Flatland Challenge

Abstract

Flatland is a railway environment simulator that is used to develop multi agent path finding algorithms. This report aims to provide a description and analysis of the approach towards solving Flatland problems in three stages: routing a single train in a domain without time or collisions, routing sequential trains in a collision free manner, and finally tackling the full multi agent version of the challenge with train deadlines, malfunctions, and replanning mechanisms.

The approaches taken in each question are intended to progressively build out a full solution to this MAPF problem, where a single agent path planner evolves into a multi agent path finding planner. These approaches advance from using A* for the single agent problem, to using Prioritised Planning with Safe Interval Path Planning for planning agents in a fixed order, to ultimately utilising SIPP in the MAPF-LNS (Multi Agent Path Finding Large Neighbourhood Search) framework and implementing a replanning mechanism for the Flatland Challenge section of the assignment.

Single agent path finding

This is the simplest case of the assignment, a relaxed single agent version of the full multi agent challenge. Only a single train is routed in each instance. There are also no collisions or expected arrival times.

A* for single agent planning

The chosen approach was to implement A* for the rail map. A* is guaranteed to find optimal paths given an admissible and consistent heuristic and is simple to implement courtesy of the *piglet* package. A* aims to find the lowest cost solution by expanding nodes in the open list ordered by the function $\mathrm{f(n)} = \mathrm{g(n)} + \mathrm{h(n)}$, where $\mathrm{g(n)}$ is the current cost from the start node to node *n* and $\mathrm{h(n)}$ is the heuristic function calculated between node *n* and the goal node.

In Flatland, agents not only have an (x, y) coordinate in their state, but a direction as well which specifies the direction the agent is moving in. In addition, agents in Flatland are restricted in their movement to the tile types, which range from straight lines, to complicated dual switches and finally dead ends which are the only times agents can move backwards. All this is handled by the valid moves *GridTransitionMap* spits out for any given (x, y, direction) state tuple. This differs from more traditional 2D grid map domains, the effect of which means there are less available moves to an agent at any point in time. A* nodes therefore store both the location and direction in their states.

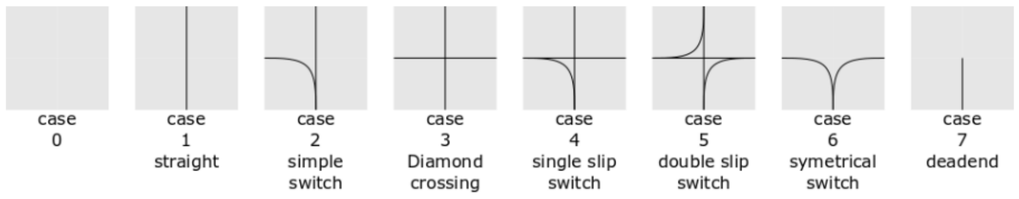

Figure 1 - Tile types in Flatland

The heuristic used was manhattan distance, as it is simple to calculate and has sufficient information to guide search in single agent domains. Manhattan distance is known to be admissible and consistent in four connected grid maps, of which Flatland is an example.

Multi agent path finding

The second question of the assignment deals with routing multiple trains sequentially, and no malfunctions occurring. Each agent is instantiated, has its path planned and actioned, moving onto the next agent. Naturally this lends itself to some form of collision free path planning method, as when more trains enter the grid domain the chance of a collision increases.

Prioritised Planning with TXA*

The concept for Prioritised Planning [Erdmann and Lozano-Pérez 1987] is to decouple MAPF into a series of single agent path finding problems, and plan agents according to some order. A variant is Cooperative A* [Silver 2005] where each problem in the series aims to avoid collisions with predecessors through the usage of a reservation table (or online collision checks), which provides a lookup of which agent occupies a given location at a point in time.

As a first test, the A* algorithm implemented in question 1 was transferred over to question 2. A* can be extended to Time Expanded A* by making use of the timestep_ attribute in the search node object. This changes the definition of an A* node from n = (x, y) to n = (x, y)@t where *t* is the timestep. As such, successor states may include the same (x, y) location but at a different timestep, representing a "wait" move.

The intention for this test was to see how well the current solution for single agent path finding could transfer to the multi agent version, and to build intuition in what was next needed to be implemented. This approach was able to find collision free paths for early instances but would grow the search space exponentially in more challenging instances. The search space for TXA* grows with the number of timesteps and is a theoretically unbounded algorithm, meaning TXA* may never terminate. As the Flatland challenge features instances with up to 150 agents, a more efficient solution would be required.

Prioritised Planning with Safe Interval Path Planning

The chosen approach to tackle sequential multi agent path finding is Safe Interval Path Planning [Phillips and Likhachev 2011], which is an extension on A* in which the notion of safe intervals is implemented. Safe intervals are contiguous periods of time where a given location on a grid does not contain an obstacle. The key benefit to using SIPP is that when these safe intervals are used to index the agent states instead of timesteps, the search space can be significantly reduced. Additionally, SIPP nodes implicitly wait in place, thus removing the need to model "wait" actions explicitly like in TXA* and reducing the search space further. Figure 2 below compares the runtimes of the implemented TXA* and SIPP across *levels 0 to 5* and demonstrates that SIPP plans agents extremely quickly compared to TXA*.

