors, & C. Witteveen (2014): “Push and rotate: a complete multi-agent pathfinding algorithm” in “Journal of Artificial Intelligence Research 51 (2014)”; pp. 443–492.

Filip Rýzner. (2020): “Multiagent path-finding for trains with breakdowns” In “Czech Technical University in Prague” https://dspace.cvut.cz/bitstream/handle/10467/87776/F3-BP2020-Ryzner-Filip-BP_FILIP_RYZNER_2020.pdf; pp. 1-74.

Hart, P. E., N. J. Nilsson, & B. Raphael (1968): “A formal basis for the heuristic determination of minimum cost paths” in “IEEE Transactions on Systems Science and Cybernetics SSC-4(2)”; pp. 100-107.

Ho, F.; Salta, A.; Geraldes, R.; Goncalves, A.; Cavazza, M.; and Prendinger, H. (2019): “Multi-Agent Path Finding for UAV Traffic Management” from “Proceedings of the Conference on Autonomous Agents and Multiagent Systems (AAMAS)”; pp. 131–139.

Hönig, W., S. Kiesel, A. Tinka, J. W. Durham, & N. Ayanian (2019): “Persistent and robust execution of mapf schedules in warehouses” in “IEEE Robotics and Automation Letters VOL. 4”; pp. 1125–1131.

Hönig, W., T. K. S. Kumar, L. Cohen, H. Ma, H. Xu, N.Ayanian, & S. Koenig (2017): “Summary: Multi-agent pathfinding with kinematic constraints” in “Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)”; pp. 4869–4873.

Li, J.; Gange, G.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. (2020): “New Techniques for Pairwise Symmetry Breaking in Multi-Agent Path Finding” In “Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)”; pp. 193–201.

Li, J., Chen, Z., Zheng, Y., Chan, S.-H., Harabor, D., Stuckey, P. J., Ma, H., & Koenig, S. (2021): “Scalable Rail Planning and Replanning: Winning the 2020 Flatland Challenge” in “Proceedings of the International Conference on Automated Planning and Scheduling, 31(1)”; pp. 477-485.

Mohanty, S. P.; Nygren, E.; Laurent, F.; Schneider, M.; Scheller, C.; Bhattacharya, N.; Watson, J. D.; Egli, A.; Eichenberger, C.; Baumberger, C.; Vienken, G.; Sturm, I.; Sartoretti, G.; and Spigler, G. (2020): “Flatland-RL: Multi-Agent Reinforcement Learning on Trains” in “CoRR abs/2012.05893”; pp. 1-25.

Sharon, G., R. Stern, A. Felner, & N. R. Sturtevant (2015): “Conflict based search for optimal multi-agent pathfinding” in “Artificial Intelligence 219 (2015)”; pp. 40-66.

Stern, R., N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. Satish Kumar, E. Boyarski, & R. Bartak (2019): “Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks” in “e-prints arXiv:1906.08291v1”; pp. 151-158.

Svancara, J.; Vlk, M.; Stern, R.; Atzmon, D.; and Bartak, R. (2019): “Online Multi-Agent Pathfinding” in “The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)”; pp. 7732–7739.

Walter, J. (2020): “Existing and Novel Approaches to the Vehicle Rescheduling Problem (VRSP): In “the Course of the Flatland Challenge by Swiss Federal Railways (SBB)” Master’s thesis, University of Applied Sciences Rapperswil, Rapperswil, Switzerland. pp. 1-68.

Wagner, G.; and Choset, H. (2017): “Path Planning for Multiple Agents under Uncertainty” in “Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)”; pp. 577–585.

W. Hönig, S. Kiesel, A. Tinka, J. Durham, and N. Ayanian (2018): “Persistent and robust execution of mapf schedules in warehouses” in “IEEE Robotics and Automation Letters”; pp. 1125-1131.

W. Hönig, S. Kiesel, A. Tinka, J. W. Durham, and N. Ayanian (2018): “Conflict-based search with optimal task assignment” in “Proceedings of the 17th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)”; pp. 757–765.