WABTEC EXCEED 3.0

AUTOMATED TRAIN SCHEDULING AND ROUTING

Ravi Sharma – MNNIT Allahabad

Priyansh Agrawal- MNNIT Allahabad

Lakshya Kumar – MNNIT Allahabad

WABTEC EXCEED 3.0

Contents:

Problem Statement:

Our Project, Rail automation System, addresses the challenge of **automated train scheduling and routing**. The goal is to design a system capable of optimizing train schedules and routes using reinforcement learning (RL) while considering multiple real-world constraints and disruptions.

Some Problem Elements

1. **Key Constraints:**
    a. **Passenger Demand:** Adjusting schedules based on real-time and predicted passenger needs.
    b. **Track Availability:** Accounting for occupied, free, or under-maintenance tracks.
    c. **Weather and Disruptions:** Managing unforeseen conditions such as extreme weather or accidents.
    d. **Safety and Headway:** Ensuring safe distances between trains.
2. **Optimization Objectives:**
    a. Minimize delays and improve on-time performance.
    b. Maximize passenger satisfaction by reducing wait times and improving seat availability.
    c. Enhance resource efficiency by optimizing energy/fuel usage and operational costs.
3. **Adaptive Challenges:**
    a. Handle real-time disruptions like track closures or delays.
    b. Scale effectively to manage thousands of trains and stations.
    c. Ensure compliance with safety regulations and infrastructure limits.

Our First Approach:

Link:- https://www.kaggle.com/code/priyanshagrawal9893/notebookf978d235fd

Our project incorporates **rule-based methods** as a fallback mechanism or for ensuring compliance with fundamental operational constraints, such as safety, headway, and route prioritization. These methods serve as a complementary system alongside the primary reinforcement learning framework to ensure robust and reliable decision-making under various conditions.

1. Static Rules for Constraints

Static rules are predefined, unchanging conditions designed to enforce safety, efficiency, and operational stability in train scheduling and routing. These include:

1. **Headway Rules:**
    - Ensures a minimum time or distance between consecutive trains to maintain safety and prevent collisions. This constraint is critical in avoiding congestion and enabling smooth traffic flow on shared tracks.
2. **Capacity Rules:**
    - Restricts the number of trains occupying a particular track segment or siding based on infrastructure limitations. This prevents overloading and ensures optimal resource utilization across the rail network.
3. **Prioritization:**
    - Assigns higher priority to certain categories of trains, such as express services, delayed trains, or those carrying critical cargo. Prioritization enhances service quality and minimizes delays for high-priority operations.
4. **Maintenance Windows:**
    - Blocks specific tracks during scheduled maintenance or infrastructure upgrades, ensuring operational safety and long-term network reliability.

2. Dynamic Rules for Real-Time Adjustments

Dynamic rules adapt to real-time conditions and disruptions, offering flexibility to maintain network efficiency under varying circumstances. Examples include:

1. **Route Redirection:**
    - Redirects train routes in response to unexpected delays, track closures, or operational bottlenecks. This adjustment minimizes disruptions and maintains the flow of traffic.

2. **Speed Adjustments:**
    - Modifies train speeds to ensure smooth connections during peak demand periods. Such adjustments help synchronize schedules and reduce passenger wait times.

3. **Safe Rerouting:**
    - Implements contingency plans for weather-related disruptions, accidents, or emergencies, ensuring safety while maintaining operational continuity.

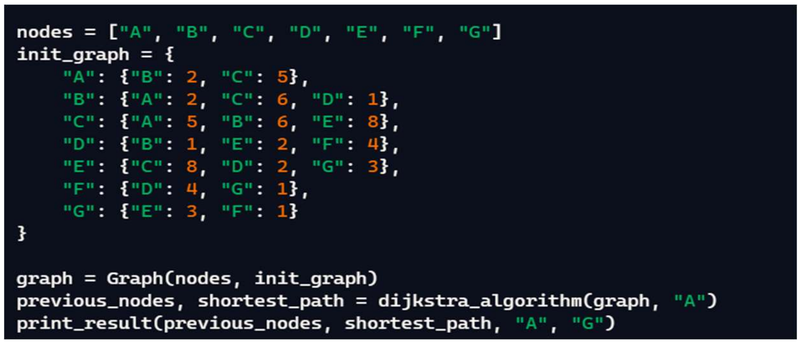
Dijkstra Algorithm

Graph Class

- **Constructor**: Initializes the graph with nodes and initial graph data.
- **Static Method** __construct_graph: Ensures the graph is symmetrical, updating paths to be bidirectional.
- **Methods**:
    - get_nodes(): Returns the list of nodes.
    - get_outgoing_edges(node): Returns neighboring nodes of a given node.
    - value(node1, node2): Returns the weight of the edge between two nodes.

Dijkstra's Algorithm

- **Function** dijkstra_algorithm(graph, start_node):
    - Initializes unvisited nodes, shortest path, and previous nodes dictionaries.
    - Uses a priority queue (heapq) to manage and update node distances efficiently.
    - Iteratively finds the node with the smallest tentative distance, updates its neighbors' distances, and keeps track of the shortest paths.



```
nodes = ["A", "B", "C", "D", "E", "F", "G"]
init_graph = {
    "A": {"B": 2, "C": 5},
    "B": {"A": 2, "C": 6, "D": 1},
    "C": {"A": 5, "B": 6, "E": 8},
    "D": {"B": 1, "E": 2, "F": 4},
    "E": {"C": 8, "D": 2, "G": 3},
    "F": {"D": 4, "G": 1},
    "G": {"E": 3, "F": 1}
}

graph = Graph(nodes, init_graph)
previous_nodes, shortest_path = dijkstra_algorithm(graph, "A")
print_result(previous_nodes, shortest_path, "A", "G")
```

Objects

Station: The Station class manages trains, passengers, and connections in a transportation system. It includes attributes for station ID, capacity, and lists of trains, passengers, and lines. Key methods allow for retrieving and updating these attributes, handling train arrivals/departures, and managing passenger movements.

The Train class manages train operations, passenger handling, and routing within a transportation system. It includes attributes for train ID, capacity, speed, and location. Key methods handle passenger boarding, setting routes using Dijkstra's algorithm, and managing train movements between stations and lines.

- **Passenger Management**:
  passenger_boards(passenger_id: str, passenger_group_size: int): Adds a passenger to the train and updates utilization.
  passenger_detrains(passenger_id: str, passenger_group_size: int): Removes a passenger from the train and updates utilization.
  all_passengers_detrain(): Removes all passengers from the train.

- **Time Management**:
  increment_time_on_current_location(): Increases the time spent at the current location by one unit.
- **Duration Calculation**:
  calculate_duration_multiple_routes(end_station_id: str, expected_arrival: int, current_passenger_id: str): Calculates duration and delay differences with a new passenger.
  calculate_duration_current_route(end_station_id: str): Calculates the duration from the current location to a specified destination.
  calculate_duration_fastest_route(end_station_id: str = None, start_station_id: str = None): Finds the fastest route using Dijkstra's algorithm and returns the duration.

The Passenger class manages passenger information in a transportation system.

Key Points:

- **Attributes**: ID, start and end stations, group size, expected arrival, current location, delay.

- **Methods**: Handle boarding (board) and detraining (detrain) with error checks.

It efficiently tracks passenger movements and ensures smooth operations.

Function and Preprocessing

Data Preprocessing and Object Mapping:

- data_preprocessing(data): Cleans and splits data into DataFrames for stations, lines, trains, and passengers.
- create_train_graphs(): Creates graphs for each train to determine travel times.
- map_df_to_objects(station_df, line_df, train_df, passenger_df): Maps DataFrames to lists of objects.

Preparation and Initialization:

- preparation(): Sets up the simulation, sorting passengers and trains, assigning starting stations, and determining maximum rounds.
- **Route Setting and Status Printing**:
- set_route_for_train(train): Determines and sets the route for a train based on passenger needs.
- print_current_situation(): Logs the current status of stations, lines, trains, and passengers.

Capacity Checks and Ending Conditions:

- check_capacities(): Ensures that capacities of trains, stations, and lines are not exceeded.
- check_ending_conditions(passenger_list): Checks if all passengers have reached their destinations.

Output Preparation:

- prepare_output(train_list, passenger_list): Initializes logs for trains and passengers to track actions for output.

Algorithm Preparation:

- preparation(): Sets up the simulation by sorting passengers/trains, assigning starting stations, and calculating maximum rounds.
- prepare_output(train_list, passenger_list): Initializes logs for trains and passengers.

Simulation Management:

- set_route_for_train(train): Determines and sets train routes based on passenger needs.
- print_current_situation(): Logs the current status of all entities.
- check_capacities(): Ensures capacities are not exceeded.
- check_ending_conditions(passenger_list): Verifies all passengers have reached their destinations.

Passenger Handling:

- passenger_boards(self, passenger_id, passenger_group_size): Boards passengers onto trains.
- passenger_detrains(self, passenger_id, passenger_group_size): Detrains passengers from trains.
- all_passengers_detrain(self): Removes all passengers from a train.

Time and Duration Calculation:

- increment_time_on_current_location(self): Increments time spent at the current location.
- calculate_duration_multiple_routes(self, end_station_id, expected_arrival, current_passenger_id): Calculates durations and delays if a new passenger boards.
- calculate_duration_current_route(self, end_station_id): Calculates duration to a specified destination.
- calculate_duration_fastest_route(self, end_station_id, start_station_id): Finds the fastest route using Dijkstra's algorithm.

Reinforcement Model:

Link:- https://www.kaggle.com/code/priyanshagrawal9893/notebookf246696eae

As we have to optimize resources like time complexity and space complexity from previous solution , we have to make use of heuristic functions using reinforcement learning model. So to develop that , we try various approaches of rl model and finally reach to a final solution. In following headings , we have outline steps to reach upto final soltion.

• Single Agent and Environment without Grid

Here we take environment as graph between stations as nodes and routes as edges. Single agent is taken which will be responsible for every train movement in that environment. For any changes in train position due to agent action we assign rewards according to our parameters i.e. passenger demand, availability of tracks, maintainence schedules and weather conditions.

Problems:

1. Each train movement is controlled by other train movement
2. Factor assignement or metric Evaluation is not done efficiently.
3. Without grid , Weather conditions and maintainence schedules metrics are difficult to evaluate as entire route can not have maintain schedules or extreme weather condition But a part of them may have which can not be evaluated.

• Multi Agent and Environment without Grid

Here we take environment as graph between stations as nodes and routes as edges. Mutiple agent is taken one agent for each train. For every time movement each agent performs actions i.e., train movement . Rewards are assigned for that train movement to its agent based on the factors demanded. Once every train reach its destination sum of each agent reward is taken and that is the final reward of that whole scenario of rl model.

Improvement:

1. Each train is independent of other as per normal scenario and factor metrics can be efficiently calculated

Problems:

1. Factor assignement or metric Evaluation is not done efficiently.
2. Without grid , Weather conditions and maintainence schedules metrics are difficult to evaluate as entire route can not have maintain schedules or extreme weather condition But a part of them may have which can not be evaluated.

- **Multi Agent and Environment with Grid**

First Deliverable: Reinforcement learning model for train scheduling optimization

As from above steps, the ideal solution to this problem is multiagent and environment with grid ,as through this solution factor metrics is easily and efficiently evaluated and by Q-learning we finally reach to most efficient solution.

Features and Characteristics of this solution:-

1. Whole railways scenario area is in form of grid, part of area where railway path is there ,it is marked with greater than or equal to 0 and other area is marked with -1.
2. Maintainence area is marked with 2.
3. Extreme weather conditions area is marked with 3.
4. If any agent or train occupies a particular area it is marked with 1 and any other train or agent can not occupy it.
5. For any train movement , reward of -1 is given.
6. If train passes extreme weather condition area, reward of -2 is given.
7. If train passes extreme maintainence condition area, reward of -3 is given.
8. If train reaches its target , reward of +10 is given.
9. There is Q-Learning agent of a train which learns from past experiences and also takes random actions to take new experiences

Hence through this metric evaluation using rewards and learning agent help us to simulate realworld problem of train movement and schedules and solve them.

Enivroment Specification:

1. **Grid Representation:**
   - The environment is a grid (grid_size) where each cell has a specific value:

     -1: Non-railway area.

     0: Railway path.

     2: Maintenance area.

     3: Extreme weather area.

     1: Occupied by a train/agent.

2. **Random Initialization:**
   - Railway paths, maintenance areas, and extreme weather zones are randomly assigned to grid cells during initialization.
   - Agents (trains) are placed at random railway positions with predefined **start** and **target** locations.

3. **Agent Movement and Rewards:**
    - Agents can move in 5 directions (up, down, left, right, wait) based on actions.
    - Rewards are determined by the cell type they move into:
        - -1 (normal path): -1 penalty.
        - 2 (maintenance): -3 penalty.
        - 3 (extreme weather): -2 penalty.
        - Target cell: +10 reward.

4. **Collision Prevention:**
    - Only one agent can occupy a railway cell (1 indicates occupied). Invalid moves (into occupied or non-railway areas) result in a penalty (-5).

5. **Reset Functionality:**
    - The reset() method reinitializes the environment, re-randomizing paths, hazards, and agent positions for new episodes. This ensures variety in scenarios.

Q-learning Agent Specification

1. **Deep Q-Network (DQN):**
    - The agent uses a fully connected neural network with three layers:
      Input layer matching the state size (state_size).
      Two hidden layers (128 and 64 units, both with ReLU activation).
      Output layer matching the number of actions (action_size) to predict Q-values for each action.

2. **Exploration vs. Exploitation:**
    - Implements an **ε-greedy policy**:
    - With probability epsilon, selects a random action (exploration).
    - Otherwise, selects the action with the highest predicted Q-value (exploitation).
    - The exploration rate (epsilon) decays over time (epsilon_decay), balancing exploration and exploitation.

3. **Replay Memory:**
    - Maintains a buffer (self.memory) of past experiences (state, action, reward, next_state, done) for training.
    - Uses random sampling of minibatches during training to reduce correlation between consecutive experiences, stabilizing learning.

4. **Training with Bellman Equation:**
    - Computes the target Q-value using the Bellman equation: Q(state,action)←reward+γ·max (Q(next_state))Q(\text{state}, \text{action}) \leftarrow \text{reward} + \gamma \cdot \max(Q(\text{next_state}))
    - Trains the model by minimizing the Mean Squared Error (MSE) between predicted and target Q-values.

5. **Adaptive Learning:**
   - Uses an **Adam optimizer** to update network weights.
   - Gradually reduces epsilon to ensure the agent shifts from exploration to exploitation as it learns optimal policies.

Final Model Running and Result Calculation

1. **Multi-Agent Setup:**
   - The environment (RailEnvironment) contains multiple agents (n_agents = 5).
   - Each agent is an independent **Deep Q-Learning Agent** (DQLAgent), initialized with its own model and experience buffer.
2. **Independent Decision-Making:**
   - Each agent observes the environment state and selects its action independently using its own policy (act method).
   - Actions for all agents are executed simultaneously, and their rewards are tracked individually.
3. **State Sharing and Updates:**
   - The shared environment state (state) is flattened into a vector for compatibility with the neural network input.
   - After taking actions, agents update their states, rewards, and experiences (remember) based on the feedback from the environment.
4. **Experience Replay for Training:**
   - Each agent uses its experience buffer to train via **mini-batch sampling** during the replay phase.
   - This stabilizes training by reducing correlations between experiences and improves learning efficiency.
5. **Performance Logging and Exploration Decay:**
   - Tracks the total and average reward per episode across all agents.

   - Gradually reduces the exploration rate (epsilon) for each agent, ensuring a balance between exploration and exploitation as agents learn over episodes.

Second Deliverable: Simulation and evaluation of model based on historical performance

1) Passenger demand : we will schedule train between source and target according to passenger demand

2) Track Availability: we will taking care of this using grid environment by assigning 1 if occupied by any other train or agent.

3)Maintainence schedules: we will taking care of this using grid environment by assigning 3 if maintaince is scheduled of any part of grid.

4)Weather conditions: we will taking care of this using grid environment by assigning 2 if any part of grid is having extreme weather conditions.

Third Deliverable: Integration plan for deploying the model in rail network management system

Integration Plan for Deploying the Model in a Rail Network Management System

1. **Requirement Analysis and Stakeholder Collaboration:**
   - Collaborate with key stakeholders (railway operators, IT teams, logistics experts) to define the integration requirements.
   - Identify the data sources (e.g., live GPS data, historical schedules, rail network maps) and system interfaces for real-time updates.

2. ***Data Preparation and Preprocessing Pipeline:***
   - Establish automated pipelines to ingest real-time and historical data into the model.
   - Ensure data standardization, such as encoding geographical data, normalizing distances, and handling missing values.

3. ***Simulation Environment Setup:***
   - Deploy the developed simulation environment as a **sandbox** for testing and validating model performance with real-world scenarios.
   - Include realistic station locations, rail conditions, and operational constraints to mimic real-world conditions.

4. ***Model Optimization and Training:***
   - Train the model on historical data to ensure it learns patterns relevant to the specific rail network.
   - Optimize the model for speed and accuracy, reducing computation time to handle real-time updates.

5. ***System Integration via APIs:***
    - Develop and deploy RESTful APIs to connect the model with the rail network management system.
    - APIs will handle tasks like fetching live train positions, submitting recommended routes, and receiving confirmation of actions.
6. ***Decision-Making Integration:***
    - Integrate the model's outputs (e.g., optimal routes, fuel-efficient schedules) into the existing decision-making dashboard.
    - Provide operators with actionable insights in an intuitive format, such as route visualizations or alerts for deviations.
7. ***Real-Time Processing Capability:***
    - Deploy the model on a cloud-based infrastructure to ensure **real-time inference** capability.
    - Implement load-balancing mechanisms to manage simultaneous requests for large-scale rail networks.
8. ***Validation and Testing:***
    - Test the integrated system with both historical and real-time data to validate its accuracy and reliability.
    - Use key metrics such as **distance reduction, schedule adherence, and computational latency** to measure performance.
9. ***User Training and Support:***
    - Train operators and staff to interpret the model's outputs and integrate its recommendations into daily operations.
    - Provide comprehensive documentation and 24/7 technical support during the transition period.
10. ***Continuous Monitoring and Updates:***
    - Implement a feedback loop to monitor the system's performance post- deployment.
    - Periodically retrain the model with updated data and enhance features based on user feedback and evolving operational needs.

By following this structured plan, the deployment ensures seamless integration, operational efficiency, and measurable improvements in rail network management.

Scalability and Further Improvements

Scalability and Further Improvements

1. **Enhanced Grid Size and Resolution:**
   - **Scalability**: Expand the grid to represent larger regions or the entire railway network of a country. Increase grid resolution to represent areas smaller than 1 km^2 for more precise routing.
2. **Multi-Agent Collaboration:**
   - **Scalability**: Allow coordination among multiple trains (agents) to handle complex scenarios like train crossings, shared tracks, and congestion.
3. **Dynamic Real-Time Updates:**
   - **Scalability**: Integrate real-time data streams, such as live GPS coordinates, weather updates, and unexpected maintenance alerts.
4. **Distributed Computing for Large Networks:**
   - **Scalability**: Use distributed systems like **Apache Spark** or **Ray** to process large-scale simulations for rail networks spanning thousands of kilometers.
5. **Integration with Advanced Technologies:**
   - **Scalability**: Leverage **Internet of Things (IoT)** devices to gather data directly from trains, tracks, and stations.
6. **Fuel Optimization and Environmental Impact:**
   - **Scalability**: Extend fuel consumption modeling to include alternative energy sources like electric trains and hybrid systems.
7. **Periodic Model Updates and Adaptability:**
   - **Scalability**: Design the system to periodically retrain the model using **automated machine learning (AutoML)** pipelines with newly available data.
8. In this solution, agent or train is having only one target but in realtime it is having multiple targets so here is the chance of improvement and scalability.

These scalability strategies and enhancements aim to future-proof the system, ensuring it remains relevant and effective as rail networks grow in size, complexity, and operational demands.