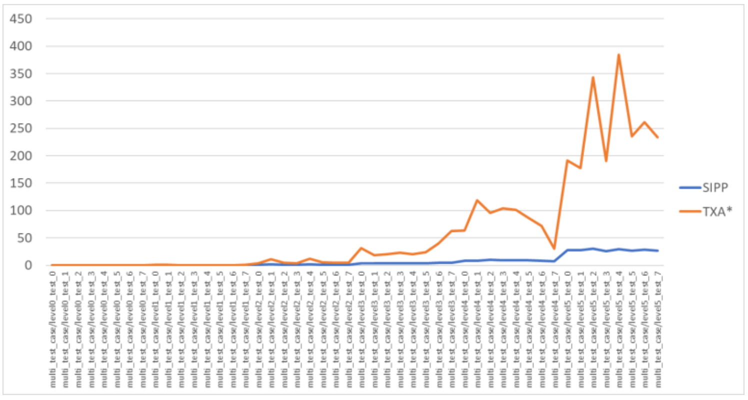

Figure 2 – Plan time (s) comparison of TXA and SIPP*

The use of the safe interval concept and the mechanic of moving on as early as possible allows Safe Interval Path Planning to have a strong advantage over other time-based A* alternatives. This is because the full dimension of time reduces to contiguous time intervals where an agent is not in collision. The number of these safe intervals is much smaller than the number of underlying timesteps. For example, a small timescale of 0…10 with an obstacle at timestep 5 is reduced to just two intervals, [0,4] and [6,10]. Thus, we only search over these two intervals, instead of all of 0…10 timesteps. In fact, we are guaranteed by Theorem 3 [Phillips and Likhachev 2011] that there can be at most n+1 safe intervals, where n is the number of obstacles that pass-through a given location. This is the mechanism which allows SIPP to create plans efficiently.

```
     Algorithm 1: A* with safe intervals
1    g(s_start) = 0, OPEN = ∅;
2    insert s_start into OPEN with f(s_start) = h(s_start);
3    while s_goal is not expanded
4    remove s with the smallest f-value from OPEN;
5      successors = getSuccessors(s);
6      foreach s' in successors do
7        if s' was not visited before then
8            f(s') = g(s') = ∞;
9        if g(s') > g(s) + c(s, s') then
10           g(s') = g(s) + c(s, s');
11           updateTime(s');
12           f(s') = g(s') + h(s');
13           insert s' into OPEN with f(s');
```

Algorithm 1 - A with safe intervals*

```
     Algorithm 2: getSuccessors(s)
1    successors = ∅;
2    foreach action in valid transitions do
3      cfg = configuration of action applied to s
4      m_time = time to execute action
5      start_t = time(s) + m_time
6      end_t = time(s) + endTime(interval(s))
7      foreach safe interval in cfg do
8        if startTime(i) > end_t or endTime(interval(i)) < start_t: continue
9        t = max(start_t, startTime(i))
10       if edge or vertex collision with existing paths: continue
11       if t > min(end_t, endTime(interval(i)): continue
12       g = t – time(s)
13       s' = state of cfg with interval i and time t, with time to act cost g applied
14       insert s' into successors
15   return successors;
```

Algorithm 2 - getSuccessors for SIPP

Algorithms 1 and 2 describe the pseudocode for A* with safe intervals and the getSuccessors method for Safe Interval Path Planning, adapted from the original 2011 paper, with some minor changes such as to getSuccessors, where line 10 explicitly checks for collisions with existing paths, line 11 checks whether t is outside the upper bound of possible time, and line 12 states that the cost to perform an action is the difference between the time t and the time of the current node s.

Safe Interval Path Planning and Prioritised Planning in general do have disadvantages when compared to optimal multi agent path finding algorithms. While PP is simple and fast, as only one agent is planned at a time, the strategy is not optimal nor is it complete. This is due to the fact that not all problems are priority solvable, and of those that are solvable, there may not exist an optimal solution. Let us imagine two scenarios with two agents: one scenario is not priority solvable, and the other is solvable but is not optimal. We discuss the implications below:

1) In the first scenario, no ordering of Agent A or B results in a complete solution. This might occur in highly constrained environments. If Agent A is given priority, there is no solution for B. Likewise if B is given priority, there is no solution for A. In practise, this is rarely the case as there are generally many paths and options available to agents in order to traverse their environment.
2) In the second scenario, solutions may exist. However, any solution as a result of agent ordering is not optimal. In practise however we can accept some sub-optimality if there are other trade-offs such as search efficiency and scalability.

While there are some downsides to PP as a whole, in practical settings including the Flatland domain, more often than not solutions are near optimal and plenty of solutions are available, which makes it a good algorithm specifically when trains are planned one after the other like in question 2.

Improving the safe interval data structure

Safe intervals are determined by iterating through all possible collision paths and determining the timesteps at which collisions happen. Naively, this can be pre-computed given all collision paths. However, in prioritised planning, agents are planned one at a time. As more agents are planned, if this is pre-computed each time, then the complexity of this grows from order of number of agents $O(P_a)$ to $O(P)$, where P_a is the path of the current iteration agent *a*, and *P* is the set of all agents paths. This is because by the time the last agent is planned, you will need to determine the safe intervals using *P-1* agents' paths. The longer these paths, the more work needs to be done each iteration. This was the safe interval table implementation used in evaluations 2 and 3 at the end of this report, and for a long time resulted in heavily suboptimal performance where only instances up to level 5 test 6 could be solved within the allotted 2 hours.

A modified version of the *grid_reservation_table* is utilised to efficiently store and update safe intervals. An issue which was encountered was that precomputing the safe interval table every time using the full list of existing paths resulted in large performance drops in the algorithm. This would result in SIPP timing out by level 5, as computing the safe intervals was quite costly in one shot.

Utilising the data structure of the reservation table defined in piglet, only the latest existing path is utilised to update the global interval table. This meant the operations performed each time an agent is planned is just an update of the safe interval table using the latest planned path. This improvement in the safe interval data structure allowed for SIPP to solve all instances in under 30 minutes, a big improvement from the previous inefficient implementation.

