Railway Scheduling Using Reinforcement Learning

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Arpit Singh
(111601031)

under the guidance of

Dr. Chandra Shekar Lakshminarayanan

INDIAN INSTITUTE
OF TECHNOLOGY
PALAKKAD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Contents

List of Figures

Introduction

The aim is to work on an algorithm for scheduling bidirectional railway network (both single- and multi-track) using a framework of Reinforcement Learning. Given deterministic arrival/departure times for all the trains on the lines, their initial positions, priority and halt times, traversal times, deciding on track allocations is a job shop scheduling problem (NP Complete). However, due to the stochastic nature of the delays, the track allocation decisions have to be made in a dynamic manner, while minimising the total priority-weighted delay. This makes the underlying problem one of decision making in of stochastic event driven systems.

First we will focus on solving scheduling problem for railway line and discuss different algorithm, there advantages, results. Then using the knowledge developed so far, we will solve the flatland environment[7] that is grid based simulator for multiagent reinforcement learning for any re-scheduling problem (RSP).

This report is organised in 7 main chapters. Chapter 2 presents the problem statement of railway scheduling on railway line, Chapter 3 provides details for the implementation of the simulator, Chapter 4 discusses the algorithm details for railway scheduling on railway line and Chapter 5 shows the experiments and results of the algorithms. Chapter 6 starts discussing the flatland problem, Chapter 7 discusses the Double deep q-learning based approach for flatland problem and presenting the results and Chapter 8 presents cooperative path finding approach for solving flatland environment and also presents the results. Last chapter presents the conclusion.

Problem Statement

2.1 Indian Railways

Let us first describe the nature of the railway system in this country. The Indian railway network is designed to consist of long 'lines' (a string of stations), which connect with each other at 'junction' stations. Each station is composed of one or more parallel **tracks**, which may be associated with a fixed direction of traffic, or they could be bidirectional. Similarly, there are one or more tracks between each neighbouring pair of stations. These tracks are typically referred to as **sections**, in order to differentiate them from tracks actually at a station. The section tracks can also be unidirectional (fixed direction of train movement) or bidirectional. The Indian network typically consists of sections with one or two tracks.

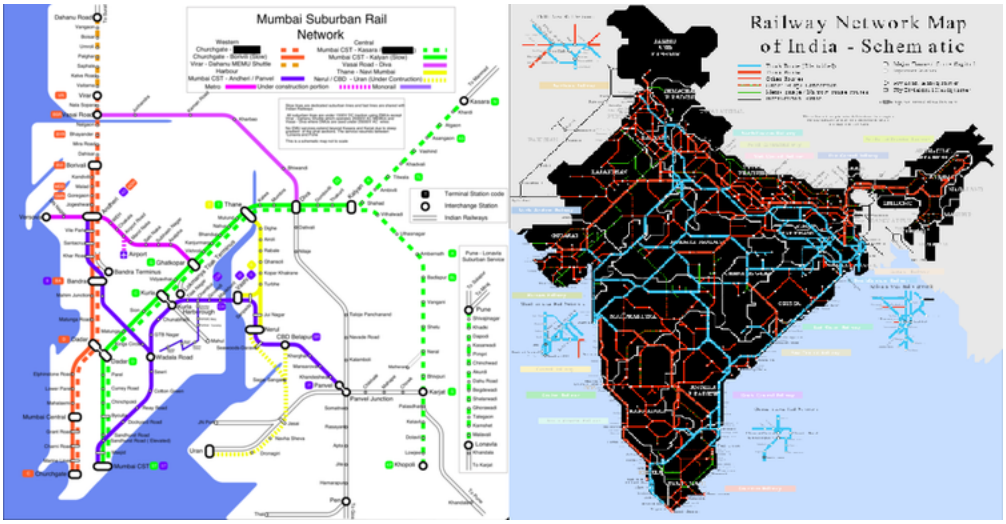

Fig. 2.1 The line and junction topology of railway networks in India [1].

The approach we are focussing on now deals with linear railway networks with multiple parallel tracks, of the type shown in figure 2.2. This restriction on topology is reasonable because rail networks are designed in the form of multi-station linear arcs connected at junction stations.

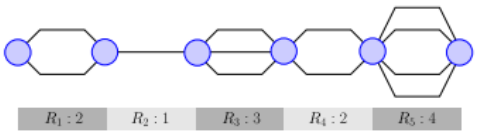

Fig. 2.2 Linear Railway Lines [2].

2.2 Railway Scheduling Problem

2.2.1 Scheduling

A specific problem instance begins by defining the resources on the railway line, as given by the number of stations, their order, and the number of parallel tracks (both at each station and between two neighbouring stations). Besides resource level information, train movements over the scheduling horizon must be described in one of two ways.

- To define a reference timetable which gives the desired arrival and departure time of each train at each station.

- To provide the earliest movement times from their current locations (or origin stations), followed by the minimum running times (on track sections between stations) and halt times (at stations) up to the destinations.

Note that the running and halt times can be completely heterogeneous: each train may have a different running/halt time in each resource, depending on the length of track, the type of halt, and the type of locomotive.

Timetabling refers to an offline planning problem for a railway network. **Given a set of trains and their origins and destinations (with or without a fixed route), the goal is to assign track resources for each train for a fixed time period, such that they all complete their journeys without conflicts.**

Such a timetable may be infeasible if the desired arrival and departure times violate the track usage rules defined earlier. The task of the scheduling algorithm is to adjust the arrival and departure times such that all rules are satisfied, while minimizing an objective called priority-weighted delay. This schedule is to be computed for all trains up to their destinations.

The railway problem has been shown in literature to be a **'blocking' version of the Job Shop Scheduling Problem (JSSP)**, where the job (train) must wait in the current resource (track) until the next resource is freed (there is no buffer for storing jobs between two resources). This version of the JSSP is also **NP complete**, with the result that exact solutions require an exponential amount of time for computation.

2.2.2 Rescheduling

Another problem is that of rescheduling. Rescheduling is the online counterpart of the timetabling problem , where the goal is to recover from a disruption to the timetable, caused by delays or faults. The mathematical differences are found in two aspects.

- The goal is to return to the original timetable using built-in slack times, instead of defining the timetable itself. This implies that the objective function would focus on minimizing delays to trains with respect to the timetable, or the time required for deviations to the smoothed out.
- The online nature of the problem implies that there is very limited time available to compute solutions, and that sub-optimal but reasonably efficient solutions are acceptable.

Simulator Implementation

The integrated reinforcement learning algorithm to derive the whole schedule is driven by a discrete event simulator. There are already some railway simulators like **OpenTrack** [8] and **RailML**[9] but they would be useful for the final analysis of the results. Once we have the desired timetable then we can use these simulator softwares to determine the quality of solution. But for the implementation of the algorithm we have to implement the simulator on own.

3.1 Railway simulator

3.1.1 Requirement

The simulator is supposed to be robust enough that it can run both toy and real life examples. The simulator is suppose to run through several episodes during training and hence need to be efficient. At the beginning of every episode, the initial locations of all the trains are reset to their original values. It is assumed that trains that have not yet started, or have finished their journeys, do not occupy any of the tracks. Following the train-to-resource mapping, the simulator creates a list of events for processing, one corresponding to each train (whether already running or yet to start its journey). Each event in the list contains the following information: the time at which to process the event, the train to which it corresponds, the resource where the train is currently located, the last observed state-action pair for the train (empty if the train is yet to start), and the direction of the train journey.

At each step, the algorithm moves the simulation clock to the earliest time stamp in the event list. If multiple events are to be processed at the same time stamp, they are handled sequentially. We are for now not focussing on how to avoid deadlock, but instead if we get into deadlock, we will detect and give huge negative reward and the RL algorithm is suppose to avoid deadlock on it's own.

3.1.2 Implementation

There are two components to railway simulator :

1. Underlying Railway Network.

2. Trains and the simulation of there movements.

For the implementation of the railway network we can use **NetworkX**[10] package of python. **NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.**

Once the network is ready we have to simulate movement of each train over the network. For that we can use **SimPy**[11] package of python. **SimPy is a process-based discrete-event simulation framework based on standard Python**. In this we can model each train as the separate process and network as the resource. We can model the movement of trains using this package. We yield events when the train starts from some station and once the train reaches the next station, event is yielded and then we can process accordingly. So whole simulation is done by generating events at points where the algorithm is supposed to take action.

Once the railway network is created, train class is used to create different instances of the trains running over the network.

3.2 Implementation Details

This section focuses on the implementation details of the railway simulator. Whole implementation of railway simulator is captured by the flow diagram below. It consists of different modules, which in turn are made of different components. First we will have a overview of each of the modules and its components and then move on to study them in detail. Whole code base is in repository [5]

3.2.1 Modules and its components

1. **Railway network and trains**

   This is the main module that contains the basic architecture of railway network and the trains. Railway network consists of two basic components :-

   - **Stations** : Stations in the railway network (synonymous to nodes in the network).
   - **Tracks** : Tracks connecting the stations (synonymous to edges in the network).

   Once the railway network is created, train class (implemented as component) is used to create different instances of the trains running over the network.

2. **Statistical Analysis and Graphs**

   This module is responsible for creating the logs and generating necessary graphs for visualization and analysis.

3. **Resource Usage**

   Whole railway network (station and tracks connecting the stations) is treated as pool of resource. Trains are ones that use this resource as they reside either on the station or the track. There are only fixed number of trains that can reside on the station and the track at a time. It is also possible for the train to wait for a resource to free up, as it is occupied by other trains. **Resource Usage Graph** component is responsible for generating the graph which show what all resources are occupied by the trains and for what all resources train is waiting. This can be useful in detecting deadlock.

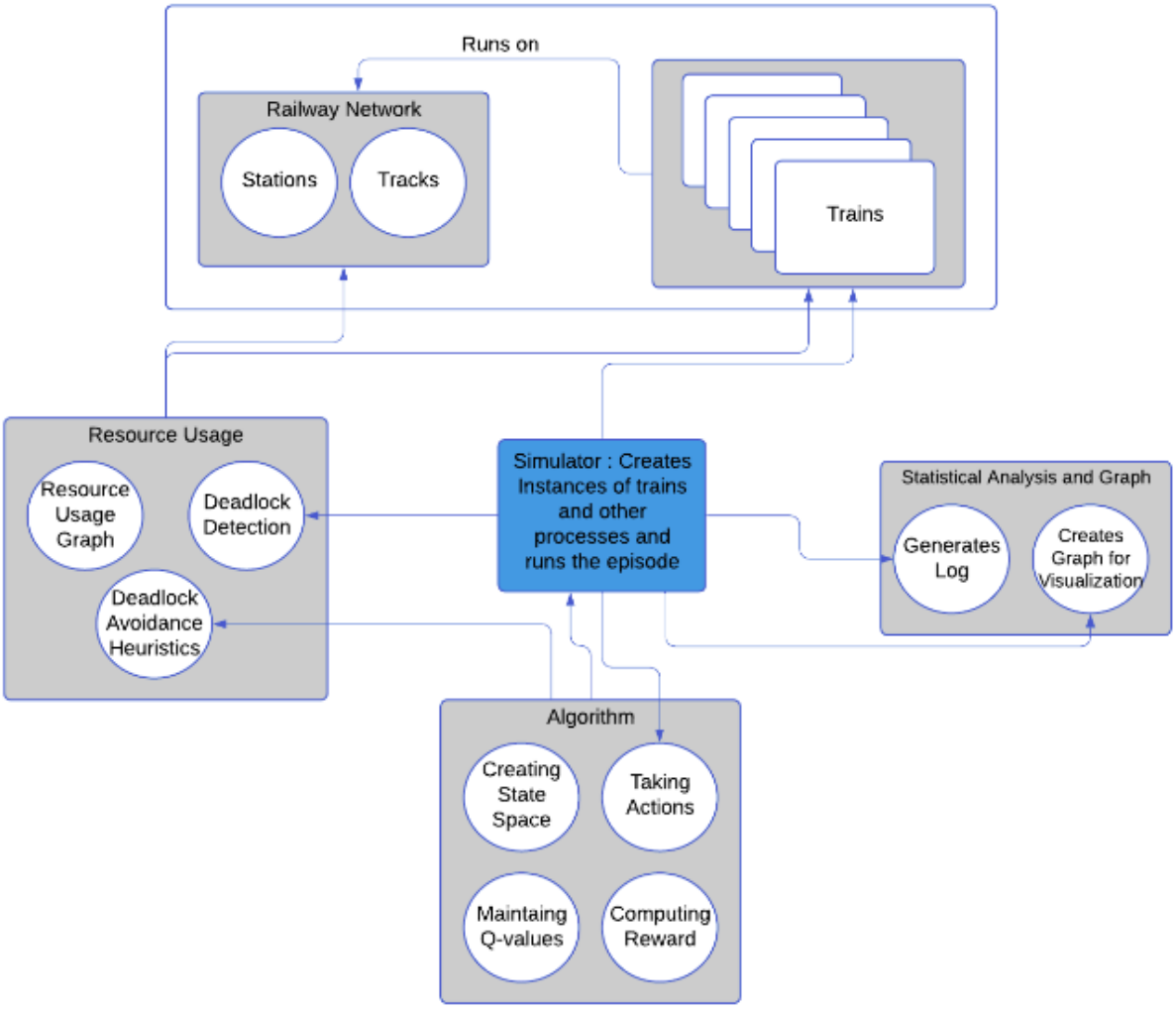

Fig. 3.1 Flow chart summarizing the implementation of railway simulator.

There is another problem of deadlock which the simulator can run in. We are going to discuss this problem in full length in the later sections. **Deadlock detection** and **deadlock avoidance** components are used for detecting and avoiding the deadlock in the systems.

4. Algorithm

This module implements the algorithm (Q-Learning or Deep Q-learning) that helps in learning the schedule. Currently only two components are implemented, creating state space and choosing action based on heuristic in [12]. More algorithms will be added in the future.

5. **Simulator**

   This is one of the most important module that is responsible for creating all the processes that runs the simulator. It creates the environment and then invokes all the processes, then it handles further processing. Because of this module, we can add more modules in the future to the existing system.

3.3 Rail Network and Train

This module is responsible for creating the underlying railway network and the trains that run on these networks. Railway network module is implemented in **network.py** and train component is implemented in **train.py**.

3.3.1 Mathematical model

This mathematical model is from [12]. The railway network is modelled as a graph $\mathcal{G}(\mathcal{N}, \mathcal{E})$ where $\mathcal{N}$ denotes the set of all nodes, and $\mathcal{E}$ denotes the set of all edges. A set of vehicles $\mathcal{V}$ is to be scheduled through this network, which implies that vehicles $v_i \in \mathcal{V}$ must be allotted time slots at successive nodes and edges, such that they can move from their respective origins to destinations via predefined routes (sequence of nodes). Each pair of nodes is connected by at most one edge, and thus routes also define the sequence of edges to be traversed. Each node $n_j \in \mathcal{N}$ and edge $e_k \in \mathcal{E}$ is assumed to be composed of one or more parallel (equivalent) resources, denoted by r_m^{nj} and r_p^{ek} respectively, where $m \in \{1, \dots R_j^n\}$ and $p \in \{1, \dots R_k^e\}$.

