School of Engineering and Applied Science (SEAS), Ahmedabad University

Probability and Stochastic Processes (MAT 277)

Special Assignment Report

Group Name : $ks - tra - 20$ **Transportation :** Group-20

Team Members - Roll No.

- (1) Kathan Thakker - AU2140040
- (2) Harsh Choksi - AU2140061
- (3) Divykumar Patel - AU2140080
- (4) Krutarth Vora - AU2140162
- (5) Dhruvi Rajput - AU2140224

Algorithms : Kruskal, Boruvka & Dijkstra

I. Problem Statement :

To design a program that utilizes a randomized algorithm to solve the train rescheduling problem, taking into account the distance and time as random variables, while keeping other variables constant.

II. Team Learnings :

The problem of train rescheduling which is retiming, rerouting, or reordering has been there since the time of the industrial revolution and the problem of this solution has been approached through many models and algorithms MIP, DCOP, and MDP. This paper teaches about the modeling of train rescheduling problems on the Petri-net network with DCOP and MDP at its core.

DCOP involves finding a set of solutions that satisfy a set of constraints while optimizing some objective function. In the case of a train rescheduling system, the objective function could be to minimize the total delay of all trains, while the constraints could include limitations on the maximum number of trains that can run on a track at a given time, the maximum speed of each train, and the minimum time between trains on a given track.

Mathematically, DCOP can be represented as follows:
minimize $f(x)$
subject to $g(x) \leq 0$
x_i $\in$ D_i, $i = 1, \ldots, n$

where x is the set of decision variables, $f(x)$ is the objective function to be minimized, $g(x)$ is the constraint functions, and D_i is the domain of the i-th variable. The solution to this optimization problem can be found using various algorithms such as Distributed Gradient Descent, Max-Sum, or DPOP.

MDP involves modeling a system as a set of states, actions, and rewards. The goal is to find a policy that maximizes the expected cumulative reward over a sequence of actions.

In the case of a train rescheduling system, the states could represent the current state of the system (e.g., which trains are running, which trains are delayed, etc.), the actions could represent the possible rescheduling decisions (e.g., changing the schedule of a train, rerouting a train, etc.), and the rewards could represent the desirability of each outcome (e.g., minimizing delays, maximizing on-time arrivals, etc.). Mathematically, an MDP can be represented as follows:

S - set of states

A - set of actions

P(s, a, s') - transition probability function, i.e. probability of moving from state s to state s' after taking action a

R(s, a, s') - reward function, i.e. immediate reward obtained when moving from state s to state s' after taking action a

γ - discount factor, i.e. the weight given to future rewards relative to immediate rewards

The goal is to find a policy π that maps each state to an action, such that the expected cumulative reward is maximized. This can be formulated as follows:

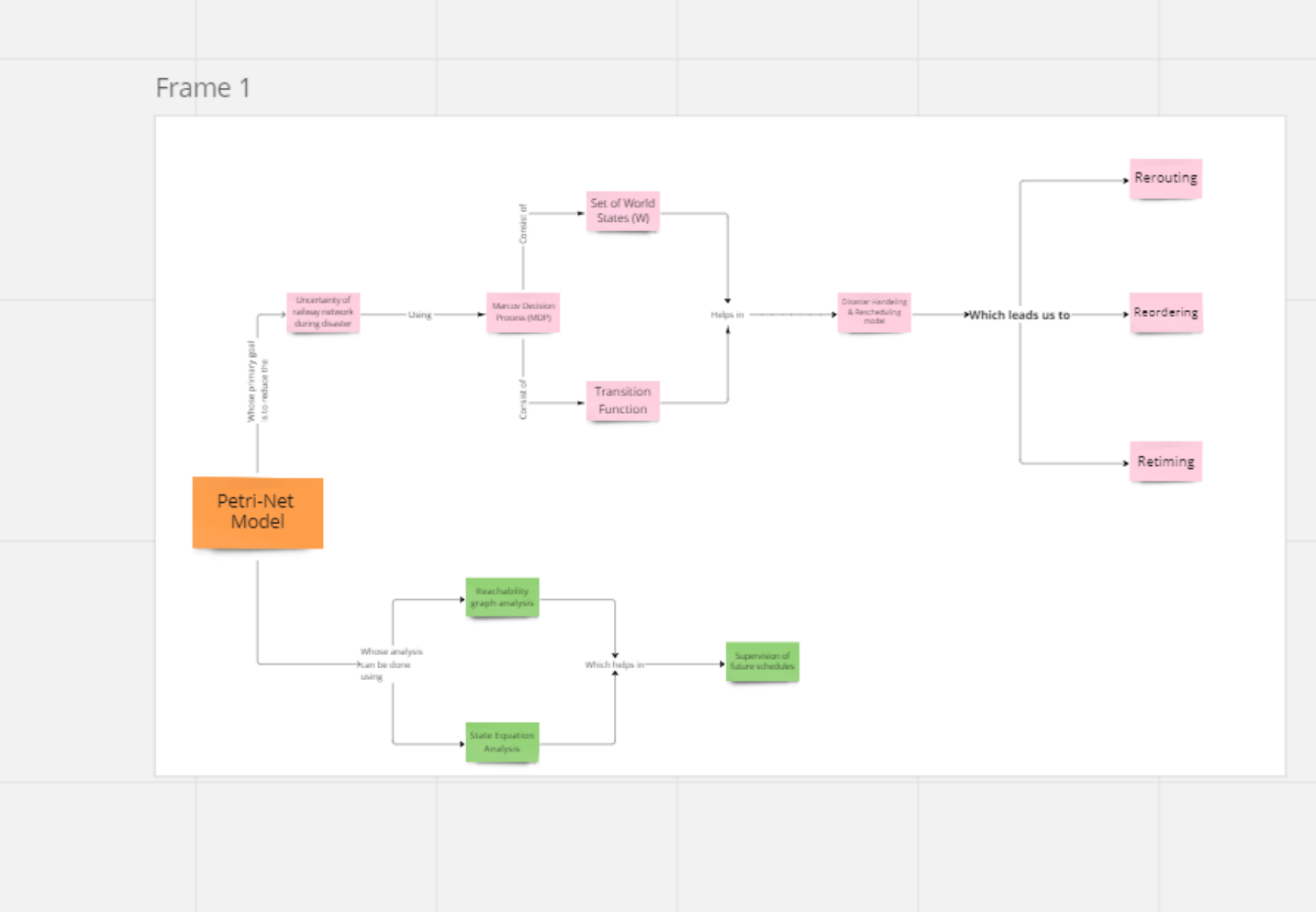
$\pi^* = \text{argmax}\, \Sigma\_s\Sigma\_a\ \Sigma\_s' \text{P}\,(\text{s}, \text{a}, \text{s}')\,[\text{R}\,(\text{s}, \text{a}, \text{s}') + \gamma \text{V}\,(\text{s}')]$
where $V\,(s')$ is the value of state s', defined as:
$\text{V}\,(\text{s}') = \text{max\_a}\ \Sigma\_\text{sP}\,(\text{s}, \text{a}, \text{s}')\,[\text{R}\,(\text{s}, \text{a}, \text{s}') + \gamma \text{V}\,(\text{s}')]$

Solving this optimization problem involves iterating over the states and actions and using dynamic programming algorithms such as Value Iteration or Policy Iteration to find the optimal policy.