Maintaining well-formed instances

As this question is designed to plan trains in a fixed ordering, there may be situations where planned obstacles pass through the start position of latter planned trains. This then results in no possible paths for the latter trains.

In the below example, train 11 is the final train to be planned. Its plan is to first wait in position until all trains clear. However, train 7 has been given the higher priority in planning and has planned to enter the dead end which train 11 inhabits as its start location. There are no paths for train 11 to take up to this point as other trains may collide with it, thus there is no solution for this instance given all previously planned paths.

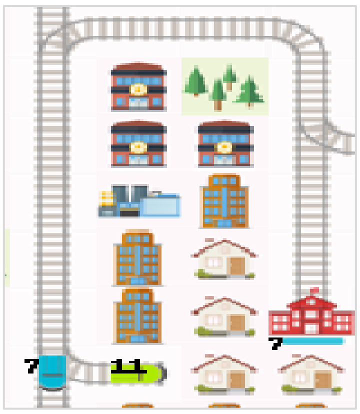

Figure 3 - Train 11 is in collision with Train 7 and has no way to avoid it as previous existing paths prevent train 11 from moving.

This situation can be checked by finding the last timestep that a train stays in the start position, and then checking at that timestep if any already planned trains pass through that position. If that is the case, then we simply return no plan for the agent which waits in its start position only to have another train collide with it. For the above case, we return an empty plan for train 11.

Improving the search heuristic

While manhattan distance is a simple heuristic to calculate and is admissible, it may not performative enough when it comes to solving problems requiring finding paths for larger numbers of agents. A more powerful heuristic can be chosen in order to speed up search and potentially solve more problems.

Two memory-based heuristics alternatives were tested, one being the differential heuristic which was precomputed using a domain specific Dijkstra search which accounted for train directions as the actions to take. The other was to simply pre-compute the true distance from the target to every other location on the grid map and utilise the true cost as the heuristic.

Differential heuristic

Differential heuristics are part of a family of memory-based heuristics called true distance heuristics [Sturtevant et al. 2009] and make use of the triangle inequality, which specifies that if we know the true distance between some pivot location and the start location, and also the distance between the pivot and the target location, then we know that the absolute difference between these pivot distances is bounded by the true distance from start to target.

$$|\mathrm{d}(\mathrm{s},\mathrm{p}) - \mathrm{d}(\mathrm{t},\mathrm{p})| \leq \mathrm{d}(\mathrm{s},\mathrm{t})$$

Differential heuristics are thus admissible as by the definition of the triangle inequality, they never underestimate the true cost. They are also very practical as they scale well compared to a pairwise oracle which stores the true distance between all points on a grid. The high-level complexity of a differential heuristic is of the order of number of pivots, as opposed to the number of agents in a pairwise oracle heuristic. This allows the differential heuristic to scale well to larger numbers of agents, as you only ever need to run Dijkstra for the number of pivots you have.

One of the nuances of the Dijkstra search implementation is that a GridTransitionMap requires a direction to generate successors. As such the Dijkstra open list is populated with multiple start nodes, as the state of a Flatland node includes the direction as the action. This enables the Dijkstra search to always have an available action to generate successors using the GridTransitionMap.

```
Algorithm 3: selectRandomPivots(n)
pivots = ∅;
while len(pivots) < n
    x ~ Uniform(0, map_height); y ~ Uniform(0, map_width);
    if (x, y) is a valid location on the map then
            insert (x, y) into pivots
return pivots;
```

Algorithm 3 - Select pivots randomly

```
Algorithm 4: selectPivotsByDistance(n)
pivots = ∅;
start_pivot = selectRandomPivots(n=1)
insert start_pivot into pivots
while len(pivots) < n
    forall x, y in 0..map_height, 0..map_width
            best = valid (x, y) with maximum Manhattan Distance with all p in pivots
            insert best into pivots
return pivots;
```

Algorithm 4 - Select pivots by maximum distance

Algorithms 3 and 4 detail the simplified pseudocode for finding pivots either randomly or by finding points which maximise the manhattan distance between the current (x, y) and all the currently chosen pivots. Algorithm 4 reduces to random pivot selection when n=1, as the initial pivot must be chosen randomly to then have a point of reference for the remaining distance maximisation.

The below compares the efficiency and solution quality of the two pivot methods below. The number of pivots used are 5 and 15 for each to examine the effects. The result is that there is almost no difference in makespan between the methods, though with some random fluctuations on an instance-to-instance basis. An interesting result is that runtime for the max distance method is faster than randomly choosing points for planning time, regardless of number of pivots. This could simple be an artefact of the search process or might hint at a potential underlying inefficiency in the random sampling itself. Regardless, the max distance method with 15 pivots was chosen as there was no significant or practical difference between that and any other configuration of the heuristic.

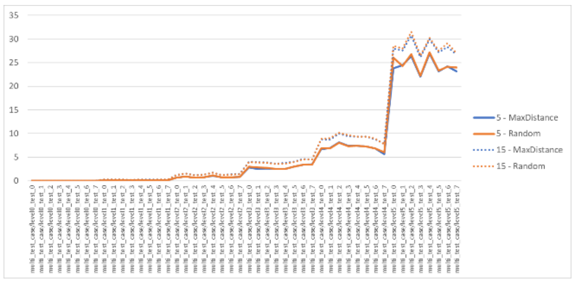

Figure 4 – Pivot selection method comparison by runtime (seconds)

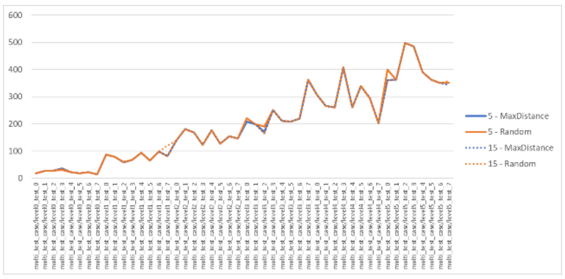

Figure 5 – Pivot selection method comparison by makespan

Pairwise Dijkstra heuristic

Another heuristic trialled was the pairwise oracle heuristic, which runs all start and goal nodes and finds their costs at any point in time. This is more trivial than the differential heuristic, in that it computes the true distance from the target to all other locations on the grid. The benefit is that we can use the true cost as a guide for the agent, which is guaranteed to be admissible as it is simply the true cost.

Pairwise oracles don't scale as well as differential heuristics, as it requires a Dijkstra search be performed for every agent, as opposed to every pivot. For larger instances, this can increase the runtime. During initial experimentation on local instances, there was no real difference in solution quality as compared to the differential heuristic with 15 pivots, and thus a differential heuristic was chosen to be used as the final heuristic.

Dijkstra on disconnected track segments

One caveat to running Dijkstra's algorithm is that in disconnected grid maps some paths do not exist and cannot be calculated. This may result in the differential heuristics failing as when the cost from a pivot to a node is looked up, depending on the location there may not be an existing solution. In the below figure, there are two groups of tracks: one for agent 4 and another set for the remaining agents. If a pivot is located on the tracks for agent 4, then all possible moves explored using Dijkstra's algorithm will be restricted to those track segments.

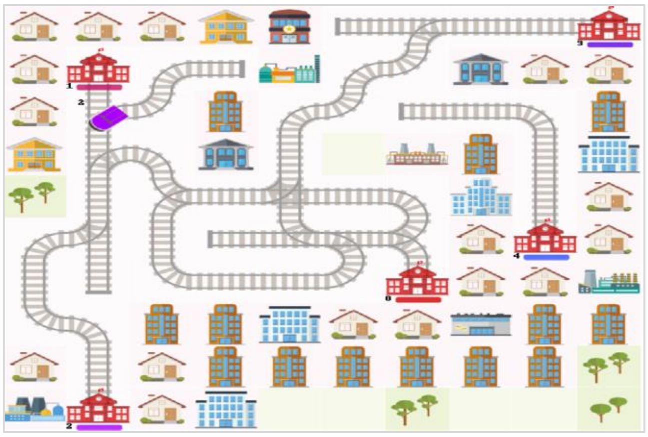

Figure 6 - Agent 4 disconnected track in Level 0 Test 7

In these circumstances, the simple answer is to relax the heuristic further. This implementation of the differential heuristic relaxes and reverts to manhattan distance as that is an admissible heuristic that is quickly calculated.

Weighted heuristics

In addition to utilising a more powerful search heuristic, using different heuristic weightings was experimented with. A weight w was applied to the cost function such that for weights $w > 1$, the function $f(n) = g(n) + w * h(n)$. Weighted heuristics have the following guarantee:

$$\text{f}(n) = g(n) + w \times h(n) \text{ where } w \geq 1 \text{ and } h \text{ is admissible.}$$

$$\text{Then any solution } C \text{ will have a bounded cost}$$

$$C \leq w \times C^{*} \; where \; C^{*} \text{ is the true cost.}$$

In some tests, there was no real difference in planning time and path solution qualities, even when using very high weightings. For example, using *level 5 test 0* as the use case, there was virtually no improvement in planning time and no difference in SIC or makespan when using $w = 1, w = 1.1, w = 1.2$ or all the way up to $w = 10$. This was regardless of which heuristic function was used. In the end the decision to weight the heuristic function was abandoned as there was no significant performance gain in exchange for possible inadmissibility.

The Flatland Challenge

The third question gives full control of all agents from the start of the Flatland simulation. In addition, trains may malfunction and require to be replanned. While some techniques such as Conflict Based Search is solution optimal and complete, it generally does not scale well to larger number of instances. Following the winning of the NeurIPS 2020 Flatland Challenge, the winning solution [Li et al. 2021a] demonstrated that utilising SIPP along with several enhancements such as conducting Large Neighbourhood Search to replan agents was a viable strategy for routing large numbers of trains with great solution quality.

The benefits to using this approach for the assignment are that it can find near-optimal solutions eventually, it can scale well to a larger number of agents, and lastly it builds on the previous questions implementation of SIPP such that work can be reused efficiently. In addition, the partial replanning mechanism mentioned in the paper is implemented, whereby any trains that malfunction are replanned along with any trains that intersects with any segment of the path that the train was going to visit.

Ordering agents for Prioritised Planning

Determining the order of trains is quite important for prioritised planning. Ideally, we want to minimise the delays and path costs of each train. Several simple approaches were trialled to find an initial ordering for the trains.

1) Order trains by ascending manhattan distance between start and end locations. This is simple and efficient, however can result in orderings where geographically close start and end nodes may be ordered first, but true path cost is much larger.
2) Order trains by ascending true distance from start to end locations. This represents the "earliest arrival time". This is achieved by running an optimal complete algorithm such as A* for each start and end goal pair, and ordering agents by true cost. This method relies on the fact that A* is optimally efficient, meaning there is no algorithm less informed that can possibly expand fewer nodes, making it the best solution for finding the true cost in single agent path finding.
3) Order trains by each train's deadline. We can prioritise getting trains with short deadlines to their destination first, before planning trains that have more lenient plans. The benefit of this approach is that we have readily available information for each agent that we can use, without needing to calculate anything.