Let us define the arrival time of a vehicle v_i at node n_j by $t_i^a(n_j)$, and its departure time to be $t_i^d(n_j)$. Complementarily, the arrival time to and departure time from an edge e_k is denoted by $t_i^a(e_k)$ and $t_i^d(e_k)$ respectively. If e_k is traversed upon leaving n_j , then $t_i^a(e_k) = t_i^d(n_j)$. If the next node after e_k is $n_j^{'}$, then $t_i^d(e_k) = t_i^a(n_j^{'})$. For simplicity, it is assumed that all parallel resources at a node are accessible from all resources at adjoining edges. Finally, we define the binary variables $b_m^{nj}(i)$ and $b_p^{ek}(i)$ to be equal to 1 if v_i is allocated to resources r_m^{nj} and r_p^{ek} at respective nodes or edges, and 0 otherwise. Each vehicle v_i has an earliest start time on its journey (arrival time at first node) given by T_i , and its computed finishing time (departure from last node) is denoted by f_i . Its minimum halt time at node n_j is given by $H_i(n_j)$, and minimum travel time on edge e_k is given by $W_i(e_k)$.

Time constraints,

$$t_i^d(n_j) - t_i^a(n_j) \geq H_i(n_j)$$

$$t_i^d(e_k) - t_i^a(e_k) \geq W_i(e_k)$$

Resource Contraints,

$$\sum_m b_m^{nj}(i) = 1$$

$$\sum_m b_m^{ek}(i) = 1$$

3.3.2 Railway Network

Railway network consists of two building blocks **Stations** and **Tracks** that connect stations. All the fields of tracks and stations are given in the diagram below. Railway network is a weighted networkx graph where nodes are stations (Station class is added as attribute to the node) and edges are tracks running between stations (Track class is added as attribute to the edge). Input is given using two separate text files, one corresponding to tracks and other corresponding to stations. More detailed info is in repository.

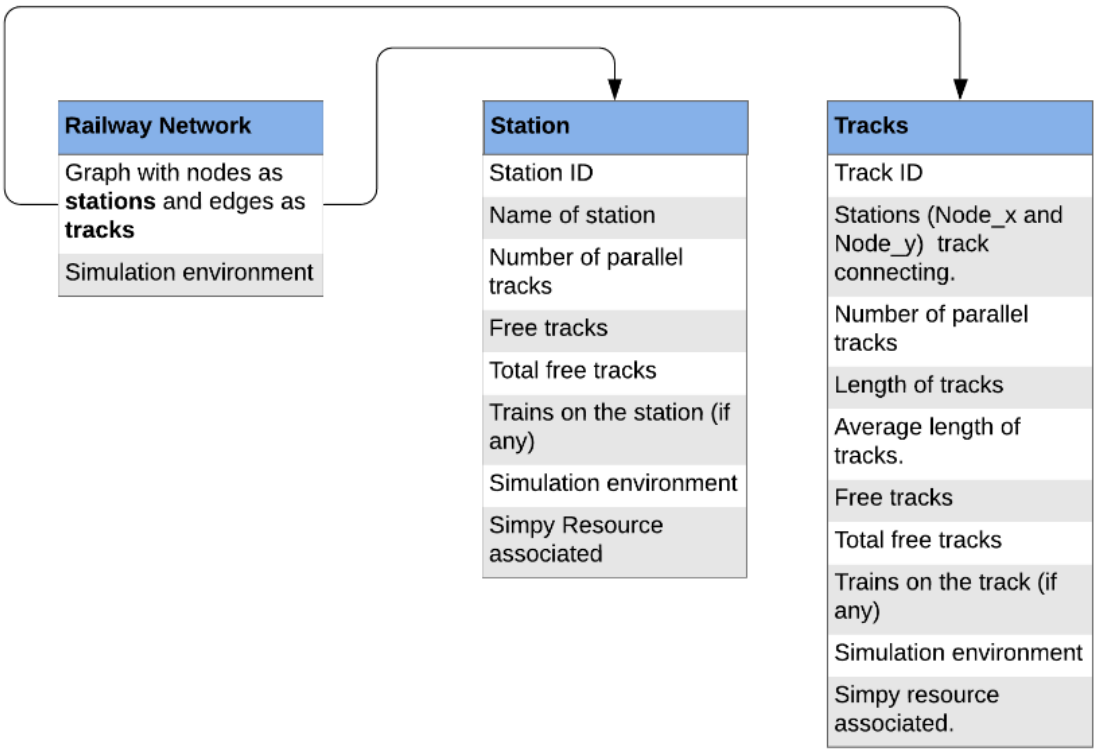

Fig. 3.2 Railway Network framework

3.3.3 Trains

There are multiple trains running in the network at a time. Train class defines all the variables and methods of this class and simulator uses this class to create process corresponding to each train (more details in simulator section).

Train movements over the scheduling horizon must be described in one of two ways. The first option is to define a reference timetable which gives the desired arrival and departure time of each train at each station. The second option is to provide the earliest movement times from their current locations (or origin stations), followed by the minimum running times (on track sections between stations) and halt times (at stations) up to the destinations. Note

that the running and halt times can be completely heterogeneous: each train may have a different running/halt time in each resource, depending on the length of track, the type of halt, and the type of locomotive. The timetable can be derived by adding the running and halt times of each train to the current time.

Trains

Variables
Train ID
Train name
Average Speed
Priority
Route
Simulation environment
Railway network
Status of train (multiple)
Resource currently using
Waiting for resource (if any)
Request (corrsponding to current resource)
Log of journey
Log file for log creation

Methods
Compute time
For log creation
Putting train in network
Wait for 1 unit time
Act (depending on action, move or wait)
Put train on first station
Move train one step
Finish journey
Check action validation
Status of train
Print Details

Fig. 3.3 Train Variables and methods

Variables in the train class are self understood. Following are the explanation and implementation details of the method. For more details look into the repository where explicit documentation is given.

1. **Compute Time** : To calculate the travel time of the train between two stations.
2. **Log Creation** : To create log corresponding to important events for each train.
3. **Put train in network** : This method puts the train in the network and generates an event corresponding to start time of the train. After start time, train movement is controlled using **Act** method.
4. **Wait** : To make train wait for predefined unit of time.
5. **Act** : This method takes one argument, either to move the train or wait. Depending on the argument, specified action is taken.
6. **Put train on first station** : This method tries to put the train onto first station and thus initiating the journey of the train. It may be possible, that the station is not free, in which case the move is invalid or it waits till the resource is freed.

7. **Move train one step** : Move the train one step. If the train is on the station, then it will try to depart to the next track or if the train is on track, then it will try to arrive at the next station.

8. **Finish journey** : This method is used when the train is at the last station and it has to free the last resource.

9. **Check action validation** : Check wether the given action (move or wait) is valid for the current status of the train.

10. **Status of train** : To give the current status of the train.

11. **Print details** : To print the details of the train.

Trains need action only for the following events :

1. If a train is standing at a station, the event processing time corresponds to the earliest time at which the train can depart, as defined by its minimum halt time at the station and by any departure time constraints enforced for passenger convenience.

2. If it is running between two stations, the event processing time corresponds to the earliest time at which it can arrive at the next station, as defined by the length of the track and the train running speed.

3. If the train is yet to start, the event processing time is the time at which it is expected at the starting station. This event is created by *put train in network* method. Once this event is generated, then *put train on first station* is used to put the train on the first station.

Once the train process is running, it is in one of the following states :

1. Train is not yet started.

2. Train is running in the network.

3. Train has reached the destination(final station in journey) but the resource is not yet freed.

4. Train has completed journey and released all the resources.

This status is used by create statistics process, that terminates the simulation if all the trains have completed it's journey.

3.4 Statistical Analysis and Graph

This module is responsible for creating the statistics and Graphs for visulalization and analysis. This module proves to be very important for debugging. Currently only two components are implemented but more can be implemented in future as per the need.

3.4.1 Log generatotion

This component generates all the log in the system. There are two sets of log. One corresponding to each train that gives info about the status of the train at different times during the simulation. Another generates log corresponding to the status of the network, how many trains are there in the network and wether the network is in deadlock or not. All logs are generated and put in the folder **Log**.

3.4.2 Graphs for visulalization

This component creates graph for visualization, while the simulation is running. Amount of detail we want in the graph can be controlled using different arguments. Note that this component slows down the simulation, so when the learning algorithm is running we can turn off this component.

3.4.3 Details of the image

Each station is represented by node and each track is represented by edge connecting these nodes. There can be more than one railway line on station or track, so the width of the nodes or the edges is directly propotional to the number of railway lines on that resource. How many lines are free on a given resource, that is encoded using the labels (on nodes and edges) and light green color. Each train running in the network is color coded. If more than 8 trains are running in the network, then all the trains will have the same color (then image just shows the resource level information about each resource in the network). We can control the amount of detail in the network by passing different arguments. For more details, look into the code repository.

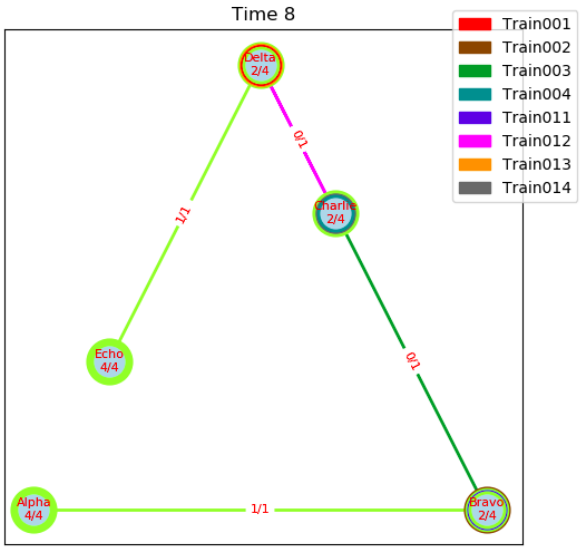

Fig. 3.4 Network with trains color coded.

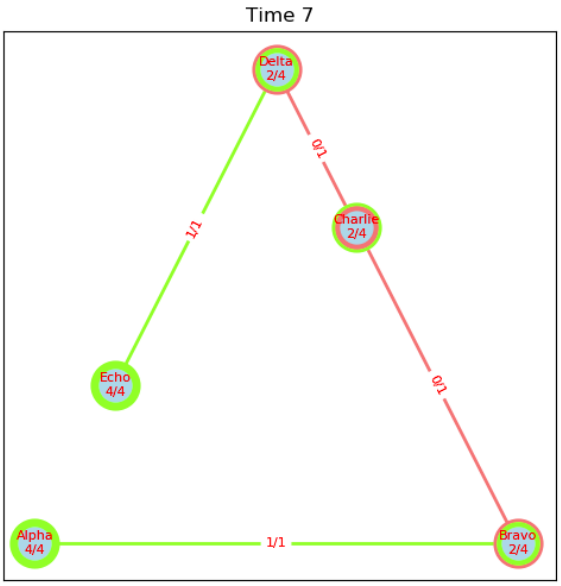

Fig. 3.5 Network showing just what all resources are free.

3.5 Resource Usage

This module monitors the resource usage in the network. This module is also responsible for detecting deadlock and implementing heuristic that avoids deadlock upto certain extent (more details in the following sections).

3.5.1 Resource usage graph

This component is responsible for creating graph, that shows which train is using which resource(track or station) and waiting for which resource(if any). Pink node corresponds to train, blue corresponds to stations and green corresponds to track. If a train is occupying a resource (station or track), then we have an arrow from resource to train. If a train is waiting for a resource to be freed, then we have the arrow from train to resource.

3.5.2 Deadlock detection

Simulator encounters deadlock if the next chosen move is infeasible because (i) vehicle v finds all resources at the next node occupied by other vehicles, and (ii) these other vehicles can only release their current resources if they move into the resource currently occupied by v. In the figure below, there are four trains at station Charlie, one train on track Delta-Charlie, trying to move to station Charlie and one train at track Charlie-Bravo, trying to move to station Charlie. Since no trains can move in this scenario, so it is in deadlock.
In this simulator, each resource (station or track) is having multiple instances (lines). If each resource have only one instance, then deadlock can be detected by cycle in resource usage graph. But since, each resource is having multiple instances, we have to use banker's

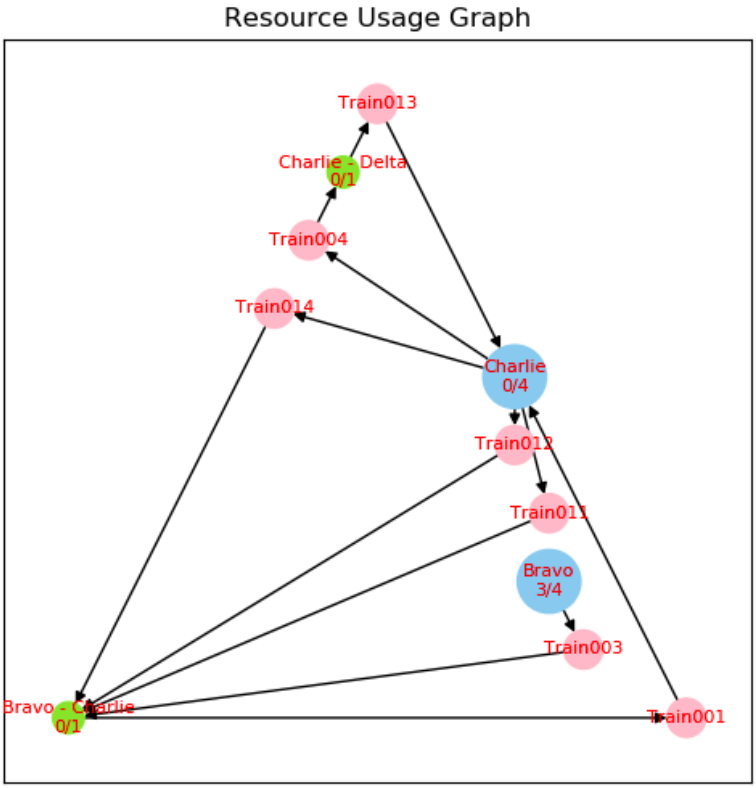

Fig. 3.6 Resource usage in the network.

algorithm to detect deadlock. Once the deadlock is detected, simulation is terminated and huge negative reward is given.

Banker's algorithm

This algorithm is originally used to avoid deadlock. We are going to modify this to detect deadlock. Here, trains are treated as processes and tracks or stations are treated as resources. Let's n be the number of processes (trains) and m be the number of resource categories (number of stations and tracks in the network). The banker's algorithm relies on several key data structures:

1. $Available[m]$ indicates how many resources are currently available of each type.
2. $Max[n][m]$ indicates the maximum demand of each process of each resource.
3. $Allocation[n][m]$ indicates number of each resource category allocated to each process.
4. $Need[n][m]$ indicates the remaining resources needed of each type for each process. (Note that $Need[i][j] = Max[i][j] - Allocation[i][j]$ $\forall i, j.$)

This algorithm determines if the current state of a system is safe, according to the following steps:

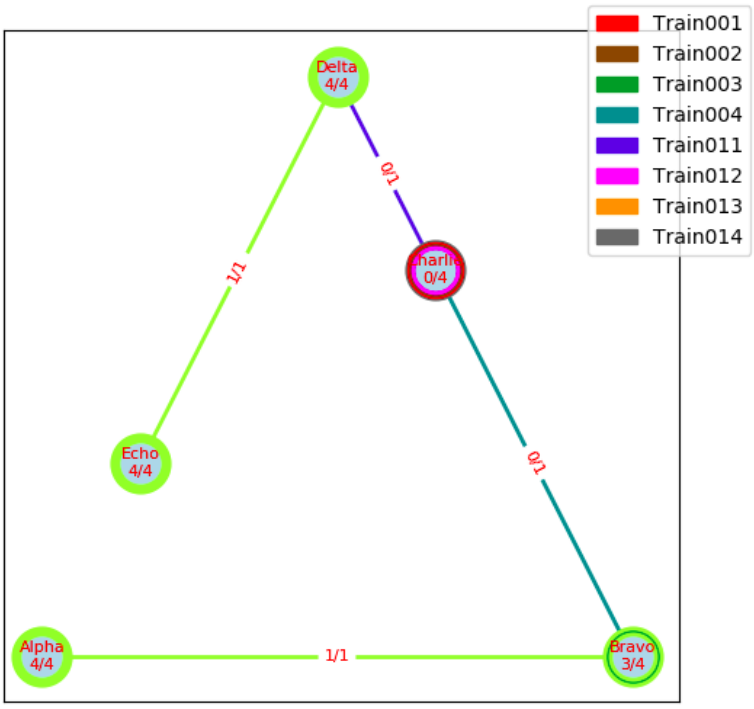

Fig. 3.7 Deadlock near station Charlie.

1. Let Work and Finish be vectors of length m and n respectively.

   - Work is a working copy of the available resources, which will be modified during the analysis.
   - Finish is a vector of booleans indicating whether a particular process can finish (or has finished so far in the analysis).
   - Initialize Work to Available, and Finish to false for all elements.

2. Find an i such that both (A) $Finish[i] == false$, and (B) $Need[i] < Work$. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.

3. Set $Work = Work + Allocation[i]$, and set $Finish[i]$ to $true$. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.

4. If $finish[i] == true \; \forall i$, then the state is a safe state, because a safe sequence has been found.

This algorithm is used to avoid deadlock. In our case *Available* is the number of resource instances (lines on stations or tracks) free. *Allocated* is the number of resource instance (lines on station or tracks) occupied by the train and *Requested* is the resource instance

requested by the train. If we use $Max = Allocated + Requested$, then above can be used to **detect deadlock** in the system.

Another important question is when to use deadlock detection algorithm. If use it very frequently then it's waste of computation as most the time system will not be in deadlock. If we use it very less then system may be in deadlock for long time. So we have to find some middle ground. Here, we are currently checking deadlock after every 20 units of time, it may be changed in the future. If the system is in deadlock, then simulation is terminated.

3.5.3 Deadlock avoidance heuristic

Multiple trains are running in the network. It is possible that multiple trains need action at a particular simulation time. So we have to pick one of these trains, and take action (move or wait) corresponding to train. This is done using deadlock avoidance heuristic based on [12]. Intuitively, pick the train which is in the most congested resource first. The lower the number of free tracks in a resource, the higher the congestion, and the earlier the processing of a train occupying that resource. This way we can avoid deadlock upto certain extent.

This algorithm takes TRAINS_NEEDING_ACTION, which is a global variable consisting of all the trains that need action at a particular time, as input and gives the name of the train that is suitable to take action as output.

Algorithm

1. Find status of all trains (not yet started , running , at last station or completed journey) in TRAINS_NEEDING_ACTION.
2. Remove all trains that have completed there journey since they don't need action and generate a warning in log as such train should not be in TRAINS_NEEDING_ACTION.
3. If there is such a train which is on last station but not freed the resource, then return that train and terminate the algorithm.
4. If there is a train that has not yet started and waiting to be put on first station, then return that train and terminate the algorithm.
5. Now all the trains are running. Construct an array where each element is a tuple of size 5 and corresponds to trains in the TRAINS_NEEDING_ACTION. Items in the tuple are :
    - Name of the train
    - Resource (track or station) it is occupying.
    - Congestion on the resource, given by number of occupied lines on the resource.
    - Priority of the resource, given by minimum of the priority of all trains on the resource.
    - Priority of the train

6. Pick train which is on most congested resource. If there is one such unique train then return it and and terminate the algorithm. If not, go to next step.

7. Out of trains chosen from step 6, pick train on resource with highest priority. If there is one train needing action on that resource then return it and terminate algorithm. If not, go to next step.

8. Out of trains chosen from step 7, pick train with highest priority. If there are multiple train then choose any one randomly and return it.

3.6 Simulator

This module is responsible for carrying out the whole simulation and putting all the components in place. The way it does this is by creating processes that interact with each other and runs the simulation. This module is implemented with the help of simpy that helps to create different processes. SimPy is a discrete-event simulation library. The behavior of active components (like trains, deadlock detection or creating graphs) is modeled with processes. All processes live in an environment. They interact with the environment and with each other via events (which is created by this module). Note all the processes are running **concurrently**. At last, Simpy is using priority queue to order the events. There is a clock in the environment and it is the simulator that runs the clock, esentially running the simulation.

Simulator module first create the network (with the help of railway network component) and then the **environment** under which the simulation is carried out. Then it creates various processes that runs in this environment. The processes are :

1. **Trains**

   There are multiple trains which are running in the network. Each train is an instance of train class implemented in the train component. Simulator creates each train as a process. These trains are running over the same resource pool (railway network) and simulator helps in scheduling and running each train. Each train have two actions, either to move or to wait and these actions are implemented using **choose action** process.

2. **Choose action**

   This process always runs and take actions for each train in the network. Initially there are no trains in the network. Simulator puts them at the initial station at approriate time (depending on the schedule train is following). Once the train is put in the network, each train is either to move to the next resource (station or track) or wait for some time at the current resource (predefined to 1 unit time, can be altered). These actions help the train to complete it's journey from source to destination. There can be multiple trains that need action at the same time. TRAINS_NEEDING_ACTION is a global variable, that keeps track of the trains that need action at current simulation time. So there are two tasks at hand :

- Choose a train from TRAINS_NEEDING_ACTION for taking an action. In this simulator, we are using deadlock detection heuristic based on [12] for picking the train. Essentially this heuristic breaks the tie when multiple trains are waiting for taking the action.
- Next step is to take the action, either to move or to wait. Choice of action depends on the state space of the train (discussed in detail in Algorithm section). We can also randomize this process by choosing the action randomly with fixed probabilities.

3. **Deadlock detection**

   This process is invoked after every predefined time (20 units) and checks if the network is in deadlock or not. **Banker's Algorithm** is used as the deadlock detection algorithm (discussed in detail in resource usage module). If the network is in deadlock, then simulation of the current episode terminates.

4. **Create Statistics**

   This process is invoked after every predefined time (20 units) and generates statistics about the current state of the network in the main log file (look at log generation component). The statistics include :

   - Number of trains not yet started.
   - Number of trains currently running in the network.
   - Number of trains that have completed their journey but the resource is not freed.
   - Number of trains the have completed their journey and all the resources are freed.

   If all the trains have completed their journey and all resources are freed then the simulation is terminated. More statistics about the state of the network can be added in future.

5. **Update Graph**

   This process is responsible for creating the running GIF of the railway network and the trains running on the network. It's purpose is only visualization that further helps in debugging and analysis.

All these processes are run by the simulator. In future, more processes can be added with different functionality. All one has to do is to create a component and then the simulator will create a process that runs the component.

Algorithm Details

The principal goal of the algorithm is to compute schedules for railway lines (either from scratch or from a given starting state) while having comparable online computation requirements. Three challenges need to be overcome to in order to achieve this objective

1. The algorithm must be able to handle different infrastructure and train service instances.
2. It must scale to large, realistic railway lines.
3. It must manage simultaneously moving trains.

The first challenge is addressed by defining a map from the specific state of the instance to a generalised state space of fixed size. The second challenge is handled by decentralising the decisions for individual trains, and limiting the feature vector to a fixed local horizon around each train. Finally, the ordering of train moves is handled by a discrete event simulator which picks the order using a previously defined deadlock-avoidance heuristic. Each component is described below.

Note that this algorithm focuses on computing schedules for **railway lines [2] , [13] instead of railway networks**. First we will define the algorithm, test it and then have a look at why it can't work in railway network setting and then try to expand the learning to the railway network.

4.1 Generalised State Representation

We compute the state space as a function of **local neighborhood** of each train. A state vector is computed for each train every time a decision about its next move is to be computed. Relative to the direction of motion, we define resources as being behind (in the direction opposite to the direction of motion) or in front (in the direction of motion) of the train. A user-defined finite number of resources l_b behind each train and l_f in front of each train are used for defining the state vector. These are referred to as local resources. Including a few resources behind the train in the state definition ensures that overtaking opportunities for fast-moving trains are not missed. The total number of local resources is $(l_b + 1 + l_f)$.

The entry in the state vector corresponding to each local resource takes one of R integer values $\{0, 1, 2, ..., R-1\}$, referred to as the status S_r of resource r. Higher values indicate higher congestion within the resource, and are driven by the number of occupied tracks. Let us define the number of tracks in resource r to be equal to N_r, out of which $T_{r,c}$ tracks contain trains converging with (heading towards) the current train, while $T_{r,d}$ tracks contain trains diverging from (heading away from) the current train. Since at most one train can occupy a

given track, we note that $T_{r,c} + T_{r,d} < N_r$. The mapping from track occupancy to resource status is,

$$S_r = R - 1 - min(R - 1, \lceil N_r - w_c T_{r,c} - w_d T_{r,d} \rceil).$$

Here, $0 \leq w_c, w_d \leq 1$ are weights that can de-emphasise the effect of converging and diverging trains on the perceived status of a resource.

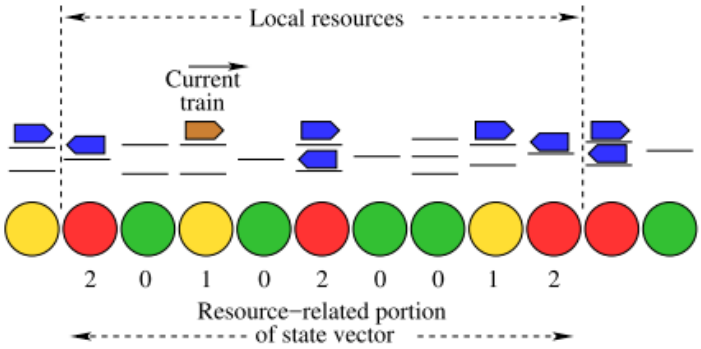

Fig. 4.1 Mapping train location and direction of movement to resource status, relative to the 'current train' [2].

In addition to the resource-related entries, the state includes an entry for the priority of the current train.

The complete state representation used is a vector x of length $(l_b + l_f + 2)$, including the integer priority value and $(l_b + 1 + l_f)$ entries for the status of local resources. If we assume that the model accommodates up to P priority levels, the size of the state space is equal to $(P * R^{l_b+1+l_f})$.**Note that this value does not depend on the scale of the problem instance, in terms of the number of trains, the lengths of their journeys, and the number of resources**. One of the key advantage of using the local horizon as the state for the train is it's independence from the size of the problem instance. Another advantage is of **transfer learning** which we will see in later sections.

4.2 Action and Policy Definition

The reinforcement learning procedure maps each state vector to a probability of choosing the action to be taken. In this study, the choice of actions in any given state is binary, with 0 representing a decision to move the current train to the next resource on its journey, and 1 representing a decision to halt in the current resource for a predefined time period (1 minute in this paper). If the train is halted, the decision-making procedure is repeated after the time period elapses. The order in which trains are selected for move/halt decision-making is given by deadlock avoidance heuristic in resource usage module. Let us assume that a

particular train occupying one track of some resource r has been selected, the state vector has been computed, and the action (move or halt) is to be chosen. In addition to the state vector, the choice of action is driven by the policy. Policy depends on the approach that we are going to use.

4.2.1 ϵ - greedy policy

Given a state, the two possible actions $a \in \{0 : \text{move}, 1 : \text{halt}\}$ result in two unique state-action pairs. Each state-action pair (x, a) is associated with a Q-Value q(x, a) which quantifies its desirability. The higher the Q-Value, the higher the desirability of the relevant pair. The ϵ-greedy policy chooses the greedy option (higher Q-Value) with probability $(1-\epsilon)$, and a randomised action with probability ϵ. The greedy choice corresponds to **exploitation** of the learning so far, while the randomised choice corresponds to **exploration** of the state-action space. We are going to use this policy with standard Sarsa(λ) the results of which are not good.

4.2.2 Modified ϵ - greedy policy

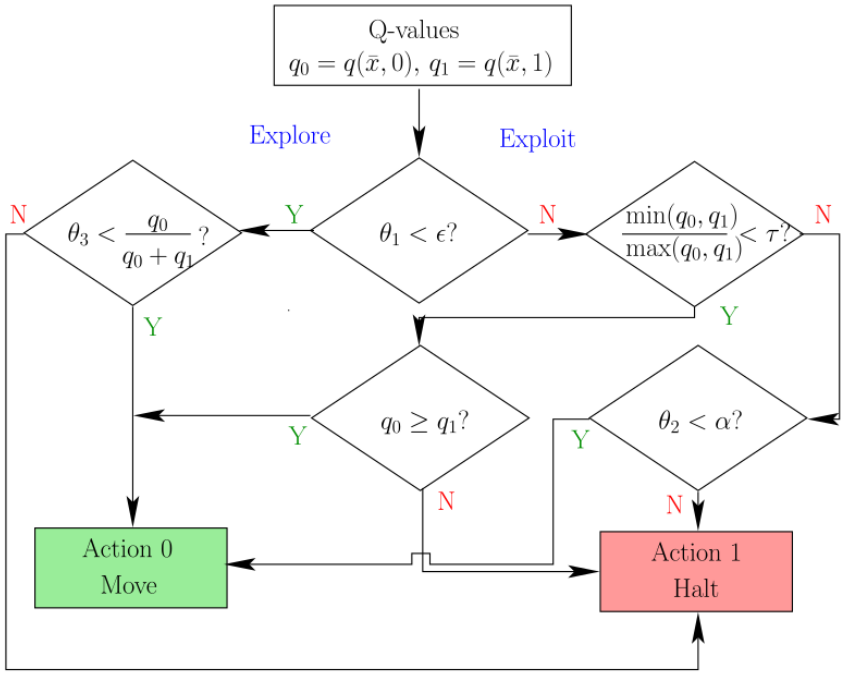

Fig. 4.2 Modified ϵ - greedy policy [2]

This is a modified version of the ϵ - greedy policy. In exploration mode, the action is chosen with the toss of a biased coin, based on the relative Q-Values q_0 and q_1 of the two actions. In the exploit mode, if $q_0 \approx q_1$ within a user-defined threshold τ , a further biased coin toss is used to compute the action. The bias in this case is given by a user-defined aggression parameter α, which controls the probability of choosing 'move' when $q_0 \approx q_1$. If q_0 and q_1

are clearly separated, the action with the higher Q-Value is chosen. The value of ϵ starts at 1 in the first training episode and decreases as more episodes are completed. This moves the policy gradually from exploration towards exploitation.

The value of ϵ remains the same throughout the whole episode. In the first episode $\epsilon = 1$ and as the episode passes the value of epsilon decreases linearly upto min ϵ, after which it remains constant throughout the whole training.

4.3 Objective Function

A number of objective functions have been used in the railway scheduling context, in order to achieve goals such as delay reduction, passenger convenience, and timetable robustness. One of the commonly used measures of schedule quality is priority-weighted delay. A delay is defined to be the non-negative difference between the time of an event as computed by the algorithm, and the desired time as specified by the timetable. The priority-weighted average delay is the mean over all trains and all stations of individual delays divided by train priorities. This quantity is used as the objective function, but the algorithm can accommodate other measures equally easily (for example, a non-linear function of delays in order to increase fairness of delay distribution).

$$J = \frac{1}{N_{r,t}} \sum_{r,t} \frac{\delta_{r,t}}{P_t}$$

where $\delta_{r,t}$ is the delay for train t on departure from resource r, p_t is the priority of train t, and $N_{r,t}$ is the total number of departures in the schedule. Note that this expression includes all events for all trains, for their entire journey.

4.4 Sarsa(λ)

One of the typical algorithm is to use the objective function defined above as the negative of the reward. In that case, we will be having the reward at the end of each episode (terminal episode). In each episode, each train is going through certain state-action pairs, forming a trajectory. And at the end of each episode, we will get the reward. We can backpropogate the reward through this trajectory to learn the Q-values. This learning is done using Sarsa(λ).

In Sarsa(λ), we are using standard ϵ - greedy policy. This section discusses the forward view of Sarsa (λ) using eligibility traces. Q-values are updated using

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \delta_t e_t(s,a) \;\; \forall (s,a)$$

where

$$\delta_t = r_{t+1} + \gamma * Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$

and

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s,a) + 1 & if\ s = s_t \ \ and\ a = a_t \\ \gamma\lambda e_{t-1}(s,a) & otherwise. \end{cases}$$

Here, α is the learning rate, γ is discount factor, r_{t+1} is the reward (we have only one reward at the end of episode). Algorithm is given below.

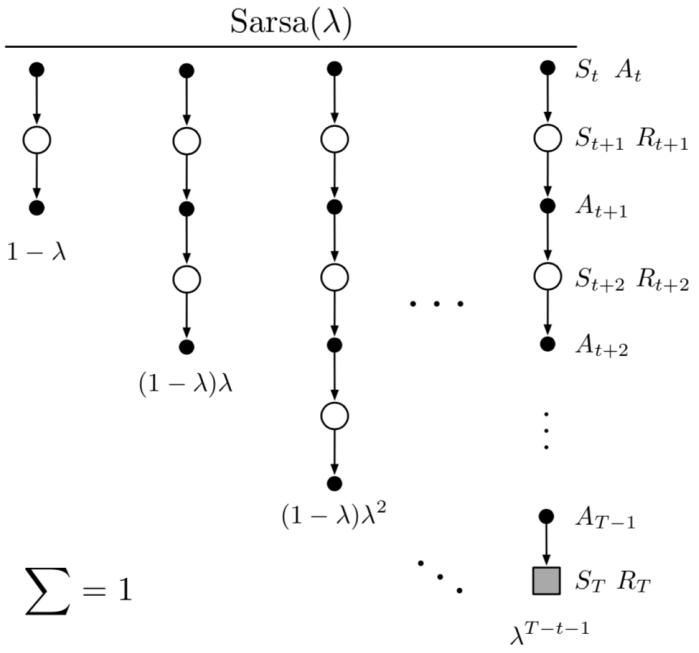

Fig. 4.3 Sarsa(λ)'s backup diagram [3]

Algorithm 1 Sarsa Lambda [3]

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0\ \forall s,a$
Repeat (for each episode):
  Initialize s,a
  Repeat (for each step of episode):
    Take action a, observe r,s'
    Choose a' from s' using ϵ - greedy policy
    $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all s,a
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'; a \leftarrow a'$
  until s is terminal

4.5 Proxy reward

The algorithm maintains a threshold of J as the goal to be achieved in each episode. This maximum acceptable level is set to a proportion $(1 + \rho)$ of the minimum J observed thus far, where $\rho > 0$ is a user-defined constant. The threshold becomes tighter as the best known J is improved upon during learning. A reward of $+1$ (success) is given if the sum of the priority-weighted delay is under the current threshold, and 0 (failure) either if it is over the threshold, or if the episode enters deadlock and does not terminate.

The Q-Values are defined using the probability of success when an episode passes through a given state-action pair. Instead of tracking the entire sequence of state-action pairs in a given episode, a binary indicator variable $b(x, a)$ corresponding to each pair (x, a) is set to TRUE whenever it is observed in a given episode. Upon termination of the episode, the number of successes (or failures) of all (x, a) where $b(x, a) = TRUE$ are incremented by 1. The success probability $\sigma(x, a)$ is computed by dividing the number of $+1$ rewards associated with the pair, by the total number of episodes that passed through this pair. If $\epsilon_{x,a}$ is the number of all episodes that passed through (x, a) at least once, and $\epsilon^*_{x,a}$ is the number of these that ended in success,

$$0 \leq \sigma(x, a) = \frac{\epsilon^*_{x,a}}{\epsilon_{x,a}} \leq 1$$

While $\sigma(x, a)$ provides a way to quantify the desirability of a given state-action pair, it does not encapsulate the state trajectory. On the other hand, a core tenet of reinforcement learning is the back-propagation of rewards through the trajectory of state-action pairs (usually upon episode termination). However, in the current context, episodes can be very long, reward is generated only upon episode termination, and state- action pairs for multiple trains are generated simultaneously. Therefore, $\sigma(x, a)$ is used in as a **proxy reward**[2]. We differ from the method mentioned in the paper by defining the Q-value differently.

4.5.1 Prior

In the paper, Q-value is defined as

$$q(x, a) = w\sigma(x, a) + (1 - w) \sum_{m=1}^{M} \frac{\sigma(x'_m, a'_m)}{M}$$

where w is a weighting factor between the success rate of a given pair, and the average success rate of its M neighbours. These M neighbours are not necessarily unique pairs, and thus neighbouring pairs that are more frequently observed have a greater contribution to the average value. It is more like two step reward, in which Q-value is defined as the combination of success probability of current state-action pair and success probability of neighbors. Instead of looking at all the way down to trajectory, we are just looking at two steps of trajectory.

4.5.2 Proposed

We propose to have a Q-value that looks all the way down to the trajectory instead of two step reward functions. We propose Q-value as,

$$q(x,a) = w\sigma(x,a) + (1-w)\sum_{m=1}^{M}\frac{q(x'_m,a'_m)}{M}$$

where w is more like discount factor, that controls how far we are looking into the future. When $w = 1$, q-value just have immediate reward, when $w = 0$, Q-value is looking all the way into the future. We will see later, that this small change changes the result significantly. Again, these M neighbours are not necessarily unique pairs, and thus neighbouring pairs that are more frequently observed have a greater contribution to the average value. We can also remove w and put γ as a discount factor,

$$q(x,a) = \sigma(x,a) + \gamma\sum_{m=1}^{M}\frac{q(x'_m,a'_m)}{M}$$

One of the key advantage of defining Q-value like this, is that we can use standard RL methods to learn the Q-values. Here, reward is the proxy reward that we have defined earlier and RL methods will try to maximize this reward, getting tighter and tighter bounds on J and thus eventually decreasing the overall objective function. Although testing is done only for the first formula and second hypothesis still needs to be tested.

4.6 Transfer Learning

Transfer learning make use of the knowledge gained while solving one problem and applying it to a different but related problem. One of the key advantage of using proxy rewards is that we can transfer the knowledge gained while training one problem instance and then use it to test it on other problem instance. That means if we have a big problem instance, then once we train our algorithm on that problem instance, we can store its Q-value, and when training on other problem instance we can use that Q-value as the starting point. Moreover, we can directly test using the stored Q-values. The results of transfer learning is shown letter in the Experiment section.

Experiments

The algorithms as described above, is tested on three hypothetical problem instances. Description of all the problem instances is defined in the next section. Two real life problem instances, Ajmer and Konkan railways still need to be tested but the hypothetical problem instances are comparable in size to the real life instances. Code for whole implementation is at [5].

5.1 Problem Instances

5.1.1 HYP-1

The simple hypothetical instance HYP-1 consists of 8 trains travelling 5 stations each. All trains are already in the system at the start. Four trains heading left to right are initially located in station Alpha, while the others are in station Echo, heading right to left. Each station contains 4 parallel tracks, and there is a single track between stations. Only for HYP-1, the trains all have the same priority, running times between stations, and halt times at stations.

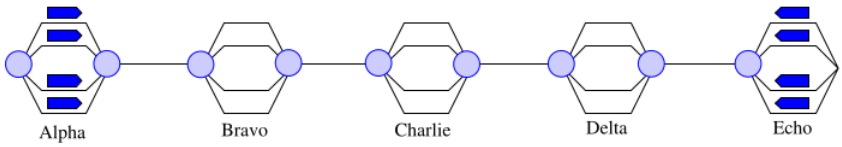

Fig. 5.1 HYP-1 [2]

5.1.2 HYP-2 & HYP-3

HYP-2 have 11 stations each having 3 tracks and each station is connected by a single bidirectional track. The scheduling problem starts from the clean slate i.e. initially no train is in the network. HYP-2 have 60 trains, 40 priority-1 train and 20 priority-2 train running. HYP-3 have double the number of trains i.e. 120 trains with 80 priority-1 train and 40 priority-2 train. Since the number of trains have doubled up, in HYP-3 each station have 4 tracks and single track is connecting these stations.

Table 5.1 Hypothetical instances

Name	Stns.	Trains (sorted by priority)	Events
HYP-1	5	8,0,0	40
HYP-2	11	15,45,0	1320
HYP-3	11	40,80,0	2640

5.2 Hyperparameters

All results use a look-forward of $l_f = 6$ resources, and a look-back of $l_b = 2$ resources. The status of each resource takes one of three values: 0, 1, or 2 ($R = 3$). The weight on converging trains is $w_c = 0.9$, while that on diverging trains is $w_d = 1$. The maximum number of priority levels is $P = 3$. In case of modified ϵ - greedy policy, the threshold for checking whether $q_0 \approx q_1$ is $\tau = 0.9$, and the aggression parameter is $\alpha = 0.9$. The threshold for determining the maximum acceptable J is $\rho = 0.25$. In case of Sarsa(λ) , $\lambda = 0.9$ is choosen.

Training is run for 500 episodes, where the ϵ is linearly reduced from 1 to 0.1 in 300 episodes. Note, that the value of ϵ remains the same in each episode. Decrease in the value of ϵ is per episode basis.

5.3 Results

5.3.1 Sarsa (λ) on HYP-1

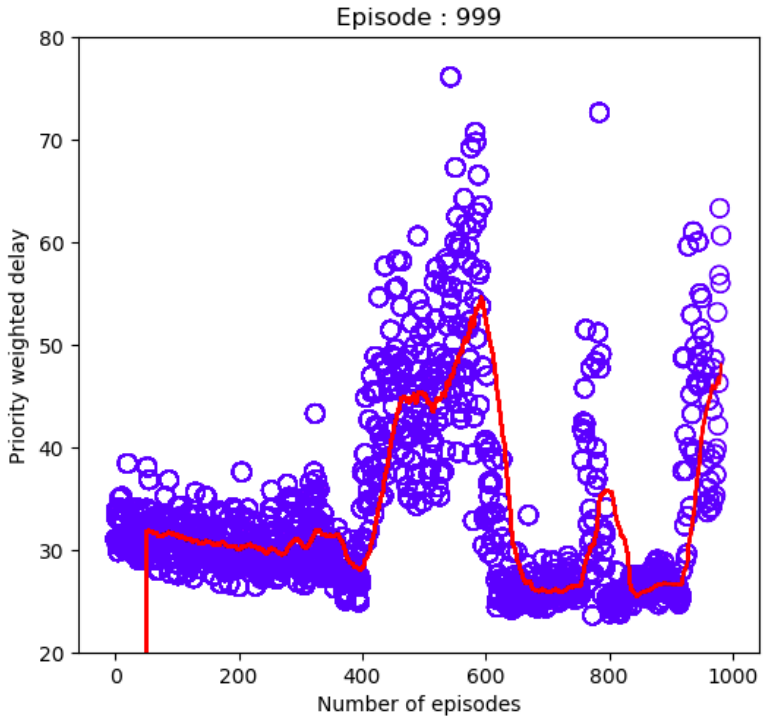

Fig. 5.2 Priority weighted delay for Sarsa(λ) on HYP-1 over 1000 episodes training

Sarsa(λ) is not able to find an optimal schedule even for HYP-1. In the above plot, blue dots are the delay for one training episode, while red line is the running mean over the last 50 episodes. The results of this algorithm is not good (in the next section) due to following reasons,

1. The back-propagation of rewards after the end of the episode is not possible, because the episode can be very long.

2. In the trajecotry of a train, it is possible to visit the same state-action pair in loop leading to large accumulation of reward at that state-action pair, leading to extreme values.

3. Moreover, the magnitude of delays (and hence the theoretical optimum value of J) is different from one problem instance to another. Quantifying rewards directly in terms of delays $\delta_{r,t}$ would create obstacles when transferring the learning from one instance to another (**obstacle in transfer learning**).

5.3.2 Proposed work on HYP-1

Q-values are initialised to 0.5 as they in some sense represent the probability of taking action either to move or to wait. Training on the HYP-1 using the proposed approach, results in the convergence of Q-values. The Q-values decreased from around 35 to around 25. **Minimum is 23.58750**. Note the delay is only for those episodes that are successfully completed (not ending up in deadlock). Total number of episodes that ended up in deadlock are only 3 out of 500 episodes.

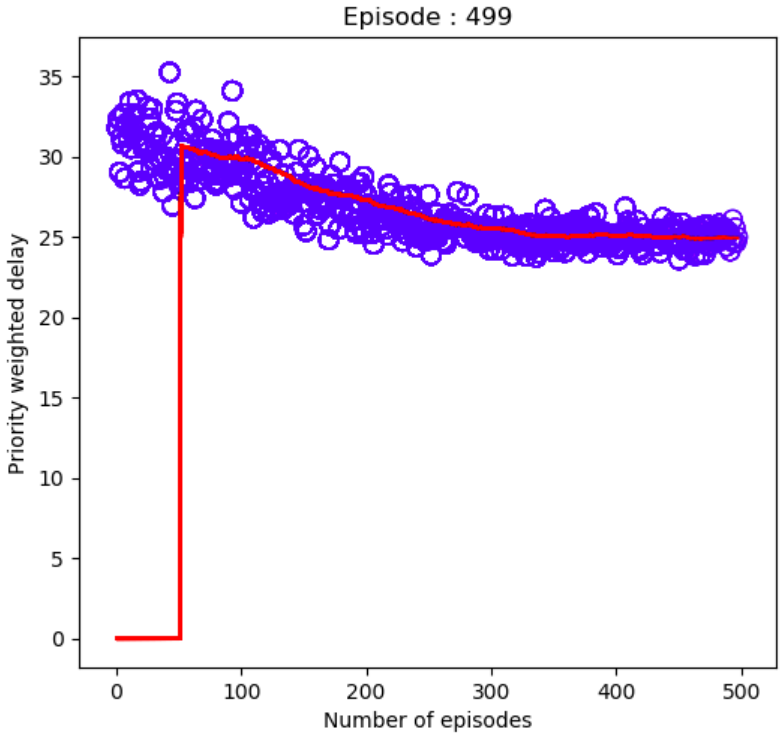

Fig. 5.3 Priority weighted delay for proposed approach on HYP-1 over 500 episodes training

The figure 5.4, shows time on the x-axis and distance on the y-axis. Each solid line shows the trajectory of one train as it moves from one end of the line to the other. The horizontal

portions correspond to halts at stations, while the inclined portions denote movement between stations. Since there is a single track between successive stations and only one train can occupy it at a time, inclined lines cannot cross each other in a feasible schedule. The horizontal dotted lines indicate specific tracks within station resources, and no two solid lines are allowed to overlap within these tracks.

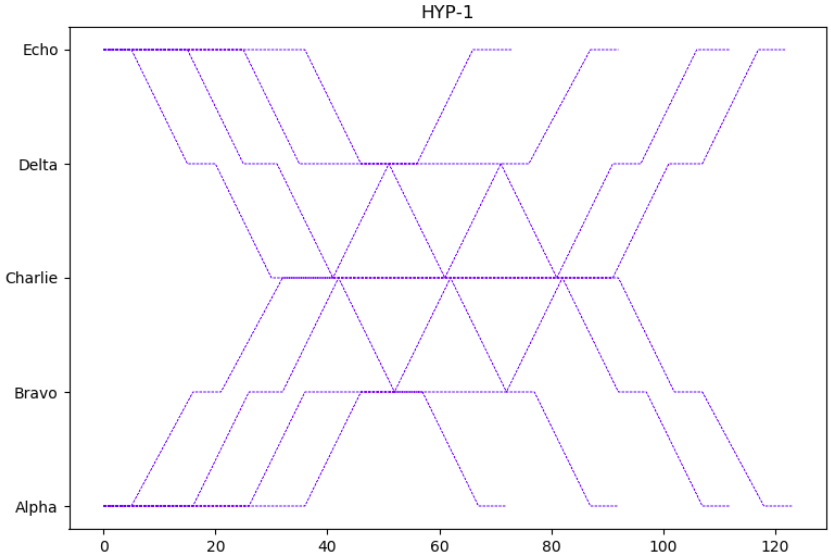

Fig. 5.4 Schedule computed using proposed approach for HYP-1

5.4 Training on HYP-2 & HYP-3

The toy instance like HYP-1 shows that the proposed approach is able to learn Q-Values (and hence the policy for choosing actions) at a small scale. However, when instances of a realistic size and complexity are run with the same initial Q-Values (0.5%), they require a large number of training instances to start producing feasible solutions. This happens because the larger problem instances require several thousand decisions to be made 'correctly' for successful completion. When such decisions are made purely randomly, the instances frequently end with trains in deadlock situations. So in order to speed up the training, the Q-values are initialized according to heuristic 5.2.

Table 5.2 Heuristic For initialising Q-values

#	States Satisfying	Action	Init Q-values
1	Next Resource is Full	Move	0
	$S_{r,next} = R - 1$	Stop	50
2	Atleast three consecutive resources are full	Move	10
	$S_r = S_{r+1} = S_{r+2} = R - 1$	Stop	15
3	Next Resource almost Full (R-2)	Move	15
	Next but one is full (R-1)	Stop	50
4	Average status of upcoming resources	Move	85
	is between 0.5 and 1.0 (moderately empty)	Stop	90
5	Average status of upcoming resources	Move	95
	is less than 0.25 (almost empty)	Stop	50

Training over different problem instances is summarized by table 5.3. First row corresponds to training using prior Q-value while second row corresponds to proposed Q-value . Second column shows the minimum objective function attained during training, third column shows number of episodes ending in deadlock out of 500 training episodes and fourth column shows the total number of episodes visited out of $3 * 3^{2+1+6} = 59049$ states. Table 5.3, shows HYP-3 is the most extensive problem instance visiting large number of states, so can be ideally most suitable for transfer learning.

Table 5.3 Training

Instance	Minimum	Deadlock	Total states visited
HYP-1	23.53750	3	354
	23.58750	3	348
HYP-2	2.60682	3	1650
	2.58447	5	1712
HYP-3	11.64754	32	3377
	11.34754	29	3021

Training on HYP-3 is able to converge from 25 to almost 12 with 11.34754 as the minimum objective value. Red curve in the plot is the running mean over the last 50 episodes. In the figure 5.6, ratio of episodes ended in deadlock as the training progresses is shown. It can be clearly seen, as the training progresses, percentage of episodes ending in deadlock starts to decrease with almost 25% in 100 episodes to almost 5% in 500 episodes. This shows that the algortihm learns to avoid deadlock as the training progresses.

Figure 5.7, shows the schedule generated for HYP-3. The plot shows the density and complexity of the schedule generated for the trains running over the HYP-3 network. Blue dashed line shows the trajectory of priority-1 train while red dashed line shows the trajectory of priority-2 trains.

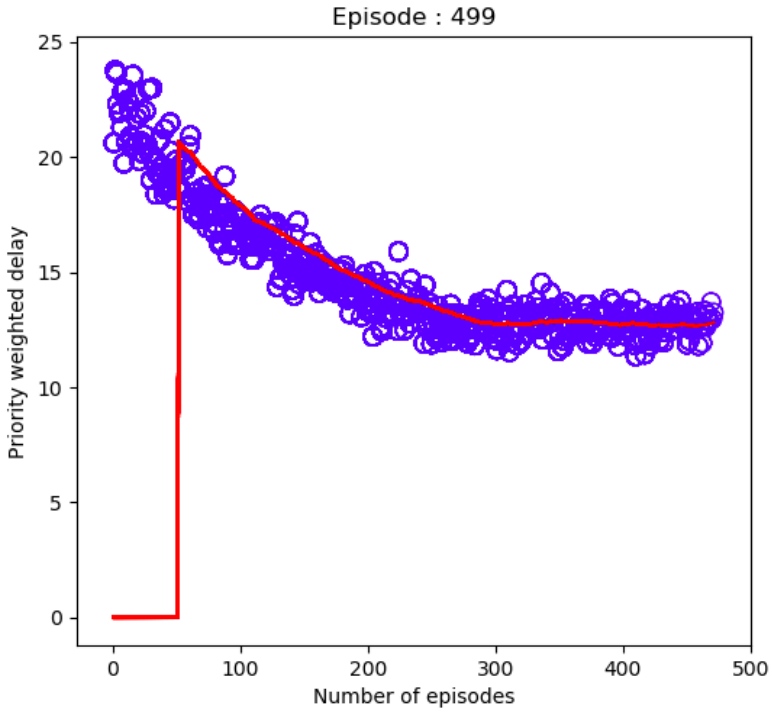

Fig. 5.5 Objective function for training over HYP-3

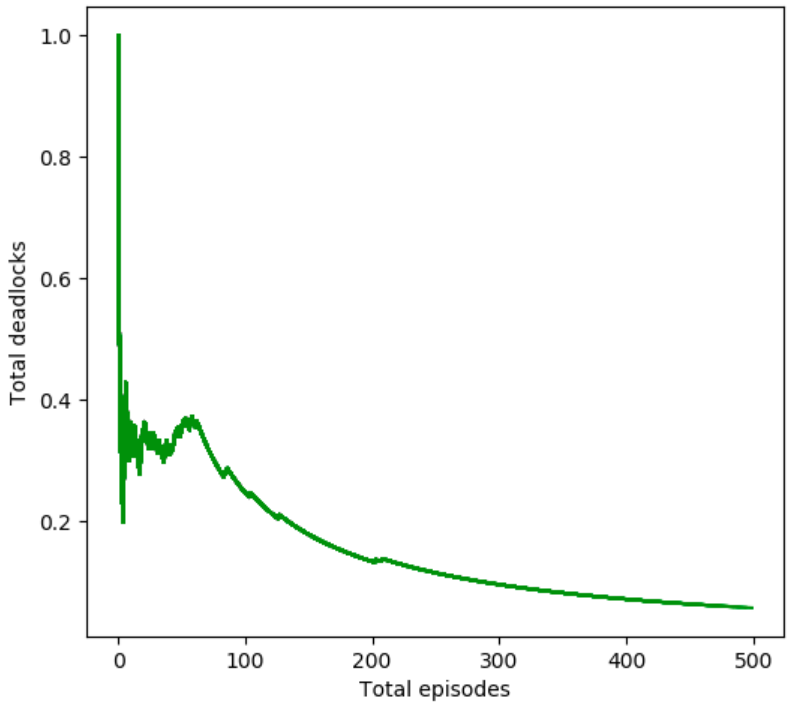

Fig. 5.6 Deadlock in % for training over number of episodes in HYP-3

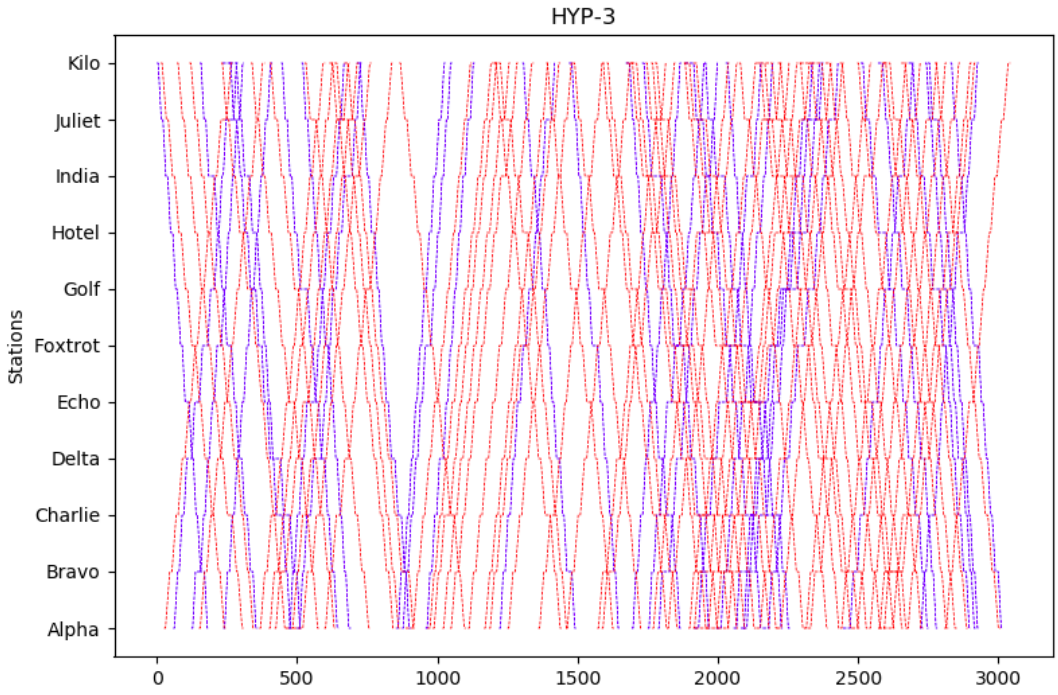

Fig. 5.7 Schedule computed using proposed approach on HYP-3 with blue line for priority-1 train and red line for priority-2 train.

5.5 Testing and transfer learning

This section summarises the result of testing and transfer learning (training on one problem instance and testing on other problem instance).Testing is done over **50 episodes**. Two types of testing are done,

1. **Zero Delay** : No external delay is introduced in the system.

2. **20% Delay** : Delay of 20% is introduced in the system. Delay is introduced by randomly increasing the running time between two consecutive stations. 20% means that running time almost shoots by 20%. Note delay is introduced in the whole network uniformly. In future, delay can be introduced only in section of the network.

Following table summarizes the results of testing. First row shows the results for prior work while second row shows the results for proposed work . It is clear from results that proposed Q-value works better compared to proposed one.

Table 5.4 Zero delay

Train	Test	Minimum	Average	deadlock
HYP-2	HYP-2	4.050	4.980 ± 0.590	0
		2.680	**3.257 ± 0.501**	**0**
HYP-3	HYP-2	3.089	**4.080 ± 0.370**	0
		2.709	4.184 ± 0.438	**0**
HYP-2	HYP-3	12.683	14.580 ± 1.058	16
		11.453	**13.083 ± 1.164**	**6**
HYP-3	HYP-3	11.855	12.954 ± 0.540	1
		11.438	**12.734 ± 0.613**	**0**

Table 5.5 20% delay

Train	Test	Minimum	Average	deadlock
HYP-2	HYP-2	9.591	11.388 ± 1.258	**0**
		8.386	**10.261 ± 0.733**	1
HYP-3	HYP-2	9.603	10.932 ± 0.881	3
		8.473	**10.472 ± 0.792**	**1**
HYP-2	HYP-3	**26.882**	31.955 ± 2.290	**23**
		27.734	**30.141 ± 1.486**	29
HYP-3	HYP-3	**26.522**	30.135 ± 1.649	3
		26.560	**29.231 ± 1.723**	**1**

In future, we are planning to test on real life datasets like ajmer and konkan railway lines. Delay can be introduced only in some part of the network to check the robustness of the algorithm. Moreover, testing on railway network (with atleast one intersection of railway lines) is also to be done. However it is very less likely that the given algorithm performs good over those instances as the state space is defined in view of only the railway line. More robust state space (observation) in case of railway networks is **tree like observations** which is the future work.

Flatland Environment

Since we have solved the problem of railway scheduling on a railway line, we want to use the knowledge developed so far and work on **railway scheduling on a railway network**. Recently, AICrowd developed a flatland environment[11] to foster progress in multi- agent reinforcement learning for any re-scheduling problem (RSP). They have defined the railway network in a complete new setting using grid instead of graph. Future course of the project is to work on more general problem of flatland (which can be used for solve railway scheduling, transport management problems) and develop algorithms that can solve scheduling problem over grid world having multiple agents.

This chapter contains flatland environment specifications and discusses about the problem statement in detail. Flatland is usually a two-dimensional environment intended for multi-agent problems, in particular it should serve as a benchmark for many multi-agent reinforcement learning approaches. The environment can host a broad array of diverse problems reaching from disease spreading to train traffic management. The flatland environment is a two-dimensional grid in which many agents can be placed, and each agent must solve one or more navigational tasks in the grid world.

6.1 Environment

Flatland is grid-like n-dimensional space of any size. A cell is the elementary element of the grid. The cell is defined as a location where any objects can be located at. The term agent is defined as an entity that can move within the grid and must solve tasks. An agent can move in any arbitrary direction on well-defined transitions from cells to cell. The cell where the agent is located at must have enough capacity to hold the agent on. Every agent reserves exact one capacity or resource. The capacity of a cell is usually one. Thus usually only one agent can be at same time located at a given cell. The agent movement possibility can be restricted by limiting the allowed transitions.

Flatland is a discrete time simulation. A discrete time simulation performs all actions with constant time step. In Flatland the simulation step moves the time forward in equal duration of time. At each step the agents can choose an action. For the chosen action the attached transition will be executed. While executing a transition Flatland checks whether the requested transition is valid. If the transition is valid the transition will update the agents position. In case the transition call is not allowed the agent will not move.

6.1.1 Tile types

Each Cell within the simulation grid consists of a distinct tile type which in turn limit the movement possibilities of the agent through the cell. For railway specific problem 8 basic

tile types can be defined which describe a rail network. As a general fact in railway network when on navigation choice must be taken at maximum two options are available.

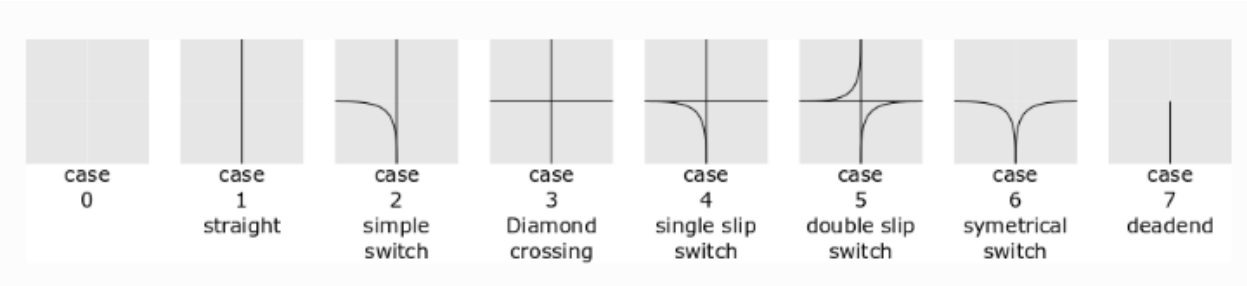

Fig. 6.1 Tile types in flatland grid [4]

6.2 Observations

In order to solve the flatland environment using RL, we need to decide the observation space and action space. Depending on the observation space, we can use different RL algorithms to solve.

6.2.1 Global Observation

Gives a global observation of the entire rail environment. The observation is composed of the following elements:

- Transition map array with dimensions (env.height, env.width, 16), assuming 16 bits encoding of transitions.
- Two 2D arrays (env.height, env.width, 2) containing respectively the position of the given agent target and the positions of the other agents targets.
- A 3D array (env.height, env.width, 8) with the first 4 channels containing the one hot encoding of the direction of the given agent and the second 4 channels containing the positions of the other agents at their position coordinates.

Global observations, specifically on a grid like environment, benefit from the vast research results on learning from pixels and the advancements in convolutional neural network algorithms. The observation can simply be generated from the environment state and not much additional computation is necessary to generate the state. Also, in this case, when we consider global observation, it is like having a superagent (a central controler), which will take action for each of the trains after having the global observation.

6.2.2 Tree Observations

Tree observations are local observation that is made for each agent in the environment whenever agent need to take the action. Also if we consider local observation of each agent then the environment becomes multiagent, as opposed to the case of global observation. The tree observation is built by exploiting the **graph structure of the railway network**. The observation is generated by spanning a 4 branched tree from the current position of the agent. Each branch follows the allowed transitions until a cell with multiple allowed transitions is reached. Here the information gathered along the branch is stored as a node in the tree. This is very similar to the observation we made in the previous chapters (in case of scheduling at railway line) where the observation is the resource information of the resource next to the train and at back to the train. The figure below illustrates how the tree observation is built:

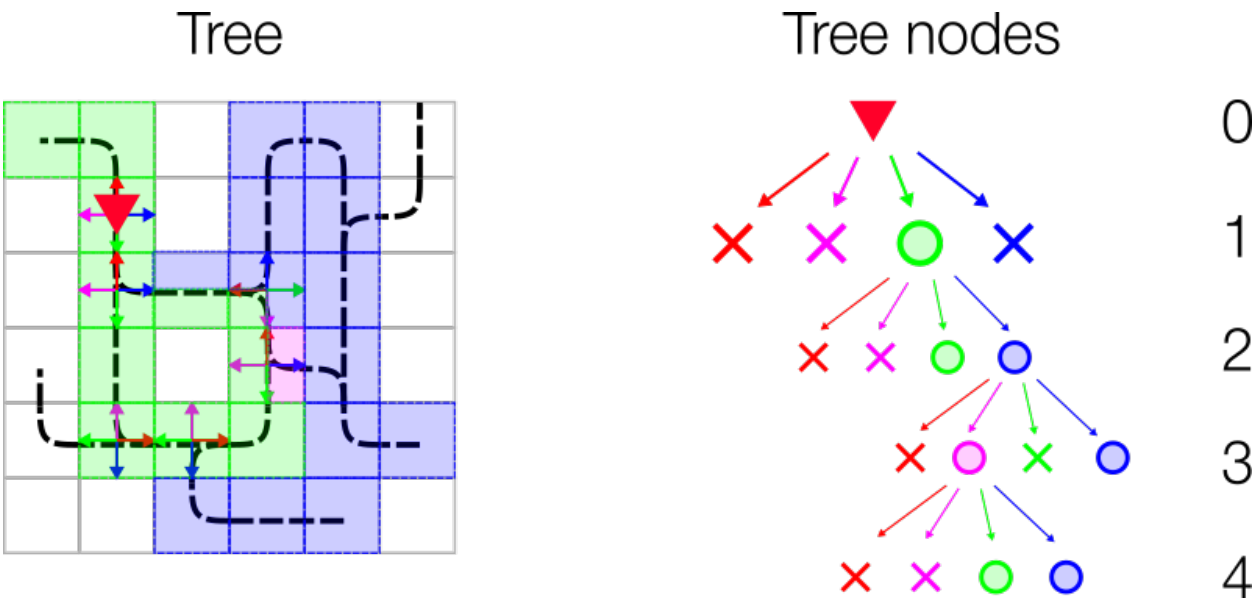

Fig. 6.2 Tree observation in flatland environment [4]

Each node is filled with information gathered along the path to the node. Each node contains 9 features:

1. If own target lies on the explored branch the current distance from the agent in number of cells is stored.
2. If another agent's target is detected, the distance in number of cells from the current agent position is stored.
3. If another agent is detected, the distance in number of cells from the current agent position is stored.
4. Possible conflict detected.

5. If an unusable switch (for the agent) is detected we store the distance. An unusable switch is a switch where the agent does not have any choice of path, but other agents coming from different directions might.

6. This feature stores the distance (in number of cells) to the next node (e.g. switch or target or dead-end).

7. Minimum remaining travel distance from this node to the agent's target given the direction of the agent if this path is chosen.

8. Agent in the same direction found on path to node.

9. Agent in the opposite direction on path to node.

6.3 Action space

Thus the actions[4] of an agent are strongly limited to the railway network. This means that in many cases not all actions are valid. The possible actions of an agent are

1. 0 **Do Nothing**: If the agent is moving it continues moving, if it is stopped it stays stopped

2. 1 **Deviate Left**: If the agent is at a switch with a transition to its left, the agent will chose th eleft path. Otherwise the action has no effect. If the agent is stopped, this action will start agent movement again if allowed by the transitions.

3. 2 **Go Forward**: This action will start the agent when stopped. This will move the agent forward and chose the go strai ght direction at switches.

4. 3 **Deviate Right**: Exactly the same as deviate left but for right turns.

5. 4 **Stop**: This action causes the agent to stop.

6.4 Problem Instances

We can generate different problem instances in the flatland environment, with different railway network (sparse as well as dense), different schedules for the agent (trains). We can also have different speed for each agent. We can also mention the malfunction parameter for each train as this will control the rate at which a particular agent malfunctions. Also this will instroduce stochasticity in the environment. For this work, we will be using three different problem instances as shown in the figure.

Fig. 6.3 A random environment. This is a toy example and minimum baseline for the algorithm[5]

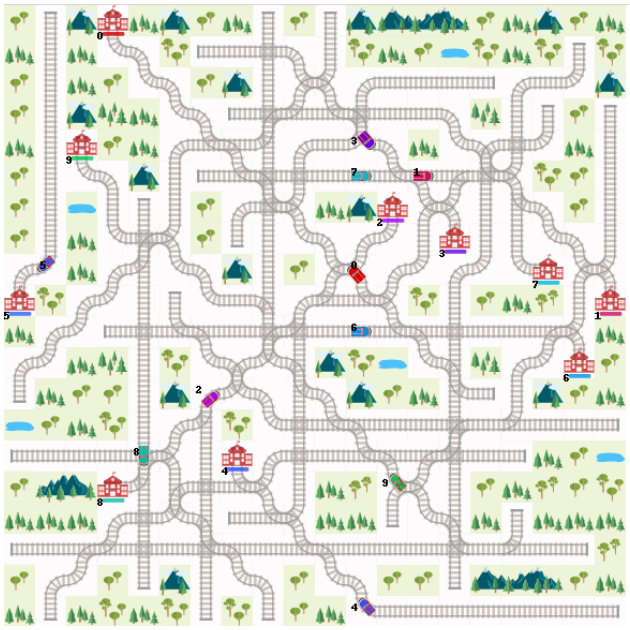

Fig. 6.4 A complex railway network. Due to the complexity of the network , there are multiple paths from each agent position to their target. Even having multiple paths doen't make this problem instance easy to solve[5].

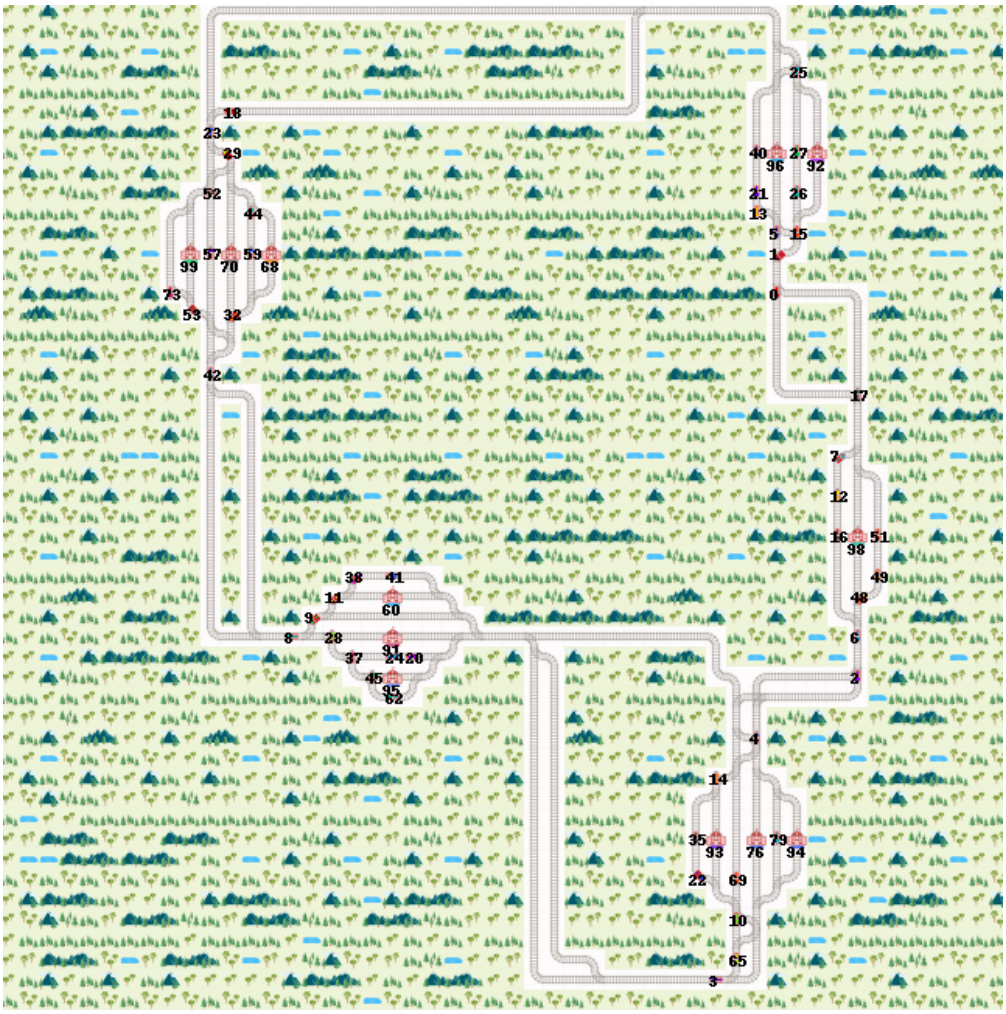

Fig. 6.5 Real life example.This example have 4 cities (where there are multiple parallel tracks and train targets denoted by red house) each connected by two parallel railway lines. We can add to the complexity by increasing the number of cities, decreasing number of parallel track. Also the number of agents in this environment can be large (like 100)[5]

Double deep Q-learning Solution

We will start this chapter by describing deep Q-learning and then present it's result.
Deep reinforcement learning is the combination of Reinforcement learning and deep learning. **Q-learning** is one of the **off-policy** learning that trains the agent and tells what is the optimal action to take in each state. This approach can be used when the number of state-action pairs are limited enough to store there q-value in the memory. Even if we are able to store q-value of each state-action pair, it is highly inefficient because the agent need to visit each state-action pair for reasonable number of times to learn their q-values. Since the total state-action pair can be very high for real life problems , better approach would be to use the **function approximator[3]** that approximates the q-value once the state and action are given as the input. Since neural networks can be used as the function approximators, so here is the point where deep learning and reinforcement learning are combined to train the agents to perform in an environment. In this report, we will be working on DDQN (double deep Q-learning), although there are whole different class of policy gradient approaches.

7.1 Algorithm

Bellman equation describing the optimal action-value function, $Q^*(s,a)$ is given by,

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim P}[r(s,a) + \gamma \max_{a'} Q^*(s',a')]$$

where $s' \sim P$ is shorthand for saying that the next state, s' is sampled by the environment from a distribution $P(.|s,a)$

This Bellman equation is the starting point for learning an approximator to $Q^*(s,a)$. Suppose the approximator is a neural network $Q_\phi(s,a)$, with parameters ϕ, and that we have collected a set $\mathcal{D}$ of transitions (s,a,r,s',d) (where d indicates whether state s' is terminal). We can set up a **mean-squared Bellman error (MSBE)** function, which tells us roughly how closely Q_ϕ comes to satisfying the Bellman equation:

$$L(\phi, \mathcal{D}) = \mathop{\mathbb{E}}_{(s,a,r,s',d) \sim \mathcal{D}} \left[\Big(Q_\phi(s,a) - (r + \gamma(1-d) \max_{a'} Q_\phi(s',a')) \Big)^2 \right]$$

When d ==True , which is to say, when s' is a terminal state—the Q-function should show that the agent gets no additional rewards after the current state.

Q-learning algorithms for function approximators, such as DQN(and all its variants[14], [15]) are largely based on minimizing this MSBE loss function. There are two main tricks employed.

Trick One : Replay Buffers All standard algorithms for training a deep neural network to approximate $Q^*(s, a)$ make use of an experience replay buffer. This is the set $\mathcal{D}$ of previous experiences. In order for the algorithm to have stable behavior, the replay buffer should be large enough to contain a wide range of experiences, but it may not always be good to keep everything. If you only use the very-most recent data, you will overfit to that and things will break; if you use too much experience, you may slow down your learning. We are using a **buffer replay of size** 10^5.

Trick Two : Target Networks. Q-learning algorithms make use of target networks. The term

$$r + \gamma(1 - d) \max_{a'} Q_\phi(s', a')$$

is called the **target**, because when we minimize the MSBE loss, we are trying to make the Q-function be more like this target. Problematically, the target depends on the same parameters we are trying to train: ϕ. This makes MSBE minimization unstable. The solution is to use a set of parameters which comes close to ϕ, but with a time delay - that is to say, a second network, called the **target network**, which lags the first. The parameters of the target network are denoted ϕ_{targ}.

The target network is just copied from the main network every some fixed number of states. In our work, the target network is updated once per main network update by polyak averaging:

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$

where ρ is a hyperparameter between 0 and 1 (usually close to 1).

7.1.1 Pseudocode

Algorithm 2 Double Deep Q-learning [14]

Initialize primary network Q_ϕ , target network $Q_{\phi_{\text{targ}}}$, replay buffer $\mathcal{D}$, $\rho \ll 1$
for each Iteration **do**:
  for each environment step **do**:
    Observe state s and select $a \sim \pi(a, s)$
    Execute a in the environment
    Observe next state s' , reward r, and done signal d to indicate wether s' is terminal
    Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
    If s' is terminal, reset environment state.
  for each update step **do**
    Randomly sample a batch of transitions, $B = (s, a, r, s', d)$ from $\mathcal{D}$
    Compute targets

$$Q^*(s, a) \approx r + \gamma(1 - d)Q_\phi(s', \underset{a'}{\text{argmax}}\, Q_{\phi_{\text{targ}}}(s', a'))$$

    Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - Q^*(s, a)\right)^2$$

    Update target network with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$

7.2 Results

We used DDQN algorithm on flatland environment using tree observation. Although, it is reasonable to assume that an observation of the full environment is beneficiary for good global solutions. it used to work only on small toy examples (even these are hard to train). However, we run into problems when scalability and flexibility become an important factor. Already on small toy examples we could show that flexibility quickly becomes an issue when the problem instances differ too much. When scaling the problem instances the decision performance of the algorithm diminishes and re-training becomes necessary. On real life instances, we do not believe that a global observation is suited for this problem. So we used DDQN algorithm only on using tree observation.

In order to use the tree observation, we flat out the tree creating a linear vector whose size depends on tree depth and number of features in each node. Since increasing the tree depth increases the size of vector exponentially, using larger tree depth (of the order of 5,6) can have vector of size of order (10^4 to 10^5). In our case we worked on tree depth 2 and 3. Also,

we include malfunction (making the problem much harder) in the agents, as this will add stochasticity into the system and helps in better training instance.

7.2.1 Single agent

When trained on environment having single agent, the results are very good. This is because, since there is only one agent, the whole environment is stationary with respect to the agent.During the course of training over 2000 episodes on real life example, as the training proceeds agent is able to reach the target within time bound ((3*(env.height + env.width)). Moreover, agent is able to find the shortest path as evident from the figure 7.2. Notice that the underlying railway network and the schedule of trains (starting position and target) keeps on changing during the course of training. Only certaing specifications like number of cities, number of parallel tracks connecting city, number of agents, malfunction rate etc. is fixed.

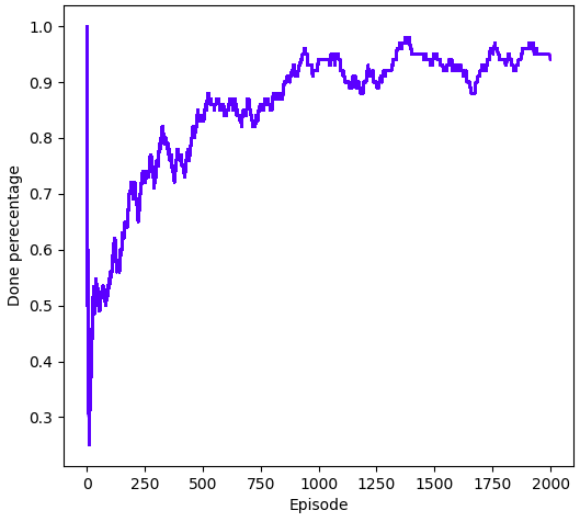

Fig. 7.1 Percentage of agents able to complete journey. Single agent, 5 cities, with 2 rails connecting each city with tree depth 2

7.2.2 Multiple Agent

The Q-learning algorithm assumes that the environment is stationary for the convergence property. Since in case of multiagent environment, environment is non-stationary from the point of view of single agent, so convergence is not guaranteed. Although, it is possible that the agent learns what is best possible action for itself given a particular observation but there is no way, agent can cooperate perfectly.

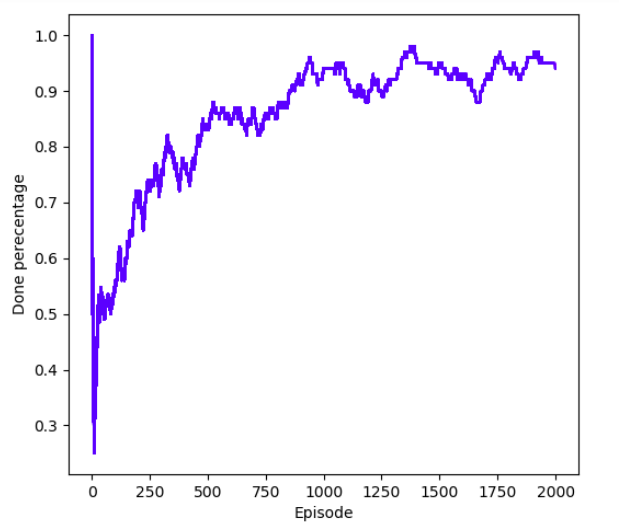

Fig. 7.2 Percentage of agents able to complete journey. Single agent, 5 cities, with 2 rails connecting each city with tree depth 2

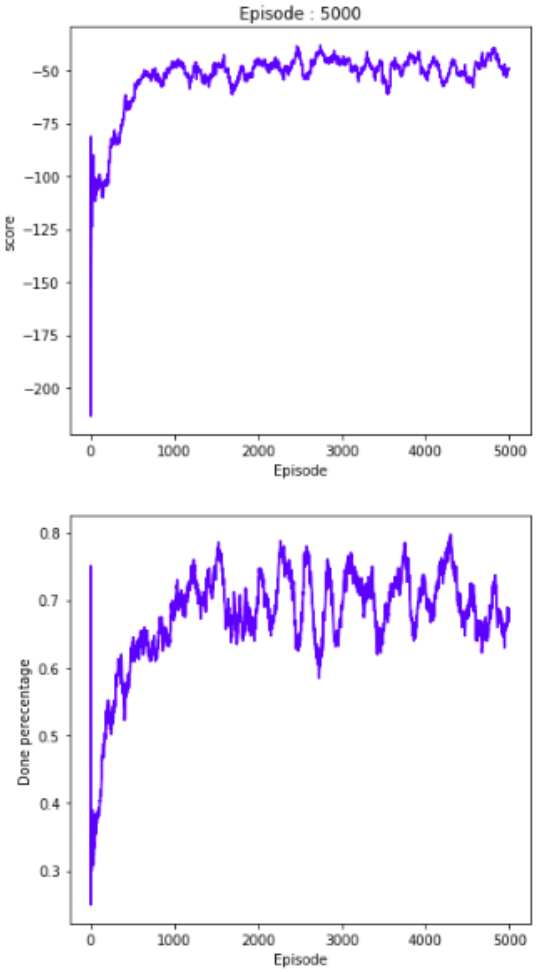

Fig. 7.3 Result of training over real life railway network with 4 agents, 3 cities and tree depth 2 for 5000 episodes. (A) Score of agent, this shows the average reward of agents as the training progresses. (B) Percentage of agents able to complete journey as the training progresses.

The result shows that, 80% of the agents are able to complete there journey using short paths when railway network have 4 agents, 3 cities. So the results are good when the number of agents is small (in this case 4).

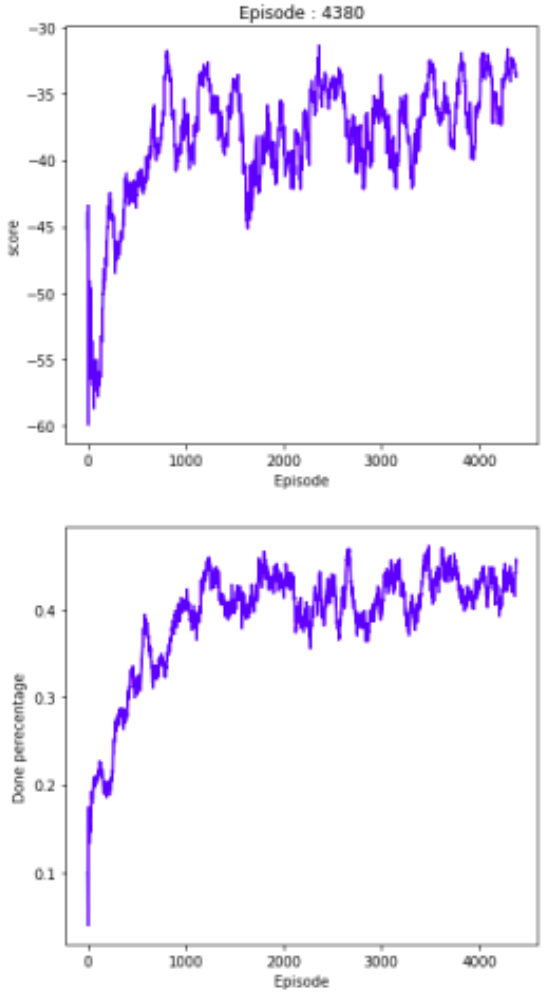

Fig. 7.4 Result of training over real life railway network with 10 agents, 3 cities and tree depth 2. (A) Score of agent, this shows the average reward of agents as the training progresses. (B) Percentage of agents able to complete journey as the training progresses.

So as the number of agents increases to 10, with everything else same, only 40% of the agents are able to complete there journey. The results are even worse, when the number of agents increases more.

7.3 Alternate RL algorithms

Q-learning algorithm is designed having the assumption that the environment is stationary with respect to the agent. But in case of multiagent environment, environmnent is non stationary with respect to the agent, violating Markov assumptions required for convergence of

Q-learning. There are RL algorithms for multiagent environment. One of the most effective is **Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments (MADDPG)[16]** based on actor critic based approach for single agent. In the next chapter, we will solve the same problem using cooperative path finding.

Cooperative Pathfinding

In this chapter, we will try to solve the problem using **Cooperative pathfinding**[6]. There are some drawbacks related to this approach that we will see at the end of chapter and ways to resolve it. When a single unit pathfinds through a map, basic A* search is perfectly adequate. But when multiple units are moving at the same time, this approach can break down, often with frustrating consequences for the player.

8.1 The problem with A*

A* search is used to find the shortest path to the destination for each unit (agent in case of flatland environment). This search ignores the presence of other units, or perhaps treats them as stationary obstacles. If a collision is imminent, the units involved will re-search and select a new path.

In case of **cooperative pathfinding**, A* search is still used, but in a way that takes account of other agents' movements. The only requirement is that agents can communicate their planned paths which is always possible in case of flatland environment. This is generally most appropriate for agents on the same side; agents units will not usually be so cooperative. Non-cooperative pathfinding raises the tricky issue of path prediction, and isn't discussed here as it is not applicable to our flatland problem.

8.2 The Fourth Dimension

To tackle the cooperative pathfinding problem, the search algorithm needs to have full knowledge of both obstacles and agents. However, when agents move around there is no satisfactory way to represent their routes on a stationary map. To overcome this problem, we extend the map to include a fourth dimension: time. We will call the original map the space map and the new, extended map the space-time map. The **space map** consists of a three-dimensional grid of locations and orientation (x, y,orientation). The **space-time** map consists of a four-dimensional grid of cells: Cell(x, y, orientation, t).

In flatland environment, agent have 5 actions: do nothing, deviate left (if possible), go forward, deviate right (if possible), stop. Executing forward (when the agent is facing North) corresponds to moving a agent through the space-time map from Cell(x, y, North , t) to Cell(x, y + 1, North , t + 1).All five actions have a cost of one, corresponding to their duration. Once the agent reaches the target, agent will disappear. A* search can now be used on the space-time map. The goal of the agent is to reach the destination at any time. A* will find the route that achieves this goal with the lowest cost.

8.3 Reservation Table

Once a agent has chosen a path, it needs to make sure that other agents know to avoid the cells along its path. This is achieved by marking each agent into a **reservation table**. This is a straightforward data structure containing an entry for every cell of the space-time map **excluding the orientation**. Each entry specifies whether the corresponding cell is available or reserved. Once an entry is reserved, it is illegal for any other agent to move into that cell. The reservation acts like a transient obstacle, blocking off a location for a single time-step in the future.

Using a reservation table and a space-time map, we are able to solve the cooperative pathfinding problem. Each agent pathfinds to its destination using space-time A*, and then marks the path into the reservation table. Subsequent agents will avoid any reserved cells, giving exactly the coordinated behavior that we desire. Note, a cell is reserved when some location is occupied at some time. Orientation does not have any role here.

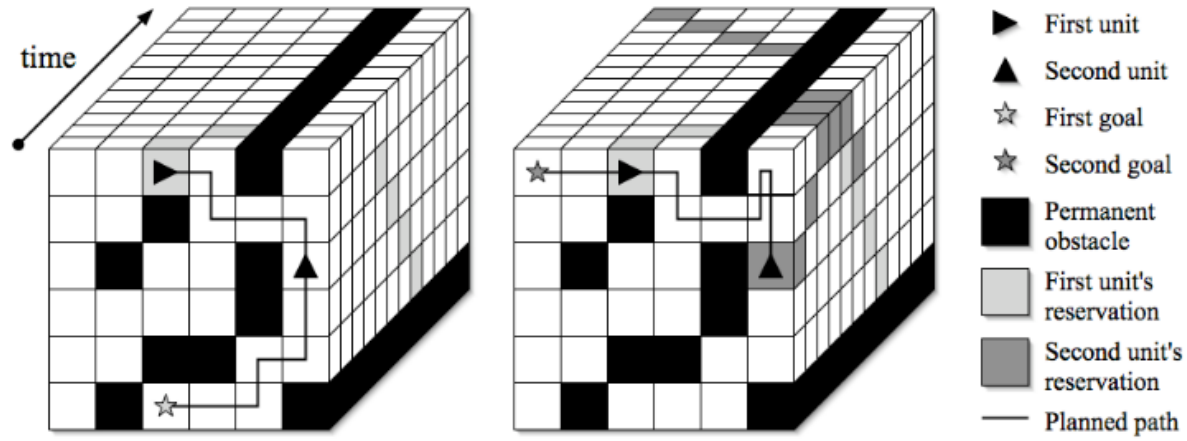

Fig. 8.1 Two agents pathfinding cooperatively. (A) The first agent searches for a path and marks it into the reservation table. (B) The second agent searches for a path, taking account of existing reservations, and also marks it into the reservation table. Orientation is not shown in this example.[6]

We are reserving one cell in the reservation table for the agent. But it can lead to deadlock when executing. Unfortunately, this way of using the reservation table doesn't prevent two agents crossing through each other, head to head. If one agent has reserved (x, y, t) and (x + 1, y, t + 1), there is nothing to stop a second agent from reserving (x + 1, y, t) and (x, y, t + 1). Note orientation is not used in case of reservation table. This problem can be avoided by making **two reservations** for each location involved in the action, one at time t and one at time t + 1. Alternatively, head to head collisions can be explicitly identified and marked as illegal actions. We will be making two reservations for each location in our work as it is able to avoid the deadlock completely. We now have a complete algorithm for cooperative pathfinding.

8.4 Choosing heuristic

The performance of A* depends upon the choice of heuristic. With the search space extended by an extra dimension, the choice of heuristic becomes even more important.

8.4.1 Manhattan distance heuristic

For grid-based maps, the Manhattan distance is often used as a heuristic. It is simply the sum of the x and y distances to the destination. It provides a good estimate of the time to reach the destination on an open map. However, if the shortest path to the destination is circuitous, then the Manhattan distance becomes a poor estimate.

During A* search, new locations are kept on the open list and explored locations are kept on the closed list. At each step, the most promising location is selected from the open list according to its f value. The f value estimates the total distance to the destination, passing through that location. It is the sum of g, the distance traveled by A* to reach the location, and h, the heuristic distance from the location to the destination.
Using the Manhattan distance heuristic, many locations on the map can have an f value that is less than the true distance to the destination. For example, the map in given figure (not showing orientation dimension to make things simple) shows the f values for all locations visited by spatial A*. Almost the entire map has been explored before the destination is found, a phenomenon known as **flooding**.

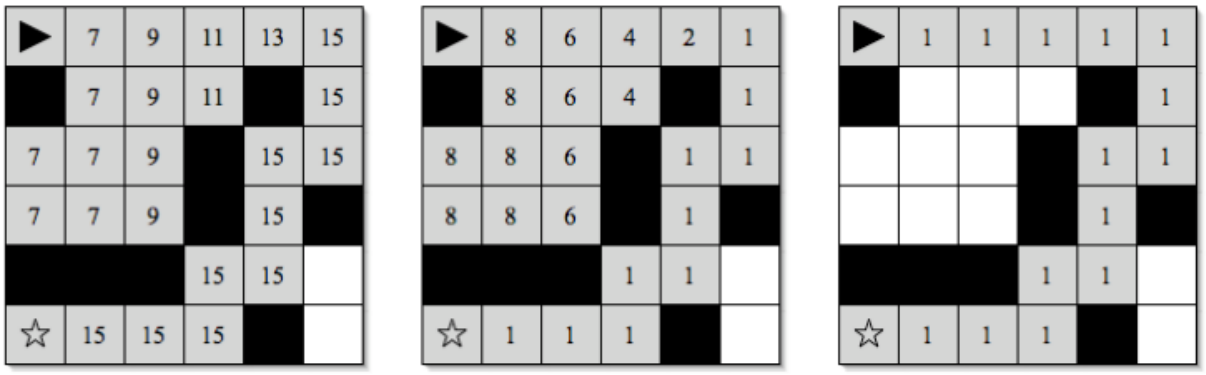

Fig. 8.2 The Manhattan distance heuristic can be inefficient. (A) Flooding occurs in spatial A* when the f values (shown) at many locations are lower than at the destination. (B) The number of visits to each location (shown) can be high during space-time A*. (C) Using the true distance heuristic, the number of visits to each location (shown) is optimal.[6]

Now consider what happens if we use the Manhattan distance in a space-time map. Again, it will work well on an open map. But take a look at the equivalent map in Figure, which shows the number of times each location is explored by space-time A*. Not only will space-time A* explore many more locations at the time of each deviation, it will explore those locations at

later times too. This is because pausing, or returning to a previous location, appears more promising than moving away from the destination. In other words, **the problem has been magnified many times**.

8.4.2 True distance heuristic

We will use true distance heuristic as our heuristic for space-time search. True distance heusristic is the shortest distance to the destination, taking account of obstacles, but ignoring agents. True distance from each location and orientation to each target is pre calculated and stored to make the algorithm faster. When using true distance heuristic in case of cooperative pathfinding, the algorithm is known as **Hierarchical Cooperative A***

Consistency

An admissible heuristic never overestimates the distance to the goal. A* search with an admissible heuristic is guaranteed to find the shortest path. However, there is a stronger property, known as consistency. A **consistent heuristic maintains the property h(A) $\leq$ cost(A, B) + h(B) between all locations A and B**. In other words, the estimated distance doesn't jump around between the locations along a path. Both manhattan and true distance heuristic are consistent.

8.5 Results

We implemented the cooperative A* (both manhattan and true distance heuristic) on flatland environment having both only one cell reserved (at time t) and two cell reserved (at time t and t+1).

8.5.1 Cooperative A* with only cell reserved at t

When only one cell is reserved at time t, the results are very bad as **most of the agents ended in deadlock**. In case of problem instance 1, only 2 out of 10 agents completed their journey. Rest ended in deadlock. In case of problem instance 2, 13 out of 20 agents completed their journey. This is still high because problem instance 2, have multiple paths from each location and orientation to there target. In case of problem instance 3 (real life example), only 6 out of 100 agents are able to complete journey. Clearly, this algorithm is not good.

8.5.2 Cooperative A* with cell reserved at t and t+1

The results in this case are very good, as all the agents are able to complete their journey with great cooperation. I am attaching a video showing the great cooperation between agents. One thing to notice is **path length** for each agent after they complete their journey. In case of problem instance 1, agents 6 and 8 take much larger path compared to shortest path. In case of problem instance 2, since there are so many paths from given location and orientation to the target, actual path taken by the agent and shortest path are almost of

same length. In case of problem instance 3 (real life example), as the agent number increases they take longer and longer routes. This is because the agent with less order number have already made reservations and agent with higher order number have to take alternate path other than the best path. In fact the extent of deviation from the shortest path shows the cooperation between the agents. So, with agent with less order number have **higher priority**. So we can assign priority to the trains based on there order number. Although this is not very good.

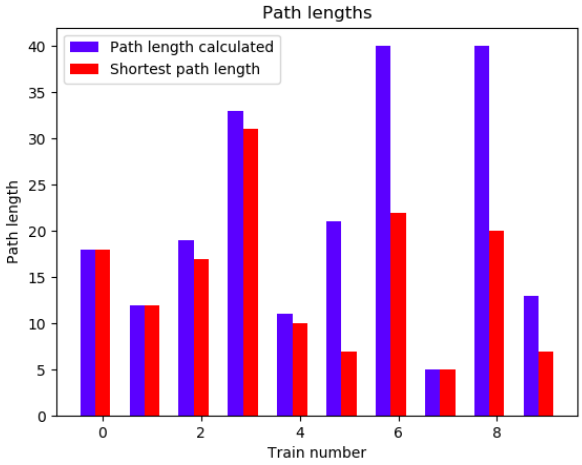

Fig. 8.3 Pathlength (problem instance 1)

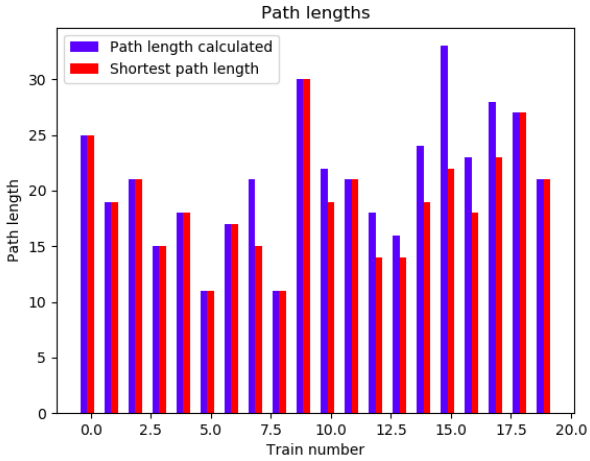

Fig. 8.4 Pathlength (problem instance 2)

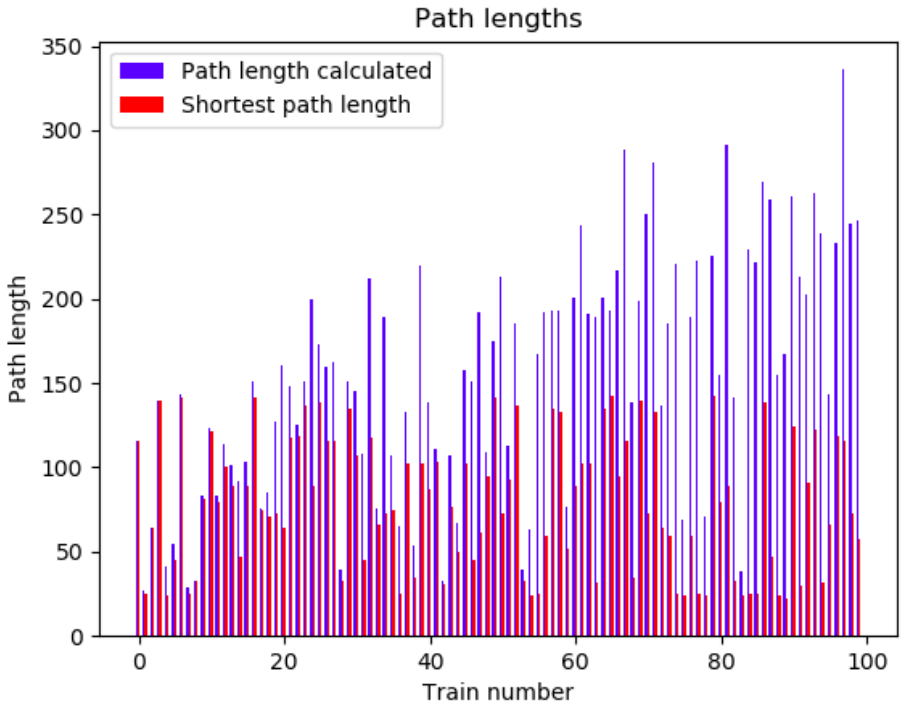

Fig. 8.5 Pathlength (problem instance 1)

8.6 Drawbacks

Hierarchical cooperative A* works really good on flatland environment, but this algorithm is deterministic. This mean, if we include stochasticity into the system, by making agent malfunction, this algorithm is likely to break. To solve this issue, instead of finding the path upto the target, agent can find path upto certain depth and then once completed that depth, can resume search. This will be more robust to stochasticity in environment. This algorithm is called **Windowed Hierarchical cooperative A* (WHCA)**. Also, in Hierarchical cooperative A*, priority is already introduced depending on the agent order. This can be resolved searching the path upto fixed depth, follow the path and then change order. This way each agent will have higher priority at some point of time and hence resolving the issue. We can also try to combine this algorithm with some RL algorithm to make it more robust to the environment with high stochasticity.

Conclusion and Future Work

This report discusses various RL based approaches for solving the railway scheduling problem. In the first half of the report, we worked on solving the railway scheduling problem on a railway line. In order to do that, we implemented our own discrete event simulator (although simulator can also be used in case of railway netowrk) that drives the algorithm. For implementing the RL algorithms to learn schedules, we need a discrete event simulator that drives the algorithm. First phase of the BTP, focuses on implementing the simulator and understanding the prior work in detail. The second phase is focused more on the implementation of the algorithm and experiments.

The algorithm for solving railway scheduling problem on railway line, treats each train as a single agent, So whole system i.e. whole network and trains, is a multiagent environment. First the algorithm discusses about the local state space of each train , actions and policy (ϵ - greedy policy) and the objective function used in this study. Next we discuss about the Sarsa(λ) algorithm with reward as the negative of the objective function. This algorithm does not perform well, since the reward is at the end of the episode, and back-propagation of reward through the trajectory is not possible. So we defined **proxy reward** which captures the probability of state-action pair to end up in a successful episode. Using the proxy reward, we defined Q-values in two different ways. First one looks at success probability of current state-action pair and success probability of its neighbors. Second one looks all the way down to trajectory instead of two step reward functions. Second definition of Q-value, gives better results. The algorithms are tested on three Hypothetical datasets We also see how transfer learning can be used in this case.

Rest of the report focuses on solving flatlant environment which is grid based simulator for multi-agent reinforcement learning for any re-scheduling problem (RSP). First, we used DDQN (double deep Q-learning) using tree observation on flatland environment. Although the algorithm performs good when the number of agents are low, but as the number of agents increases, the results becomes worse. This is bacause, in case of multiagent environment, environmnent is non stationary with respect to the agent, violating Markov assumptions required for convergence of Q-learning. Next we worked on cooperative A* path finding to solve flatland environment, which gives very good results. Only drawback is, it works in non-stochastic environment (i.e. agents should not malfunction and there should not be external delays). However, it can be improved using windowed hierarichal cooperative path finding that uses A* upto certain depth, implement the path so far and then calculate further.

Flatland environment is a very new environment and there is very little development in multiagent RL for re-scheduling problem. Although cooperative pathfinding, gives a good starting point and using RL will further improve the results.

References

[1] H. KHADILKAR, "Artificial intelligence for indian railways," https://www.ai4bharat.org/articles/railways, June 2019.

[2] H. Khadilkar, "A scalable reinforcement learning algorithm for scheduling railway lines," *IEEE*, 2018.

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction*. MIT Press, 1992.

[4] http://flatland-rl-docs.s3-website.eu-central-1.amazonaws.com/04_specifications_toc.html.

[5] https://github.com/arpit23697/Railway-Scheduling-using-RL-BTP-.

[6] D. silver, "Cooperative pathfinding," 2005.

[7] E. Nygren and G. Spigler, "Flatland challenge: Multi agent reinforcement learning on trains," 2019.

[8] http://www.opentrack.ch/opentrack/opentrack_e/opentrack_e.html.

[9] https://www.railml.org/en/.

[10] https://networkx.github.io/.

[11] https://simpy.readthedocs.io/en/latest/contents.html.

[12] H. Khadilkar, "Scheduling of vehicle movement in resource-constrained transportation networks using a capacity-aware heuristic," *Proc. Amer. Control Conf.*, May 2017.

[13] D. Šemrov, R. Marsetič, M. Zura, and L. Todorovski, "Reinforcement learning approach for train rescheduling on a single-track railway," 2016.

[14] D. Silver and A. Graves, "Playing atari with deep reinforcement learning," *Nature*, Dec 2013.

[15] D. Silver, "Human-level control through deep reinforcement learning," *Nature*, Feb 2015.

[16] R. Lowe, Y. Wu, and A. Tamar, "Multi-agent actor-critic for mixed cooperative-competitive environments," Mar 2020.