Overall, by using DCOP and MDP, we can mathematically model and optimize a train rescheduling system to minimize delays and maximize on-time arrivals.

III. Concept Map

For a better clear view, please visit : Click Here



IV. Background

 the article "Multi-agent-based dynamic railway scheduling and optimisation: a coloured petri-net model" appeared in the journal IEEE Transactions. The paper was written by Kaushik Paul and Poulami Dalapati.

The crucial dates for the publishing of this work are listed below :

Received : 21 February 2021 / **Revised :** 2 February 2022
Accepted : 16 May 2022 / **Published online :** 16 June 2022

The necessity for a more effective method of rearranging a static timetable in the case of a disaster in a big and complex railway network system forms the background of this article. To reduce the effects of such circumstances, it is necessary to dynamically reschedule trains, which causes the goal function to change over time along with the restrictions. With the rising need for rail transportation in recent years, the complexity and size of rail networks have also greatly expanded. This has made more advanced and effective ways for planning and enhancing railway operations necessary.

In order to address the unpredictability and probabilistic nature of recovery time, the authors suggested a unique approach that leverages Petri-Nets and Markov decision processes. The distributed constraint optimization (DCOP)-based approach is then used to solve the rescheduling problem. The method is intended to handle the difficulties posed by dynamic and unpredictable occurrences in railway systems, such as train delays and unexpected changes in passenger demand.

The proposed model and the algorithms applied to improve railway scheduling are described in depth in the study. The method is assessed using a case study of railway networks in powerhouse nations like China and India. The outcomes show how effective the suggested method is in enhancing the railway system's performance and the outcomes show how effective the suggested method is in enhancing the performance of the railway system.

Overall, the paper makes a significant contribution to the world of transportation, particularly the railways, since its main focus is on the scheduling and optimization of trains, offering a fresh and practical method that can be used with a variety of railway systems.

V. Motivation

With millions of people dependent on public transport networks to get to work, education, and other locations, the field of transportation is vital to today's society. Transportation is an important field of research since the effectiveness and dependability of these systems can have a substantial impact on people's daily lives. Moreover, transportation systems can involve challenging mathematics and optimization issues that call for original solutions.

For a number of reasons, our group has found the study "Multi-agent-based dynamic railway scheduling and optimisation: a coloured petri-net model" to be extremely pertinent and fascinating. Using mathematical methods like Petri-Nets, MDPs, and DCOP-based tactics, the research suggests a solution to the railway rescheduling problem. It is therefore a useful tool for individuals who are interested in mathematical modeling and optimization.

The approach put out in this study has both mathematical and social advantages. From a social standpoint, the strategy can enhance the effectiveness and dependability of railway systems, reducing delays and enhancing passengers' overall travel experiences. This can benefit society by lowering the inconvenience and annoyance brought on by delays.

From a mathematical standpoint, the usage of Petri-Nets, MDPs, and DCOP-based techniques in the work offers a novel method for resolving the railway rescheduling issue that can be used to solve other optimization issues outside of railway systems. This has important ramifications for many different sectors of the economy and fields, hence the paper's suggested method is a significant advancement in the optimization field.

In conclusion, our group believes that this study is instructive and beneficial for anyone with an interest in mathematical modeling, optimization, and the effectiveness and dependability of railway systems. The suggested method can be advantageous from both a social and a mathematical standpoint, which makes it an important contribution to the optimisation field.

VI. Algorithm Description

We solved the proposed problem using a randomised algorithm.The given code implements Kruskal's Algorithm with randomized edge selection and Dijstras'a Algorithm to get the shortest path between two nodes

We randomized a greedy algorithm known as Kruskal's algorithm which generates a minimum spanning tree. In vanilla Kruskal's algorithm, the next lightest edge that does not form a cycle is added to the collection of edges once it has started off empty. Once each vertex is connected to a single component, the method is finished. To randomize the original algorithm, the sequence in which the edges are considered is randomly shuffled.

Here, the vertices represent the stations, the edges represent the tracks and the weight of the edges are taken to be the time taken to cover the cover the distance.

→ Now, the steps involved in vanilla Kruskal's algorithm are :

1. Sort each edge according to its weight in non-decreasing order.
2. Choose the smallest edge.
3. Test to see if a cycle can be built using the currently formed spanning tree. Include the edge if the cycle cannot be formed. If not, discard it.
4. Until the spanning tree has $(V - 1)$ edges, repeat step #2.

In the randomized Kruskal's algorithm,

1. To obtain a random order in which to consider the edges of the graph, permute them at random.

2. Create a disjoint-set data structure with each vertex in its own tree as the beginning state to represent the existing forest of trees.

3. Process the edges in the step one random order that was obtained. Check to see if the endpoints of each edge belong to different trees. If so, combine the two trees and add the forest's boundary. Throw the edge away if they are not.

4. Stop when there is just one tree left in the forest, which is the graph's MST.

Here,the graph is treated as a forest and every node it has as an individual tree

Now, in order to find the shortest path, we used Dijiktra's algorithm. The algorithm operates by keeping track of both visited and unvisited nodes. Except for the starting node itself, which is initially set to 0 , all other nodes' distances from the starting node are initially set to infinite.

After repeatedly choosing the unvisited node with the least distance, the algorithm updates the distances of its neighbors and designates that node as visited. It terminates when all the nodes are visited once.

→ **Dijiktra's Algorithm :**

1. **Set all distances to infinity initially :** The initial node's distance is set to 0 and all other distances are set to infinity at the beginning.

2. **Add a visitation marker to the starting node :** The distance to the starting node is set as the current minimal distance and is marked as visited.

3. **Update the distance of neighboring nodes :** If a shorter path is discovered, the distance is updated for all of the current node's neighboring nodes.

4. **Select the unvisited node with the smallest distance :** The algorithm selects the unvisited node with the smallest distance as the new current node and marks it as visited. 5. Repeat steps 3-4 : Repeat the steps until the destination node is visited or all reachable nodes are visited. Until it reaches the destination node or all reachable nodes have been visited, the algorithm updates the distances and visits the next smallest unvisited node.

5. **Terminate the algorithm:** When the process is finished, it returns the shortest route between the starting and ending nodes.

Deterministic Algorithm

The code uses Kruskal's Algorithm and Dijkistra Algorithm

$\Longrightarrow$ **Kruskals Algorithm :**

1. Create empty list $\rightarrow$ Hold edges

2. priority Queue : sort according to weights

3. Create parent Dictionary $\rightarrow$ store he parent of each node in the disjoint set

4. add its edges $\rightarrow$ exclude edges connected to removed node $\rightarrow$ add priority queue

5. Until priority Queue not empty
   $\rightarrow$ pop edge with smallest edge

6. Find the roots of the two nodes connected by the edge using the "find root" function:
   $\rightarrow$If the roots are not the same, add the edge to the MST and set the parent of one of the roots to the other

7. Sort MST

8. *Return MST*

$\Longrightarrow$ **Dijkstra's Algorithm :**

1. Set all distances to infinity $\rightarrow$ store prevoius nodes empty dictionary

2. Set starting distance to 0 $\rightarrow$ add to priority Queue

3. Until prioity queue $\rightarrow$ not empty popnode with smallest distance : iterate through neighbours

4. if neighbour removed skip it

5. Calculate the distance to the neighbor by adding the distance from the current node to the weight of the edge between them

6. If the calculated distance is less than the current distance to the neighbor, update the distance and the previous node

7. Add neighbour$\rightarrow$ priority queue

8. destination reached $\rightarrow$ reconstruct path

9. If the distance $\rightarrow$ infinity, there is no path, return None. Otherwise, return the shortest path

Randomised Algorithm

The code uses the randomized edge selection process for finding the minimum spanning tree.

$\Longrightarrow$ Working of Randomized Kruskal's

Randomly Shuffle the edges of input graph

Store the edges of MST

Create a parent and rank dictionary

1. *parent dictionary stores parent of each node*
2. *rank dictionary stores the rank , which initally set to zero*

Now from the shuffled list of edges

1. *if edge connects to the avoided node than skip the edge*
2. *find the parent nodes* ***using find ()***
3. *if the parent nodes are different add to MST*
4. *merge the two connected components by making the parent node with the higher rank the parent of the other node.*
5. *If both parent nodes have the same rank, make one of them the parent and increase its rank by 1.*

Returns an MST

$\Longrightarrow$ **Dijkstra's algorithm with avoid node :**

1. avoid node $\rightarrow$ remove it and its edges also
2. Initialize a priority queue dictionary distance
3. set start distance to 0 and other to infinity
4. add nodes
5. Priority Queue not empty extract nodes smallest distances
6. calculate distance from start upto neighbour node
7. if the tentative distance is smaller than the current distance of the neighbor, update the distance and add the neighbor to the priority queue.
8. Once reached $\rightarrow$ distance
9. if node avoid provided, add it back to graph with its original edges
10. Return distance

$\Longrightarrow$ **Alternate Path algorithm with avoid node :**

1. If avoidnode is not None: Remove avoidnode and its edges

2. Find MST using Kruskal's algorithm without considering avoidnode

   For each edge (u, v, weight) in the minimum spanning tree:

   (a) Remove the edge from the graph by removing u from the list of v's neighbors and vice versa

   (b) If there is an alternate path from start to end after removing the edge, append it to alternate paths

   (c) Add the edge back to the graph by adding u to the list of v's neighbors and vice versa

3. If avoidnode is not None: Add avoidnode back to the graph and connect it to all its original neighbors

4. Return alternate paths.

1. **Time Complexity Analysis**

Deterministic Algorithm

The time complexity can be done as follows :

Kruskal MST() :

Time Complexity : O(E log E)

Here V is the vertices and E edges of the graph

Using the priority queue

Dijkstra avoid node():

Time Complexity : O((V + E) log V)

Here V is the vertices and E edges of the graph

Using heap to track distance

Find alternate route() :

Time Complexity : O(E log E + V + E)

Here V is the vertices and E edges of the graph

It involves finding the MST using Kruskal's algorithm, then traversing the graph using a depth-first search to find the alternate route.

Putting it altogether :

$=O(E \log E + V + E) + O((V + E) \log V) + O(E \log E)$

$= 2\ \mathrm{O(E \log E)} + \mathrm{O(V + E)} + \mathrm{O((V + E) \log V)}$

$= O(E \log E + E \log V + E)$

Therefore, the time complexity of the given code is as follows :

$O(E \log E + E \log V + E)$.

2. **Time Complexity Analysis**

Randomized Algorithm

The time complexity can be done as follows :

random edges():

Time Complexity :O(V*E)

Here V is the vertices and E edges of the graph

Complexity as it iterates through all the edges and shuffles randomly

find():

Time Complexity :O(log(N)

Here N are the number of nodes

kruskals randomized algorithm ():

Time Complexity is :O(ELog(E))

Here E are the number of edges

it is because it generates the random set of edges and sort them according to the weight,
then iterates through each edge and performs a union-find operation

dijkstras algorithm ():

*Time Complexity is :O(E*log V)*

E is the number of edges in the graph and V is the number of vertices.

This is because it uses the priority queue to determine the minimum distance

Alternate route algorithm ():

Time Complexity is : O(Elog E + Elog V),

The function calls the kruskals and Dijistra's Algo and then iterates through the minimun spanning tree

Putting the above time complexity together :

By simplifying : O(VE) + O(Elog E + Elog V + Elog V) + O(log N)

O(VE) + O(E*log VE) + O(log N)

The highest term in the above order is O(V*E)

Now checking the which Algo is faster (Approximation)

Let's substitute V=1000 and E=499500 in both expressions and simplify them to compare:

O(E log E + E log V + E) = O(499500 log 499500 + 499500 log 1000 + 499500) = O(4,997,487.92)

O(V*E) = O(1000 * 499500) = O(499500000)

As we can see, O(E log E + E log V + E) is much smaller than O(VE) for the given values of V and E. Therefore, O(E log E + E log V + E) is a more efficient time complexity than O(VE) for this case.

VII. Application

1. **Image Segmentation :**

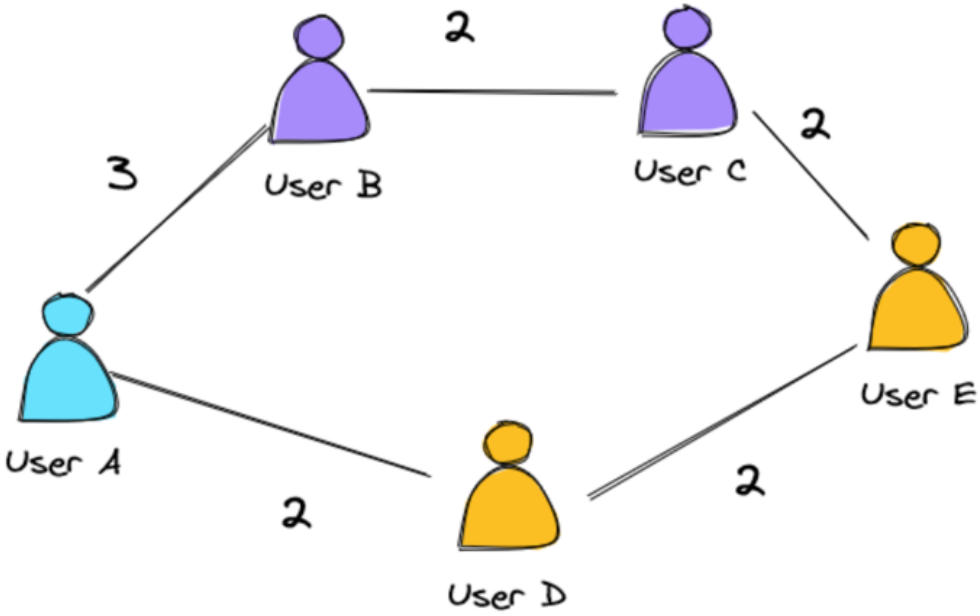
   Data mining makes extensive use of segmentation. An image's pixel is viewed as a node, and its edge is viewed as similarity and dissimilarity between two adjacent points and their weights are determined based on the similarity measure. Then, we apply Kruskal's algorithm to find the minimum spanning tree. The segmentation of the image into distinct sections, each of which contains pixels that are comparable to one another, is represented by the tree.

2. **Supply Chain Management :**

   In supply chain management, the Kruskal algorithm can be used to identify the least expensive route for moving supplies and goods between various points. The nodes represent the various locations, while the edges signify the expense (which includes distance, mode of transportation, and any additional fees)of moving materials and things between those places.

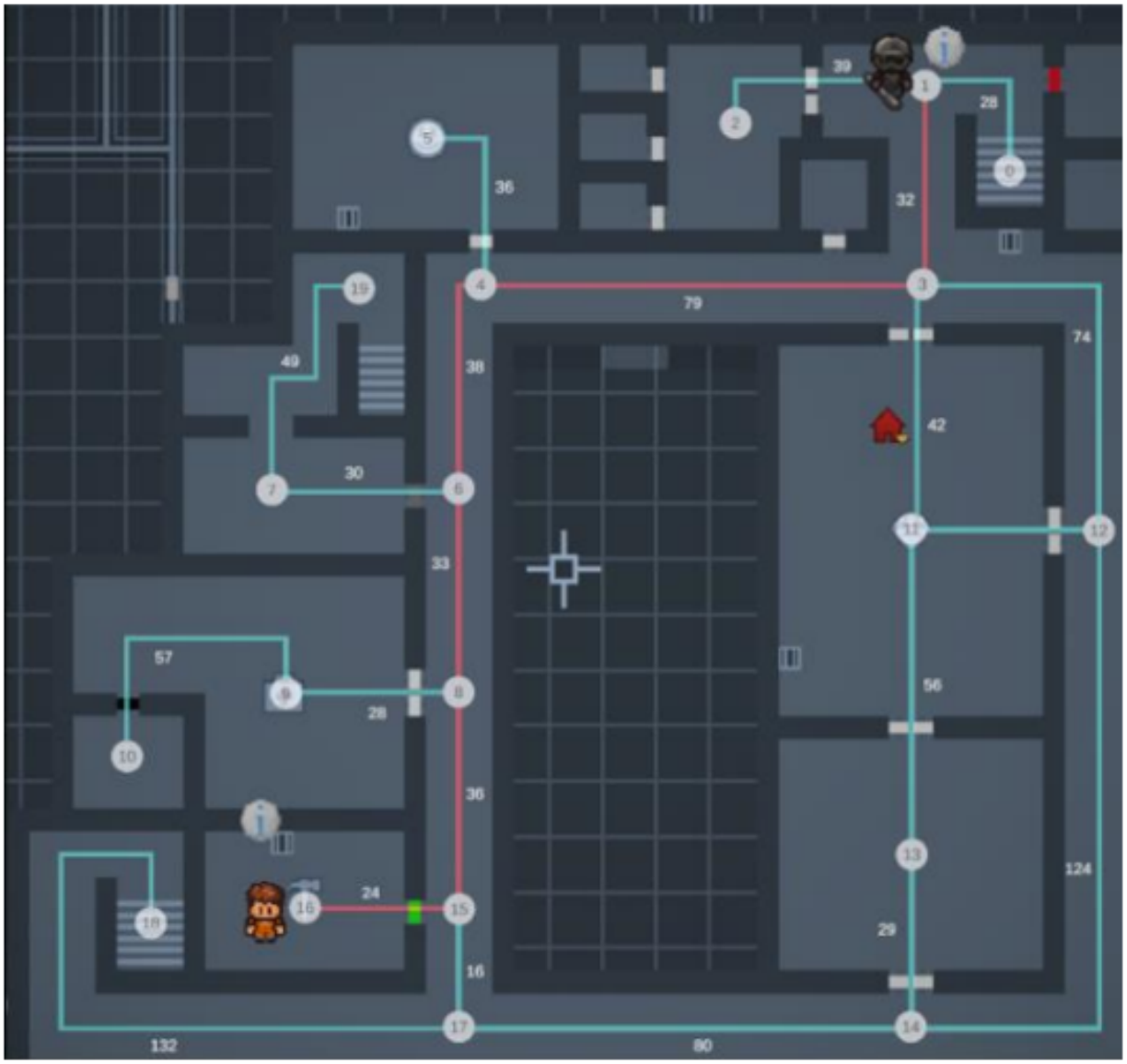
3. **Social Networks :**

   To determine the shortest route between two individuals in social networks, Dijkstra's method might be utilized. Here, the nodes represent the users, the edges represent the relations between them and the weights signify the closeness between them. The algorithm will provide us the least path.



4. **Game AI :**

   While Kruskal's algorithm can be used to create randomised mazes for games, Dijkstra's algorithm can be used to determine the best route for a gaming character to take in order to achieve a goal. For eg,



Here, the nodes will be the rooms and the edges are the paths.(The red path is the shortest).

VIII. Mathematical Analysis

Using the MDP (Markov Decision Process) and minimal spanning tree, train rescheduling can be mathematically modeled using a number of mathematical concepts and methods, including:

1. **Graph theory:** To determine the shortest route between nodes, the minimal spanning tree approach makes use of graph theory.

2. **Linear algebra:** In most cases, matrix multiplication, eigenvectors, and eigenvalues are used to solve the matrices that describe the MDP.

3. **Theory of probabilities:** The MDP is a stochastic process that takes into account the probabilities of various states and actions. These probabilities are computed using probability theory, and the best course of action is determined.

4. **Game theory:** In some circumstances, it is possible to depict the train rescheduling issue as a contest between the railway operator and the passengers. The best solution is determined by using game theory to evaluate the strategic interactions between the various parties. The Multi-agent model uses numerous agents which strongly depend on the behaviour of agents, some external events and also on the present state.

As the railway system is very prone to disasters, the uncertainty of the situation makes the environment more vulnerable. In this paper, this scenario is modeled as Markov decision process (MDP) with its states (W) and transitions (υ).

$\Longrightarrow$ **Set of world state** (W)

W represents the set of agent's state(s) in railway network under disturbance. Train agents can sense three kind of states; if S_i is assumed to be a station, where disaster happens, then T_j is either on track connecting S_i or at a platform of S_i or at a platform of station $S_{i'}$, connected to S_i. i.e. $W = \{(L_{jil} = 1), (P_{jik} = 1), (P_{j'i'k} = 1)\}$

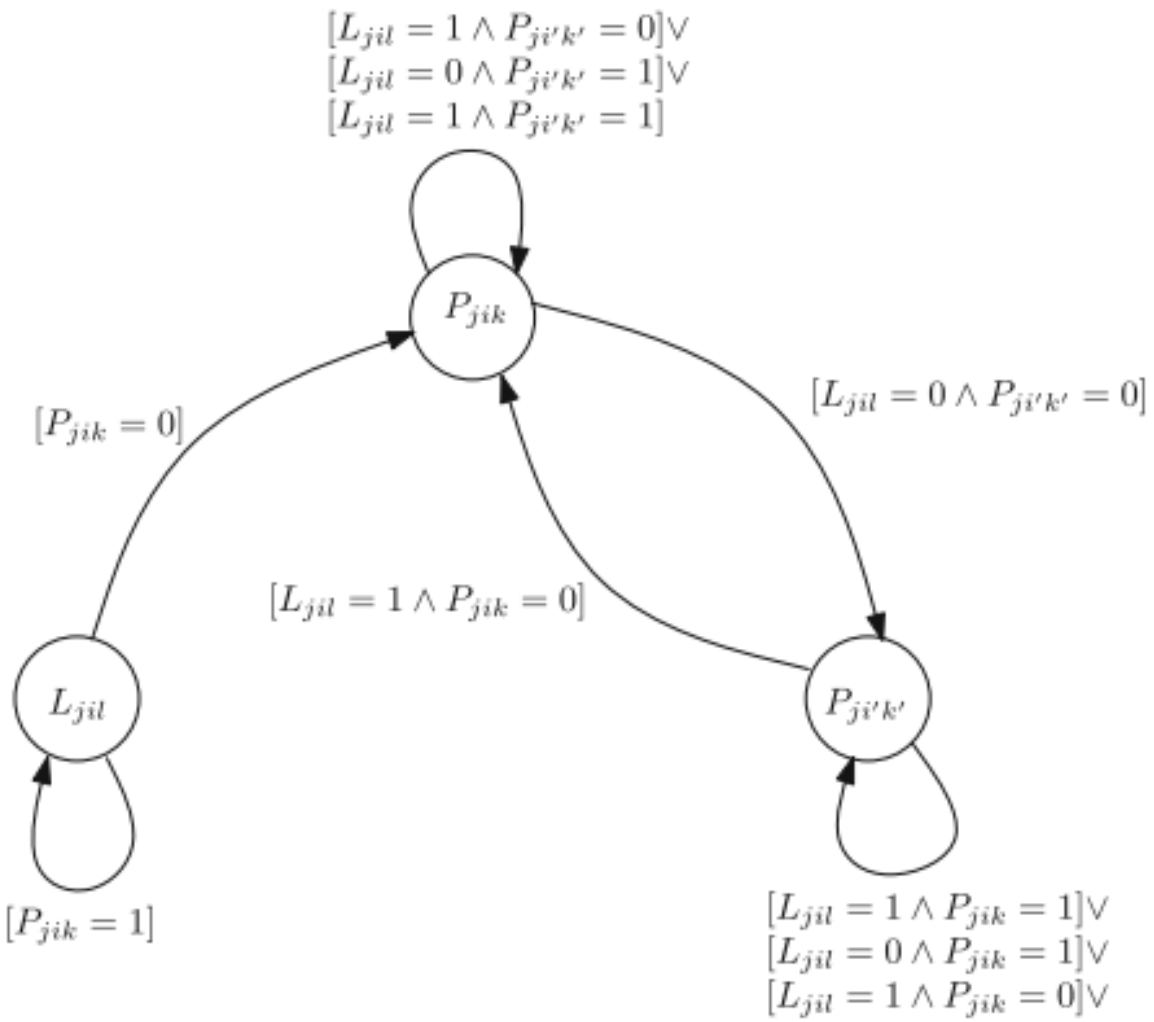
$\Longrightarrow$ **Transition function** (Ψ)

The state transition function is denoted as $\Psi(\omega, C, \omega')$. In our proposed approach, each action C maps to constraint(s) of DCOP to satisfy to reach from state ω to the next state ω', where $\omega, \omega' \in W$.

Every node represents a state of the railway environment and each arc represents an action which is indeed a constraint. The transition from one state to another state happens if and only if the corresponding agent satisfies the specific constraint(s).

- In the figure below,

Ljil = 1 $\rightarrow$ Train j $\rightarrow$ present at station i $\rightarrow$ at Lth trek
Pjik = 1 $\rightarrow$ no platforms available
Pjik = 0 $\rightarrow$ platforms free
Pji$'k'$ = 1 $\rightarrow$ platforms at next station not free


$[L_{jil} = 1 \wedge P_{ji'k'} = 0] \vee$
$[L_{jil} = 0 \wedge P_{ji'k'} = 1] \vee$
$[L_{jil} = 1 \wedge P_{ji'k'} = 1]$
P_{jik}
$[P_{jik} = 0]$
$[L_{jil} = 0 \wedge P_{ji'k'} = 0]$
$[L_{jil} = 1 \wedge P_{jik} = 0]$
L_{jil}
$P_{ji'k'}$
$[P_{jik} = 1]$
$[L_{jil} = 1 \wedge P_{jik} = 1] \vee$
$[L_{jil} = 0 \wedge P_{jik} = 1] \vee$
$[L_{jil} = 1 \wedge P_{jik} = 0] \vee$

Disaster handling and rescheduling approach

According to real-time scenario, in case of platform blockage or track blockage, disaster handling and rescheduling model of railway system refers three situations:

(1) Delay or stop at the station or on track (Retiming).

(2) Change in departure sequence of trains at the station depending on priority of trains (Reordering).

(3) Reschedule to alternative path (Rerouting).

$\Longrightarrow$ Case 1: Partial node deletion from the graph G

*Astation*S$_i$ faces problem due to disaster and the train T_j is on track l, approaching to the station S_i, that is

$$L_{jil} = 1$$

- **Case 1.1**

If S_i has a free platform at the time, when T_j reaches to S_i, then the system can allow T_j to reach S_i, if and only if the priority of the incoming train T_j has the highest priority amongst all trains T and the resource (Re) required for any other high priority train $T_{j''}$ does not hamper the resource requirement of T_j :

$$P_{jik}|_{x_{ji}^{AT}} = 1, \text{ if and only if } \left[\text{Prio}\left(T_j\right) > \text{Prio}\left(T_{j'}\right)_{j\neq j', j\in[1,m]}\right]$$
$$\wedge \left[\text{Re}\left(T_{j''}\right)|^{\tau^B}_{\text{Prio}\left(T_{j''}\right)>\text{Prio}(T_j)} \neq \text{Re}\left(T_j\right)|^{\tau^B}\right]$$

. Route of Tj after departure from Si at time $xDTji$ is [Rou$(Tj)|t \leq xDTji+$ Rou$(Tj)|xATji > x$ DT ji], that is

$$\text{if Rou}\,(T_j)|_{x_{ji}^{AT} > x_{ji}DT} = \bigwedge_{i'>i}^{n-1} (P_{ji'k} L_{ji'l}) \cup P_{jnk} = 0$$

- **Case 1.1.2**

If case 1.1.1 is invalid, then stop T_j at S_i until recovery is done or any other alternative path becomes free. Therefore, T_j occupies one of the platforms at S_i, that is

$\longrightarrow$ Analysis of PN2:

Tables 5 and 6 presents the description of the places $P = \{P1, P2, P3, P4, P5, P6, P7\}$ and transitions $Tr = \{\text{Tr}\,1, \text{Tr}\,2, \text{Tr}\,3, \text{Tr}\,4, \text{Tr}\,5, \text{Tr}\,6, \text{Tr}\,7, \text{Tr}\,8\}$ and the initial marking is $M_0 = [1, 1, 0, 0, 0, 0, 0]$.

$\longrightarrow$ Reachability graph analysis:

Reachability graph analysis is the simplest method to analyse the behaviour of a Petri-Net. It decides whether the system is bounded and live or not. From our resultant tree in Fig. 4 it can be proved that: (a) the reachability set $R\,(M_0)$ is finite, (b) maximum number of tokens that a place can have is 2 , so our $PN2$ is 2-bounded, (c) all transitions can be fired, so there are no dead transitions.

$\Longrightarrow$ **State Matrix :**

The statement behaviour of the petri-net model can be measured using the Algebraic Analysis called the incidence matrix.

Incidence matrix represents the connections between places and transitions, the number of total transitions can be represented through the incidence matrix.

No $\rightarrow$ Initial Marking
N $\rightarrow$ Reachable Marking

It can be done as : $\boldsymbol{M}_0 + [\boldsymbol{A}] \times X_\sigma = \boldsymbol{M}$

$\Longrightarrow$ **Incidence Matrix :**

The order of the places in the matrix is as follows :

$P = \{P1, P2, P3, P4, P5, P6, P7\}$ denoted by rows and

Tr = {Tr 1, Tr 2, Tr 3, Tr 4, Tr 5, Tr 6, Tr 7, Tr 8} denoted by columns

X_σ is an m-dimensional vector with its j th entry denoting the number of times transition t_j occurs in σ :

$$\boldsymbol{A} = \begin{bmatrix} 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \end{bmatrix}$$

Thus, if we view a marking $\boldsymbol{M}_0$ as a k-dimensional column vector in which the i th component is $M_0(p_i)$, each column of $[\boldsymbol{A}]$ is then a k-dimensional vector such that

$\boldsymbol{M}_0 \stackrel{\sigma}{\rightarrow} \boldsymbol{M}$

In our system, marking $\boldsymbol{M} = [1, 1, 0, 0, 0, 0, 0]$ is reachable from initial marking $\boldsymbol{M}_0 = [1, 1, 0, 0, 0, 0, 0]$ through the firing sequence $\sigma_1 = \text{Tr } 2, \text{Tr } 3, \text{Tr } 4, \text{Tr } 6, \text{Tr } 7$.

$M_0 \stackrel{Tr2}{\longrightarrow} M_1 \stackrel{Tr3}{\longrightarrow} M_2 \stackrel{Tr4}{\longrightarrow} M_3 \stackrel{Tr6}{\longrightarrow} M_4 \stackrel{Tr7}{\longrightarrow} M_5(= \text{'old } ')$

$$\begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$ *Similarly we can reach th*

$M_0 \stackrel{Tr2}{\longrightarrow} M_1 (= \text{ 'old } ')$.
$M_0 \stackrel{Tr2}{\longrightarrow} M_1 \stackrel{Tr3}{\longrightarrow} M_2 \stackrel{Tr5}{\longrightarrow} M_3 \stackrel{Tr8}{\longrightarrow} M_2 \stackrel{Tr4}{\longrightarrow} M_4 \stackrel{Tr6}{\longrightarrow}$
$M_5 \stackrel{Tr7}{\longrightarrow} M_6(= \text{ oold }) :$

Currently multiple trains are on various platforms which follow a sequence accordingto

τ^B. Here, τ^B is related to the disaster recovery time t_R of that station. The track is free but more than one

$L_{jil} = 0$

$\sum_{j=1} P_{jik} \leq (p-1)$

As the station S_i faces disaster, a particular k th platform can not be used until the recovery time has elapsed. If there is any incoming train T_j within buffer period τ^B, the system allows T_j to reach S_i if a platform is available, i.e. $P_{jik} = 0$. The system also checks for the priority of T_j to reorder the departure schedule of all trains from S_i introducing delay δ_{ji}toT_j, if needed.

$$\text{i.e. } \forall j, \text{ if } \text{Prio}\,(T_{j'}) > \text{Prio}\,(T_j)$$

$$x_{ji}^{DT} = o_{ji}^{DT} + \delta_{ji}$$

Otherwise, if all the resources are available for T_j and it has the highest priority among all the trains currently waiting at S_i, the scheduled departure of T_j is the original departure time as per the original railway timetable, that is

$$x_{ji}^{DT} = o_{ji}^{DT}$$

$$\textit{if and only if } \text{Prio}\,(T_j) > \text{Prio}\,(T_{j'})_{j \neq j', j \in [1,m]}$$

Analysis of PN3

$P = \{P1, P2, P3, P4, P5, P6\}$ and transitions $Tr = \{\text{Tr } 1, \text{Tr } 2, \text{Tr } 3, \text{Tr } 4, \text{Tr } 5\}$ and the initial marking is $\boldsymbol{M}_0 = [2, 3, 0, 0, 0, 0]$.

The order of the places in the incidence matrix $\boldsymbol{A}$ is $P =$

IX. $\{P1, P2, P3, P4, P5, P6\}$**, denoted by rows and the order of the transitions is** $\text{Tr} = \{\text{Tr } 1, \text{Tr } 2, \text{Tr } 3, \text{Tr } 4, \text{Tr } 5\}$**, denoted by columns:**

$$\boldsymbol{A} = \begin{bmatrix} -1 & 1 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Here, marking $\boldsymbol{M} = [1, 2, 0, 1, 1, 0]$ is reachable from initial marking $\boldsymbol{M}_0 = [2, 3, 0, 0, 0, 0]$ through the firing sequence $\sigma_1 = \text{Tr } 1, \text{Tr } 2, \text{Tr } 3, \text{Tr } 4. M_0 \xrightarrow{Tr1} M_1 \xrightarrow{Tr2} M_2 \xrightarrow{Tr3} M_3 \xrightarrow{Tr4} M_4$

$$\begin{bmatrix} 2 \\ 3 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 1 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Train T_j is neither waiting at the station S_i, where disaster happened, i.e. $P_{jik} = 0$ nor on the connecting track, i.e. $L_{jil} = 0$. However, T_j reaches the station S_i within τ^B.

Train T_j is at station $S_{i''}$, where $S_{i''}$ is in neighbourhood of S_i
that is

$$P_{ji''k} = 1, \quad S_{i''} \in S \backslash S_i \text{ and } i \in [1, n]$$

If any platform is available at the next station and the connecting track is also free, the system checks for the priority of the train T_j. T_j maintains its original schedule if and only if it has the highest priority while reaching S_i, i.e., if

$$(P_{jik'} = 0) \wedge (L_{jil} = 0) \wedge \Big(\text{Prio}\,(T_j)|_{t=x_{ji}^{AT}} > \text{Prio}\,(T_{j'})|_{j \neq j', j \in [1,m]} \Big)$$

then

$$x_{ji}^{AT} = o_{ji}^{AT}$$

$$L_{jil} = 1 \text{ and } P_{jik'}|_{t=x_{ji}^{AT}} = 0$$

$$P_{jik'}|_{t=x_{ji}^{AT}} = 1 \text{ and } L_{jil} = 0$$

Here, damaged platform is $k.k' = \{1, 2, \ldots, p\} \backslash \{k\}$

Case 3.2 Train T_j is at $S_{i'}$, i.e. $P_{ji'k} = 1$, where $i, i' \in [1, n]$ and $i \neq i'$

There are multiple tracks between two stations S_i and $S_{i'_.}$. i.e. $1 < l \leq 4$. If the track l breaks down due to disaster, it is assumed that track l is not free, that is

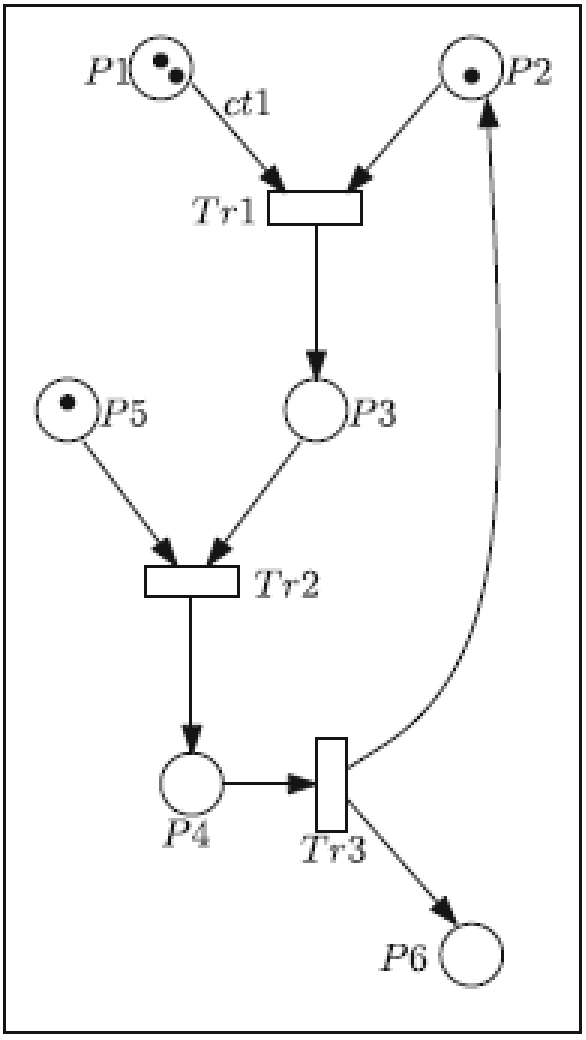
$$L_{ji'l} = 1, 1 \ \leq \ l < 4$$

Then, the trains which are scheduled to use that track face problem. In that case, first $S_{i'}$ checks for other available tracks, one of which can be allotted to T_j, provided T_j has the highest priority satisfying all the constraints and there is no resource conflict within τ^B.

$$\begin{aligned}[L_{ji'l'} = 0] \wedge \Big[& \text{Prio}\,(T_j) |\, t = xjj'^{DT} \\ & > \text{Prio}\,(T_{j'}) |\, j \neq j',, j \in [1, m]] \wedge \Big[\text{Re}\,(T_j)|_{xji^{AT}} \\ & \neq \text{Re}\,(T_{j'}) |\, xji^{AT} \Big]\end{aligned}$$

Figure 6 represents Petri-Net model for the scenario of Case 3

Analysis of PN4: Tables 11 and 12 presents the description of the places $P = \{P1, P2, P3, P4, P5, P6\}$ and transitions $Tr = \{\text{Tr}\,1, \text{Tr}\,2, \text{Tr}\,3\}$ and the initial marking is $\boldsymbol{M}_0 = [2, 1, 0, 0, \ 1, 0]$.



(a)

M_0=[2 1 0 0 1 0]
$Tr1$
M_1=[1 0 1 0 1 0]
$Tr2$
M_2=[1 0 0 1 0 0]
$Tr3$
M_3=[1 1 0 0 0 1]

(b)

Fig. 6 Petri-Net $PN4$. **a** Petri-Net model $PN4$ of Case 3. **b** Reachability tree of $PN4$ for different firing sequences

X. Code :

For a better preview, please refer the below collab file :

https://bit.ly/3zIvWjB

1. **Finding Alternate Route using Kruskal's Algorithm, Dijkstra's Algorithm and Depth-First Search**

The code begins here :

```
import time

# Start the timer
start_time = time.time()
from queue import PriorityQueue
from collections import defaultdict

start = 'A'
end = 'D'
avoid_node = 'C'

# The graph is defined as a dictionary where each key represents
graph = {
  'A': {'B': 2, 'C': 3},
  'B': {'A': 2, 'D': 4},
  'C': {'A': 3, 'D': 1},
  'D': {'B': 4, 'C': 1},
}

def kruskal_mst_with_removed_node(graph, removed_node=avoid_node)
    # Initialize variables
    mst = []
    edges = PriorityQueue()
```

```
    parent = {}

    # Define a function to find the root of a node in the disjoin
    def find_root(node):
        while parent[node] != node:
            parent[node] = parent[parent[node]]
            node = parent[node]
        return node

    # Initialize the parent dictionary and add all edges to the p
    for node in graph:
        parent[node] = node
        for neighbor, weight in graph[node].items():
            if node != removed_node and neighbor != removed_node:
                edges.put((weight, node, neighbor))

    # Loop through all edges and add them to the MST if they don'
    while not edges.empty():
        weight, u, v = edges.get()
        root_u = find_root(u)
        root_v = find_root(v)
        if root_u != root_v:
            mst.append((u, v, weight))
            parent[root_u] = root_v

    # Sort the MST based on the order of the nodes in the edge tu
    mst.sort(key=lambda x: (graph[x[0]][x[1]], x[0], x[1]))

    return mst

import heapq

def dijkstra_shortest_path(graph, start, end, removed_node=avoid_
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    queue = [(0, start)]
    prev_nodes = {node: None for node in graph}
```

```
    while queue:
        current_distance, current_node = heapq.heappop(queue)

        # Skip nodes that have been removed from the graph
        if current_node == removed_node:
            continue

        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            if neighbor == removed_node:
                continue
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                prev_nodes[neighbor] = current_node
                heapq.heappush(queue, (distance, neighbor))

    path = []
    node = end
    while node is not None:
        path.append(node)
        node = prev_nodes[node]

    path.reverse()

    return path if distances[end] != float('inf') else None


# Define a function to find an alternate route for a delayed trai
def find_alternate_route(mst, origin, destination, closed_track,
    # Initialize variables
    edges = PriorityQueue()
    path = []
    visited = set()
```

```
stack = [origin]
distances = {node: float('inf') for node in graph}  # distanc
distances[origin] = 0
time = {node: float('inf') for node in graph}  # time from or
time[origin] = 0
parent = {}

# Define a function to find the root of a node in the disjoin
def find_root(node):
    while parent[node] != node:
        parent[node] = parent[parent[node]]
        node = parent[node]
    return node

# Initialize the parent dictionary and add all edges to the p
for node in graph:
    parent[node] = node
    for neighbor, weight in graph[node].items():
        edges.put((weight, node, neighbor))

# Loop through all edges and add them to the MST if they don'
while not edges.empty():
    weight, u, v = edges.get()
    root_u = find_root(u)
    root_v = find_root(v)
    if root_u != root_v:
        mst.append((u, v, weight))
        parent[root_u] = root_v

# Sort the MST based on the order of the nodes in the edge tu
mst.sort(key=lambda x: (graph[x[0]][x[1]], x[0], x[1]))

# This function performs a depth-first search on a graph to f
def dfs(node):

    visited.add(node)
    path.append(node)
```

```
    if node == destination:
        return True
    for u, v, weight in mst:
        if u == node and v not in visited and v != closed_tra
            if heuristic == 'shortest_distance':
                if distances[node] + weight < distances[v]:
                    distances[v] = distances[node] + weight
                    parent[v] = node
                if dfs(v):
                    return True
            elif heuristic == 'shortest_time':
                if time[node] + weight < time[v]:
                    time[v] = time[node] + weight
                    parent[v] = node
                if dfs(v):
                    return True
    # True if the destination node is found, False otherwise.
    path.pop()
    return False

# Call the DFS function to find the path
dfs(origin)

# Check if the path is valid
if not path:
    print("No valid path found.")
elif path[-1] != destination:
    print("No valid path found.")
else:
    # Given a path and a destination, print out the alternate
    print("Alternate Route:")
    for i in range(len(path)-1):
        u, v = path[i], path[i+1]
        for edge in mst:
            if edge[0] == u and edge[1] == v:
                print(f"{u} - {v} : {edge[2]}")
                break
```

```
# Find the minimum spanning tree using Kruskal's algorithm
mst = kruskal_mst_with_removed_node(graph,avoid_node)
print("Minimum Spanning Tree:")
for edge in mst:
    print(f"{edge[0]} - {edge[1]} : {edge[2]}")

shortest_path = dijkstra_shortest_path(graph, start, end, avoid_n
if shortest_path is not None:
    print(f"Shortest path from {start} to {end} with {avoid_node}
else:
    print(f"No path found from {start} to {end} with {avoid_node}

# Find an alternate route using DFS
alternate_route = find_alternate_route(mst, start, end, closed_tr

# