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 lines (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. The primary advantage of the proposed algorithm compared to exact approaches is its scalability, and compared to heuristic approaches is its solution quality.Improved solution quality is obtained because of the inherent adaptability of reinforcement learning to specific problem instances.

This report presents the problem statement, discusses about the approaches, implementation and future work.

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.

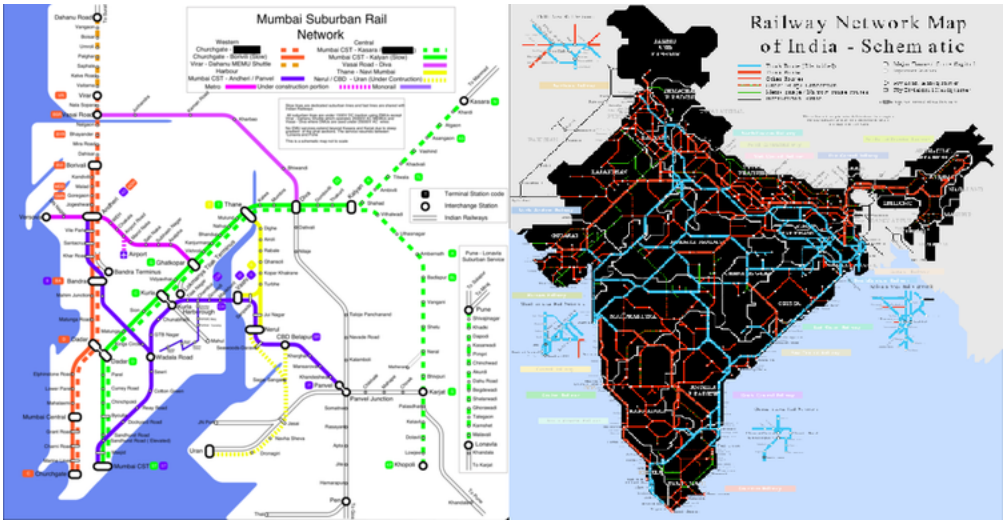

Fig. 2.1 The line and junction topology of railway networks in India [1].

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.

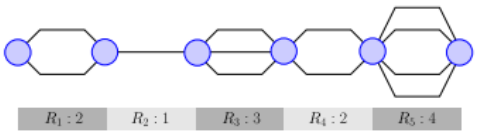

Fig. 2.2 Linear Railway Lines [2].

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.

Reinforcement Learning Approach

3.1 Brief Description

Reinforcement learning works by learning to map the state of the system to the choice of available actions based on long-term reward (or penalty). In this report we will discuss review in brief the prior work [2] and then discuss the work that will be done as part of the project. The key difference in both the the approaches is:

- In the prior approach, we find state space with respect to train and it is the train that is making the decision (So we have the environment where each train is acting as the agent i.e. multiagent environment)
- In the proposed work, we have the central controller as the single agent and it is that agent that is scheduling the trains.

3.2 Prior Work

3.2.1 State Representation (Local state space)

Here, 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. We have one more entry in the state space for the priority of trains. If we assume that the model accommodates up to P priority levels,

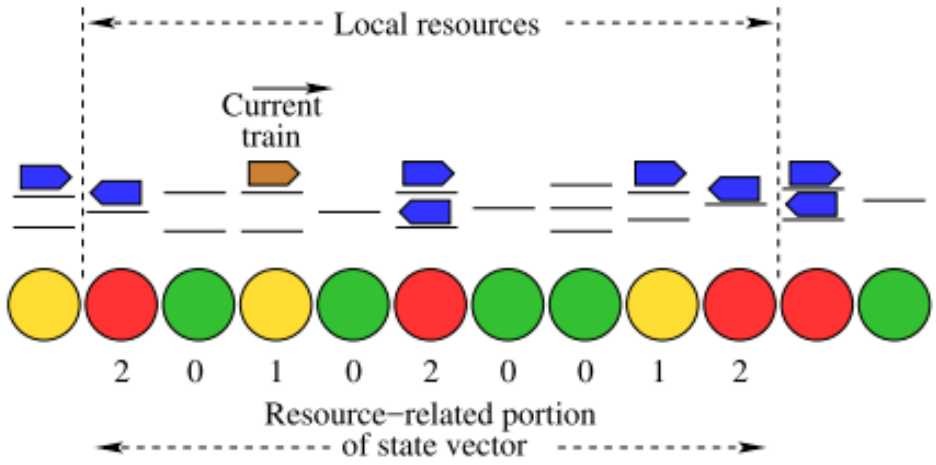

Fig. 3.1 Mapping train location and direction of movement to resource status, relative to the 'current train' [2].

the size of the state space is equal to $P \cdot R^{l_b+1+l_f}$. 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.

3.2.2 Action and Policy Definition

In this approach, state space is computed for each train locally. 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 policy that can be used to take the action is ϵ-greedy with the value of ϵ-decresing over the training. It comprise of three steps

- Select the train on which to take the action.
- Compute the state space for the train.
- Choose the action depending on the ϵ-greedy policy.

3.3 Proposed Approach

3.3.1 State Space Representation

We are planning to use the whole network along with the positions of each train to take as the state space. The key idea is to let the RL algorithm find which part of the state space is important to make decision. In the prior approach, we are kind of tuning how far the train can see and it's action depends only on local neighborhood but in our work we are going to

include everything in the state space. Since, including everything can blow the size of the state space really quickly, so to tackle with this problem we can use function approximation methods (Deep Q-Learning) to learn the state space and then work on it.

3.3.2 Action And Policy Definition

In our proposed approach, we take the whole network topology and the position of the trains as the state space. So we have the central controller and that central controller will take the action for each train depending on state space that takes everything into account. At a particular time during the simulation, let's say we have n number of trains waiting for the action (ready to move or stop), then the controller have the action space of 2^n and it has to make decision for each of the trains (although in some order, as we are running the simulator and taking action for one train at a time).Size of the action space depends on the number of trains waiting for the action to be taken. Here again we can use the ϵ-greedy policy for exploration in the initial phase and then further exploitation.

3.4 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.
Prior work uses priority weighted delay as the objective function. In our proposed approach as well, we can shape the reward functions as to use the same objective function.

Implementation Overview

As for now, we are focussing on how to implement the first approach and then move onto the second approach. The integrated reinforcement learning algorithm is driven by a discrete event simulator. There are already some railway simulators like **OpenTrack** [3] and **RailML**[4] 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.

4.1 Railway simulator

4.1.1 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.

4.1.2 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**[5] 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**[6] 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.

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 [7]

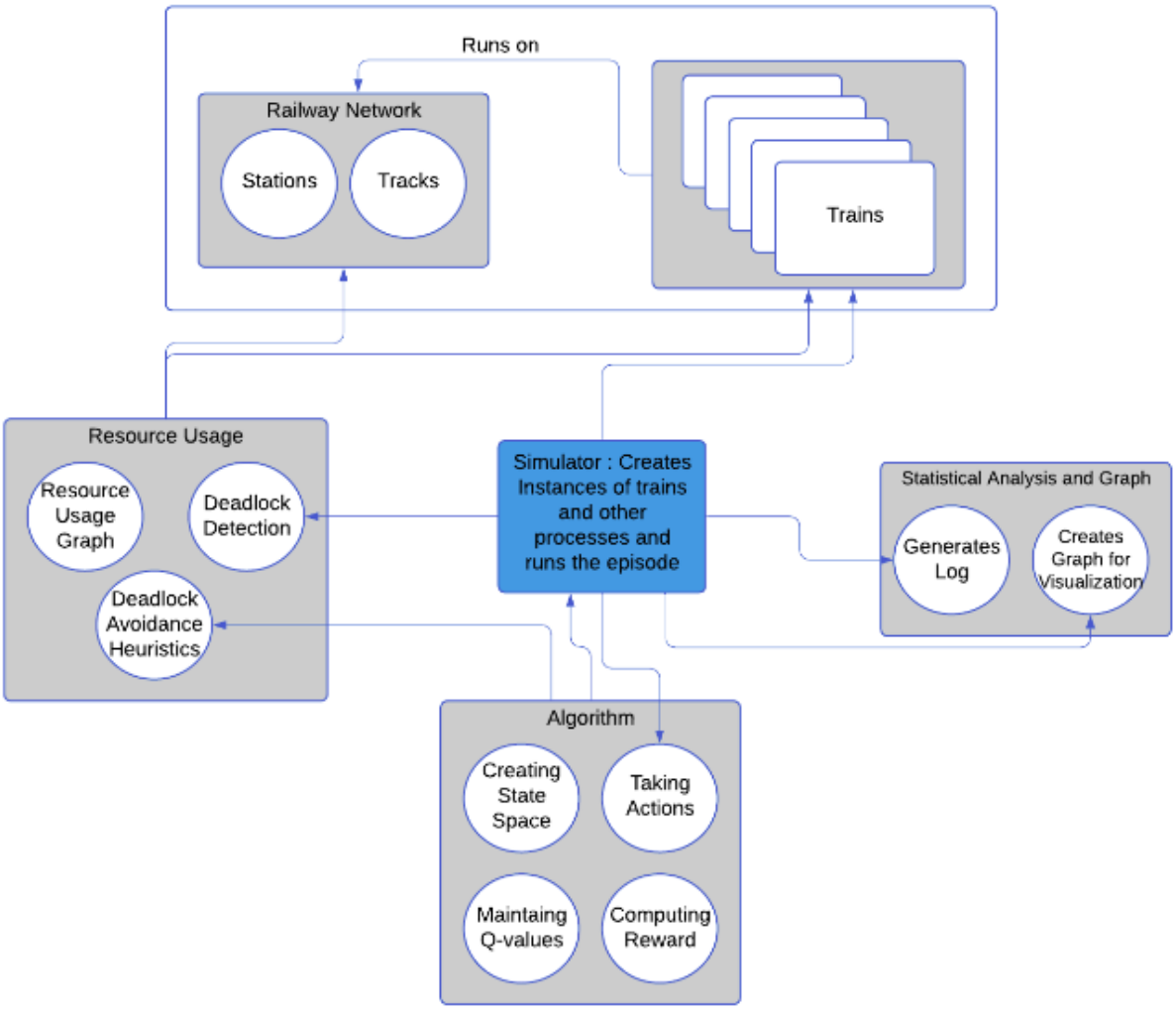

Fig. 5.1 Flow chart summarizing the implementation of railway simulator.

5.1 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.

   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 [8]. 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.

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**.

6.1 Mathematical model

This mathematical model is from [8]. 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,

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

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

6.2 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.

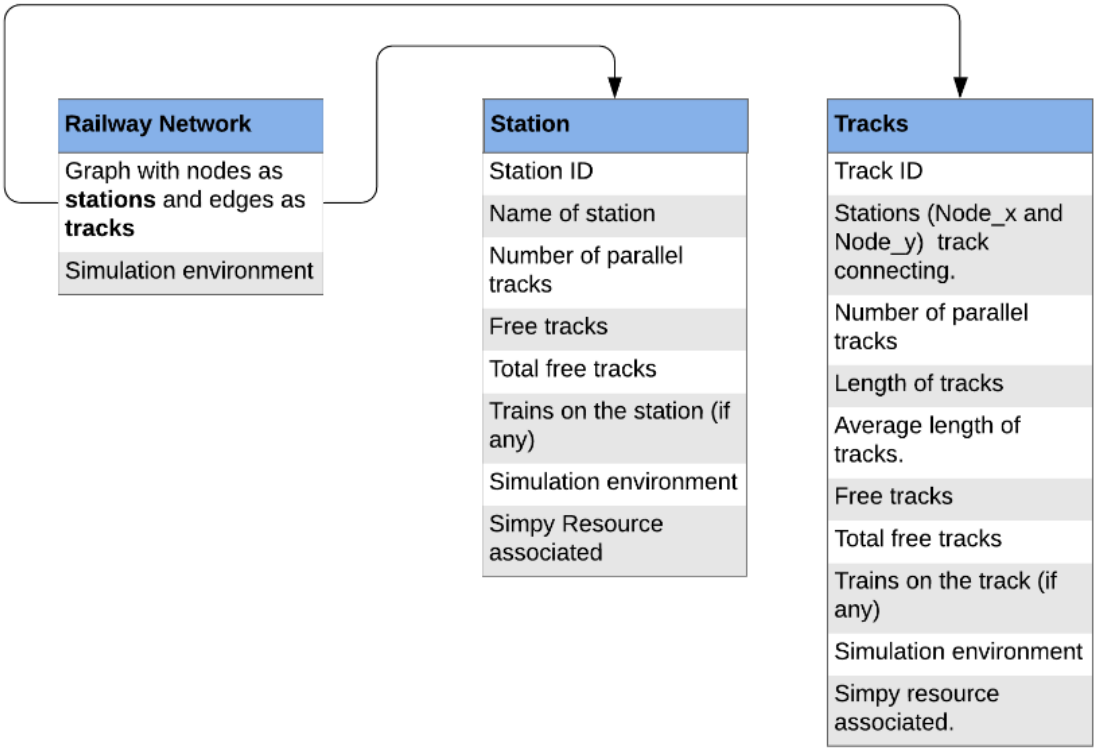

Fig. 6.1 Railway Network framework

6.3 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. 6.2 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.

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.

7.1 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**.

7.2 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.

7.2.1 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.

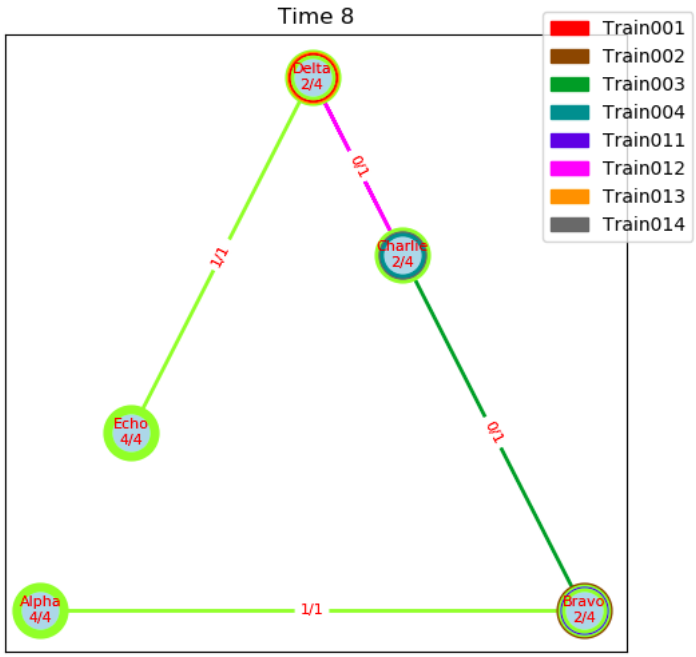

Fig. 7.1 Network with trains color coded.

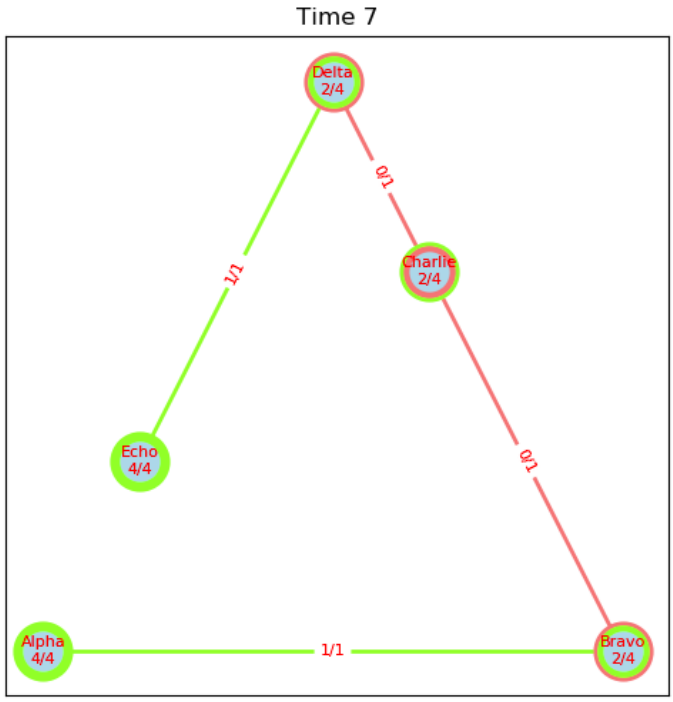

Fig. 7.2 Network showing just what all resources are free.

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).

8.1 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.

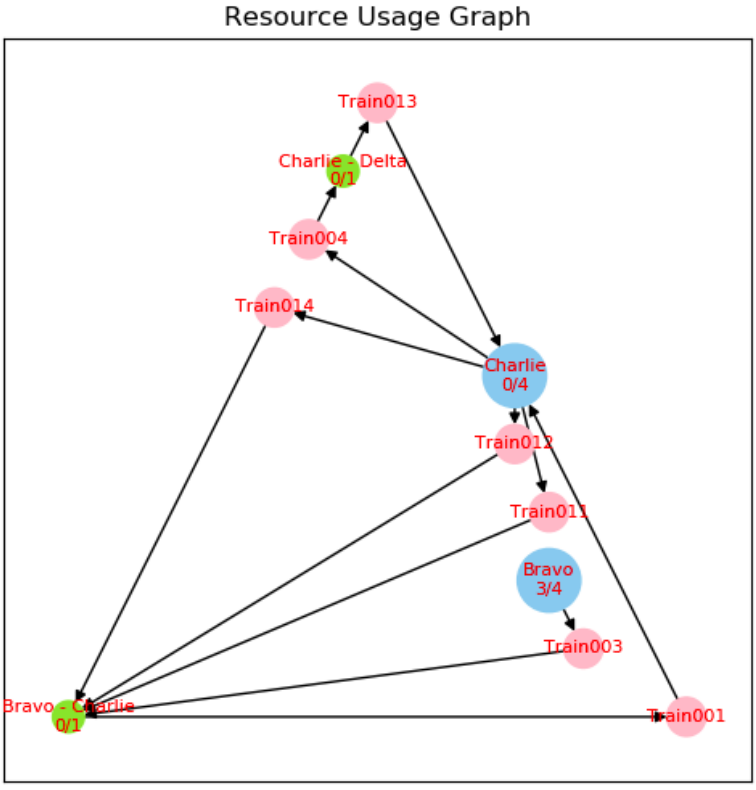

Fig. 8.1 Resource usage in the network.

8.2 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.

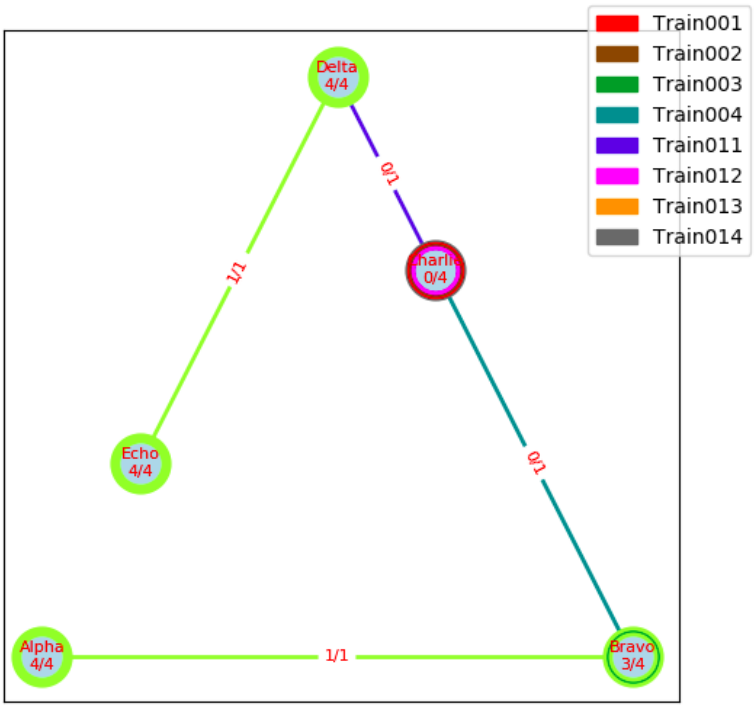

Fig. 8.2 Deadlock near station Charlie.

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

8.2.1 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:

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.

8.3 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 [8]. 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.

8.3.1 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.

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 [8] 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 proposed 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.

In this approach, 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 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.

10.1 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.

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.

10.2 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.

10.2.1 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.

10.2.2 Modified ϵ - greedy policy

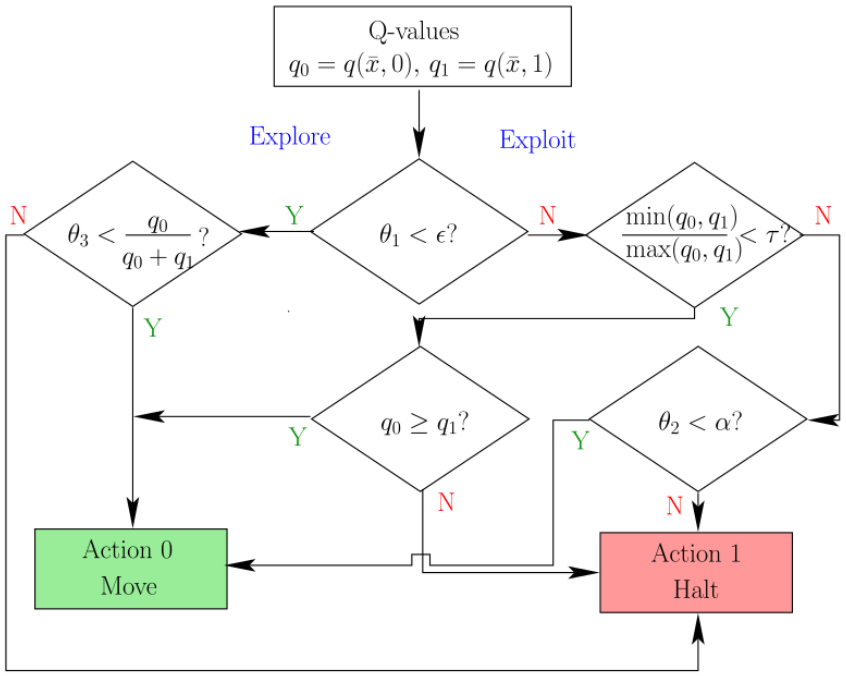

Fig. 10.1 Modified greedy policy

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.

10.3 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.

10.4 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.

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**).

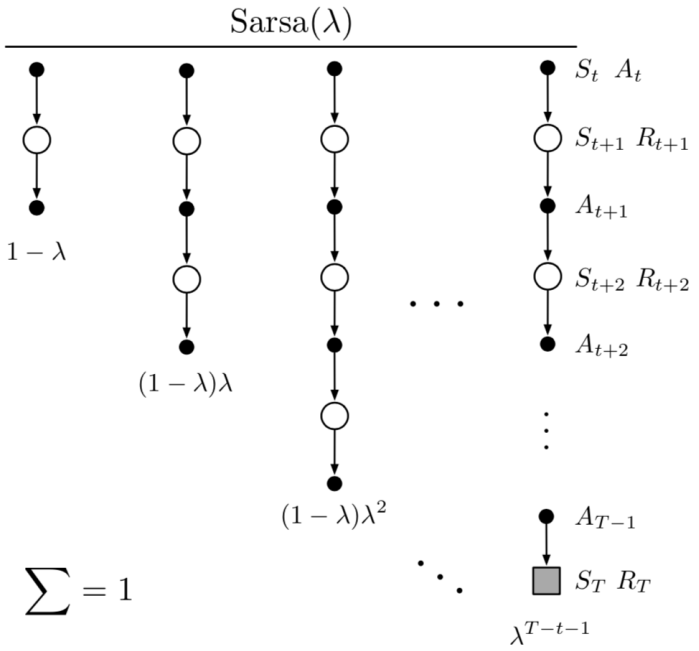

Fig. 10.2 Sarsa(λ)'s backup diagram

Algorithm 1 Sarsa Lambda

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

10.5 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**.
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 the weighing factor, (x'_m, a'_m) are the neighbors (say M) of (x_m, a_m) during an episode.

Conclusion and Future Work

For implementing the RL algorithms to learn schedules, we need a discrete event simulator that drives the algorithm. So far, focus is to implement the simulator. Simulator is organised into different modules and these modules work together to create the whole system. Railway network and train module implements the underlying railway network and trains that run over these networks. Statistical analysis and graph module creates graphs for visualisation and generates log for analysis and debugging. More components will be added in future for more analysis as per the need. Resource usage module keeps track of the resources (station and track) in the network. It generates Resource usage graph, implements deadlock detection algorithm for detecting deadlock in the network and deadlock avoidance heuristic for choosing action when multiple trains need action at the same time. Algorithm module forms the backbone of the whole simulator and contains implementation of RL algorithms (Q-learning and Deep Q-Learning) for learning the schedule. Finally simulator module, puts each piece together by creating each important component as process. Each train in the network is treated as process and runs simultaneously. Other processes are also created for detecting deadlock, generating graph for visualisation, checking status of the network (number of trains in the network and there status) and choosing action for trains that need action.

Now the future focus is on the algorithm module that will contain implementation of prior as well as proposed approach. The report discusses two approaches, to solve the problem. Once the prior approach is implemented, we can see how good it is working, how good it is scaling to real life problem instances, how good it is performing compared to the present approaches and how to improve upon it. Next step is to improve on this algorithm and add deep learning to it for learning the state space. Finally, we drive towards the proposed approach and improve upon it.

Since we are tackling blocking version of the JSSP problem, so the approach that we will develop can be used to solve the JSSP problem with reasonable approximation. So the future plan is also to use the developed approaches on other similar problems as well.

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

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

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

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

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

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