In the paper on winning the 2020 Flatland Challenge [Li et al. 2021a], initial orderings by earliest arrival time were empirically found to produce better results but that did not dominate other ordering choices. This meant that ordering agents naively by their IDs (the default of question 2) produced at times the best solutions, while other orderings without further meta heuristic search produced worse solutions. The simplest example in the assignment is that of *level 0 test 0*, where the ordering of agents by IDs 0 through 4 gave an SIC of 60, while ordering by the earliest arrival time produced an SIC of 64.

There are also more sophisticated methods for agent ordering in the literature surrounding PP, one of which being Priority-Based Search [Ma et. al 2019]. At a high level, a depth-first search is performed to construct a priority tree, which greedily chooses agents to have a higher priority, and backtracks when a collision occurs in order to incrementally find the best solution. PBS has been shown to outperform many other prioritised algorithms and scaled to 600 agents without reaching the runtime limit of one minute. This approach was not chosen due to time constraints and is an avenue for further exploration.

In the end, the final chosen approach was to simply order the trains by their agent IDs, as it generally found the best solutions when a meta-heuristic was not used. As found in the 2020 Flatland Challenge winning paper implementation, any feasible ordering is acceptable and does not have any unique benefit empirically compared to others. As there is access to only one thread on the contest server, the option to run Parallel LNS is not available and thus any feasible ordering should be sufficient.

Replanning malfunctioning trains

In terms of replanning, the first approach is to simply delay all trains on the map by the malfunction time of the current malfunctioned train. This ensures the ordering of trains is preserved, however causes highly unnecessary delays, and can result in the instance timing out as there is a 1/10 chance that a train may experience a malfunction on instances where malfunctions occur.

An improvement over the naïve replanning approach is what is called Minimum Communications Policy, which stops a selection of trains affected by the malfunction. This can be an effective method; however, it can result in situations where trains are stopped unnecessarily when they might actually be able to move freely.

Partial replanning aims to address this, where the goal is to collate all trains which will visit any intersection that the malfunctioned train would have already visited and replan all those trains one after the other. The benefit of this approach is that only trains that are affected by the malfunctioned train need to be replanned, while the remaining trains can continue to follow their conflict free paths in the background. Two types of partial replanning can occur depending on implementation:

1. Collecting agents in a set to replan. As a set is unordered, and replanning trains from here will break the assumption of MCP that trains maintain their order.
2. Collecting agents into an ordered list to replan. This allows for ordering to be maintained by replanning trains in order of when they first intersect with a malfunctioned trains path.

When agents are replanned, the search begins from their current position rather than the initial position. The existing paths until the malfunction timestep is kept, and the replan continues from there.

The implementation of replanning in this report is not complete due to most development time being spent on addressing inefficiencies in SIPP, and thus the replan mechanism is not as sophisticated as it could be. Nevertheless, the pseudocode for the replan mechanism is provided in Algorithm 5.

Algorithm 5: replan(agents, malfunction_data, timestep)

```
1   malfunctioned_agents = agents that have malfunctioned according to malfunction_data
2   all_paths = $\{(\text{path}(a)) \mid \forall a \in agents\}$
3   paths_from_timestep = $\{p[timestep \ldots] \mid \forall p \in all\_paths\}$ # slice path from t onwards
4   malfunctioned_paths = ∀malfunctioned_agents (paths_from_timestep)
5   affected = {}
6   **for** m_path **in** malfunctioned_paths do
7       agent_id = 0
8       **for** path **in** paths_from_timestep \ malfunctioned_paths **do** #ignore malfunctioned paths
9           **if** path intersects m_path ∧ agent_id **not in** malfunctioned_agents **then**
```

```
10          insert agent_id into affected
11       agent_id += 1
12 for m_agent in malfunctioned_agents do
13    delayed_path = repeat (x, y) location path(m_agent)@t for length of malfunction
14    replace original with delayed_path in all_paths for m_agent
15 for agent in affected do
16    replan = call Algorithm 1 with affected_agent and all_paths \ path(agent) as collisions
17    replace original with replan in all_paths for agent
18 return all_paths
```

Algorithm 5 - Simple replan mechanism inspired by partial replanning

Large Neighbourhood Search

While Prioritised Planning can plan agents relatively quickly, the solution quality is not always great and there are cases where certain orderings lead to incomplete solutions. One meta-heuristic to address this in multi agent path finding is MAPF-LNS [Li. et al, 2021], the application of Large Neighbourhood Search [Shaw 1998] to multi agent path finding. MAPF-LNS is a framework that can be used with any MAPF algorithms, in this case being used with Prioritised Planning using SIPP.

MAPF-LNS has been shown to find near-optimal quality solutions for large numbers of agents over various benchmarks. This assignment implements most of the original MAPF-LNS and is tweaked to maximise the number of collision free paths. Part of the solution quality check is to see if the number of paths with collisions are reduced in each iteration. While this is not the complete MAPF-LNS2 approach to dealing with initially infeasible solutions, it is a relatively simple change to the underlying algorithm in order to produce at least better solutions than if MAPF was not used.

```
   Algorithm 6: randomLNS(agents, early_stop=5)
1  best_SIC = $\sum_{a\in agents}(|path(a)|)$ - |agents|;
2  best_penalty = $\sum_{a\in agents}(\max(0, |path(a)| - deadline(a)))$
3  best_score = best_SIC + best_penalty;
4  best_paths = {path(a) | $a \in agents$}
5  count = 0
6  while iterations < max_iterations do
7     if count > early_stop return best_paths;
8     n ~ Uniform(1, |agents|);
9     insert n agents into sampled_agents;
10    new_paths = {agents} \ {sampled_agents}; # destroy
11    for agent in sampled_agents do
12       path = call Algorithm 1 with agent and new_paths as collisions
13       insert path into new_paths;  # repair
14       if score(new_paths) < best_score then
15          best_score = new_score
16          best_paths = new_paths
17          count = 0
18       else count = count+1
19 return best_paths
```

Algorithm 6 - Large Neighbourhood Search with random agents

MAPF-LNS is an iterative algorithm that terminates after a maximum number of iterations. There were many instances where more complicated maps resulted in difficulties finding even an initial plan for the trains. As such, the decision to implement an early stopping strategy for LNS that scaled with grid size was used to empirically control the max number of iterations so that the largest number of problems could be solved.

[Li et al. 2021a] used a Simulated Annealing model to find the number of iterations to use for each instance. However, as the implementation of SIPP in this report was not finalised until a few days before submission, there was not enough time to run and tune a Simulated Annealing model. Therefore, the decision to derive the number of max iterations from a number of empirical observations was chosen. The equation for finding the maximum number of iterations is given by the below, where parameters were found empirically from observations during testing. This allows for large amounts of iterations to occur on smaller (and less complicated) instances, while less planning time can be devoted to larger instances as having a single iteration threshold for all instances can result in latter instances timing out.

$$max\_iterations = \left\lceil 250 \times \exp\left(-0.015(num\_agents + map\_width)\right)\right\rceil$$

The decision to use an early stopping approach was so that after some trials with no improvement, the neighbourhood search would stop, and the planner would take the best current plan to the evaluator. While using simulated annealing and maximising the usage of the runtime to find the best LNS solution would be ideal, this approach would be limited in effectiveness in this case as unfortunately the implementation of the path planner does not find paths for complicated instances as efficiently as it could.

Random neighbourhoods

Large Neighbourhood Search works by searching through “neighbourhoods”, which are used to identify groups of agents to have their initial solutions erased and recreated with the goal of finding better solutions over time. These neighbourhoods can be combined using ALNS, a technique which probabilistically chooses between different destroy strategies, which remove agents’ paths and replans them.

The simplest of these strategies is the random agent neighbourhood, which essentially selects agents at random, destroys their paths and then repairs them by replanning, hopefully decreasing SIC and improving the makespan. While random agent selection is not domain specific, it is intended to nonetheless provide an improvement in SIC over naïve prioritised planning.

Unfortunately, the implementation of SIPP in this paper was not greatly improved until only a few days before submission, and thus not enough time was available to fully implement domain specific neighbourhood solutions such as intersection-based neighbourhoods. However, in the original paper the random agent neighbourhood was shown to be surprisingly effective for congested instances.

Empirical Evaluation

All three answers to the assignment questions are evaluated across a suite of instances that increase in difficulty as the number of agents and the complexity of the environment increases. These instances were run locally using devices with 16GB and 32GB of RAM and were evaluated on the contest submission server which provides 4GB of memory and a single thread. For evaluations 2 and onwards, only evaluation results from the server are presented as that is where assignment marks are achieved and simplifies the communication of results.

Assignment questions 2 and 3 are evaluated against the instances in Table 1. Question 1 is tested across a reduced version of 40 instances, where only a single agent is planned in each instance.

Map (height x width)	***N agents***	***Max T***	***Number of Tests***
10 x 10	5	160	8
25 x 25	12	400	8
50 x 50	25	800	8
75 x 75	37	1200	8
100 x 100	50	1600	8
150 x 150	75	2400	8
150 x 150	150	2400	8

Table 1- Instances to be evaluated.

Evaluation 1: Single agent path finding

All 40 instances were able to be solved on both the server and locally. This was the expected result as A* with an admissible heuristic is optimal and complete, which means that it is guaranteed to find the optimal solution where one exists. Each instance was executed almost instantly, with the runtime statistic giving zero for each instance. As A* is an optimal and complete algorithm, in theory these results represent the true costs for each instance.

	Local	**Server**
SIC	1823	1681
P-score	45	42.025

Table 2 - A single path results*

Evaluation 2: Initial Prioritised Planning implementation

Prioritised planning with SIPP in the case of no collisions and using a manhattan heuristic resulted in fast solving for instances up to 12 agents, and then became very slow to solve with only the first instance with 37 agents being solved with in 868.46 before memory usage being exceeded on the server. This evaluation illustrated that the current implementation of SIPP required revisiting. What was discovered with this approach was that SIPP in Flatland only requires the (x, y) location, the Flatland direction, and the interval when it comes to state checking. As discussion in the earlier section on safe intervals, the full-time dimension is reduced to just the number of safe intervals, which cuts down the search space significantly. In this evaluation only 352 agents were able to be solved, with an F-score of 25.43.

Evaluation 3: Prioritised Planning with different heuristics

Improvements to the search heuristic and state checking allowed for 1529 agents to be solved instead of 352. This allowed for instances up to level 5 test 6 to be fully solved, though by that point the runtime would exceed 2 hours. The F-score for this section was 46.6 and a P sum of 26798, with early instances being solved relatively quickly with only instances past level 4 requiring more than a minute to solve.

As running select instances is easier on the local test cases, the runtime and quality of the manhattan distance, differential heuristic and pairwise oracle are evaluated over the first five levels to demonstrate the behaviour of the search with each one. This gives enough results to show the difference between the heuristics while being able to test multiple configurations without needing to wait a long time for the full test suite to run. The results show that as the problems get more difficult, the differential heuristic is slower than the manhattan distance heuristic as Dijkstra search takes longer to pre-compute costs on the larger instances.

However, what was a surprise was the solution quality and the number of paths found, as the manhattan distance heuristic was more consistent than the differential heuristic when it came to finding good quality and collision free paths.

What Figure 8 clearly shows is that there was an issue with the implementation of the differential heuristic, as it was set as a global variable that was not reset every time the evaluator called a new instance. This was resolved by implementing a simple reset switch, triggered in *get_path()* if the agent ID was 0 in question 2 (indicating a new instance) or at the start of the function in question 3 due to having full control of all agents from the beginning.

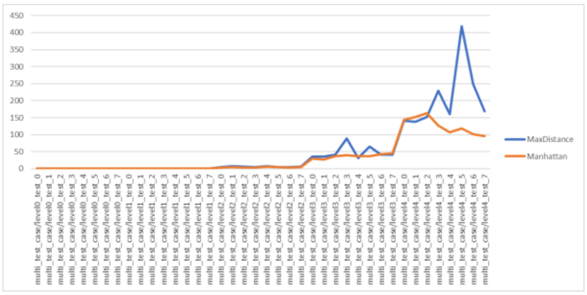

Figure 7 - Heuristic comparison on plan time (seconds)

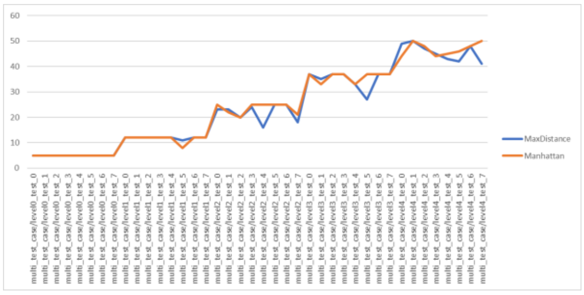

Figure 8 - Heuristic comparison on agents done

Evaluation 4: Prioritised Planning with improved safe interval data structure

Utilising a more efficient way of storing safe intervals by extending the *grid_reservation_table* to include safe intervals for every agent instead of pre-computing all intervals every search resulted in far superior efficiency and algorithmic performance. This enabled Prioritised Planning to run for all instances within 26 minutes, where before the solver would time out at 2 hours having only reached *level 5 test 6*.

The final result was 2784 out of 2832 agents had paths found, with a P-score of 7198 and an F-score of 53.195. The longest instance to solve was *level 6 test 5*, which was 92.48 seconds. In comparison with evaluation 3, the longest planning time for an instance was 811.19 seconds for level 5 test 5, a massive improvement in performance.

Figures 7 and 8 are redone in figures 9 and 10 below using the improved SIPP implementation, which clearly shows the reduction in runtime and improvement in agents done.

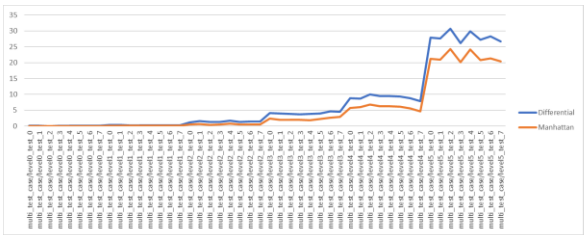

Figure 9 - Heuristic comparison on improved SIPP plan time (seconds)

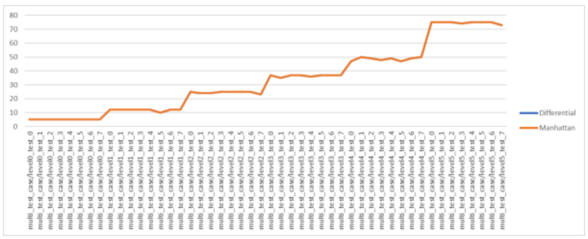

Figure 10 - Heuristic comparison on improved SIPP agents done

Interestingly, utilising the differential heuristic resulted in improved makespans for instances *level 1 test 7* and *level 3 test 2*. What was noticed was then when running these instances, certain agents would have much longer paths compared to others. For example in the first case (*level 1 test 7*), agent IDs 8 through 11 had paths of length 68, 73, 47 and 61 with the differential heuristic. With manhattan distance, these paths were much longer at 100, 116, 108 and 122. In the second case (*level 3 test 2*), the difference in path lengths was less evident, however singling out agent ID 31 with the differential heuristic gave a makespan of 166, where the manhattan heuristic gave 192. This would indicate that as more agents get planned, a stronger heuristic would be required to find shorter paths and improve makespan.

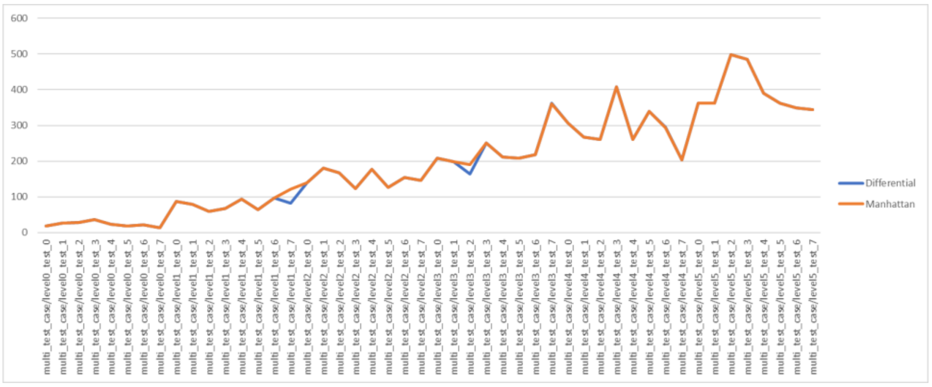

Figure 3 - Heuristic comparison on makespan

Evaluation 5: Flatland Challenge with Prioritised Planning

The implementation of Prioritised Planning from question 2 was utilised for question 3. This was with using agent ID as the ordering and no meta-heuristic such as LNS being applied. Replanning only involved delaying malfunctioned trains.

This resulted in 2321 agents having paths planned with 2012 DDLs being met, with an F-score of 25.75 being achieved and a P-sum of 18145. This was run in under 31 minutes, meaning that almost an hour and a half could be dedicated to improving the solution iteratively.

Evaluation 6: Flatland Challenge with random agent LNS

A simple variant of Large Neighbourhood Search was implemented and evaluated across the instances. This was to compare the effects of LNS compared to the base implementation. Replanning in this evaluation again only involved delaying malfunctioned trains.

While only 2160 agents were planned, 1985 DDLs were met, resulting in an increase from 86.7% to 91.2% of DDLs met over all agents planned. This would seem to indicate that with proper replanning and intersection-based neighbourhoods that this percentage of DDLs met could increase. However, the final F-score for this section was 16.89 and the P-score was 21492, which is worse compared to evaluation 5. This would indicate that the low hanging fruit would be to address replanning in order to boost the number of overall trains being planned.

Evaluation 7: Flatland Challenge with train replanning

The replan mechanism described in algorithm 5 is utilised. Somewhat expectedly this resulted in subpar results as this replan mechanism was not as sophisticated compared to other potential implementations. During testing, many deadlocks would occur even after trains were replanned, which would indicate that agent ordering at track intersections (cases 2-6 in Figure 1) would be critical to resolving many conflicts.

Only 1752 agents were able to be planned, with 1479 deadlines being met for a P sum of 35929 and an F-score of 22.331. What was noticed was that many instances took a long time to solve, such as tests 4 to 7 of level 6 taking anywhere from 391 to 914 seconds to solve, and in the end only finding a very small fraction of the paths. This indicated that the simple replan mechanism was not implemented correctly or was not effective to begin with, resulting in many deadlocks.

Evaluation	*Agents Done*	*% DDLs met*	*P-Sum*	*F-Score*
2	352	-	61296	25.43
3	1529	-	26798	46.6
4	**2784**	-	**7198**	**53.195**
5	**2321**	86.6%	**18145**	**25.75**
6	2160	**91.2%**	21492	16.89
7	1752	84.4%	35929	22.331

Table 3 - Summary results for Q2&Q3 MAPF evaluations

Conclusions

This report demonstrated that Prioritised Planning is a viable approach to solve practical multi agent path finding problems. Utilising more advanced implementations of time space A* such as SIPP that reduce the search space significantly to create a much more efficient planner for agents operating in environments with potentially hundreds of dynamic obstacles.

During the majority of time of this assignment, the implementation of SIPP was severely underperforming and not efficient until the final few days before submission when the computation of safe intervals was drastically improved. This meant that there was little time to investigate a large number of advanced ideas to improve the performance of the agents. While Prioritised Planning with meta-heuristic search such as LNS was utilised to great success in the 2020 Flatland Challenge by the winning team, that same success was unfortunately not able to be fully replicated for this assignment. The future inclusion of different neighbourhoods along with the implementation of ALNS would be beneficial to improving the solution.

However, what the results from the evaluations have been able to show are that these approaches can be viable for increasing the number of deadlines met. Given more effective and efficient implementations, the effectiveness of LNS as reported in the literature may be more fully realised.

Finally, handling train malfunctions through replanning would need to be improved in order to better leverage the efficiency of SIPP. This would involve devoting adequate time to implementing at least a Minimum Communications Policy strategy for handling malfunctions during replanning, and then improving it by using the partial replanning mechanism discussed in the earlier sections of the report.

References

[Erdmann and Lozano-Pérez 1987] Erdmann, M. A., and Lozano-Pérez, T. (1987). On multiple moving objects. Algorithmica 2:477–521.

[Li et al. 2021a] 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. Proceedings of the International Conference on Automated Planning and Scheduling, 31(1), 477-485.

[Li et al. 2021b] Li, J., Chen, Z., Harabor, D., Stuckey, P. J., & Koenig, S. (2021). Anytime Multi-Agent Path Finding via Large Neighborhood Search. In Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS '21). International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1581–1583.

[Ma et. al 2019] Ma, H., Harabor, D., J. Stuckey, P., Li, J., & Koenig, S. (2019). Searching with consistent prioritization for multi-agent path finding. In P. Van Hentenryck, & Z-H. Zhou (Eds.), *Proceedings of AAAI19-Thirty-Third AAAI conference on Artificial Intelligence* (pp. 7643-7650).

(Proceedings of the AAAI Conference on Artificial Intelligence; Vol. 33, No. 1). Association for the Advancement of Artificial Intelligence (AAAI).

[Phillips and Likhachev 2011] Phillips, M. & Likhachev, M. (2011). SIPP: Safe interval path planning for dynamic environments. Proceedings - IEEE International Conference on Robotics and Automation. 5628 - 5635. 10.1109/ICRA.2011.5980306.

[Shaw 1998] Shaw, P. Using constraint programming and local search methods to solve vehicle routing problems. In CP, pages 417–431, 1998.

[Silver, D. 2005] Cooperative Pathfinding. In AIIDE, 117–122.

[Sturtevant et al. 2009] Sturtevant, N.R., Felner, A., Barrer, M., Schaeffer, J. and Burch, N., (2009). Memory-based heuristics for explicit state spaces. In Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 609–614.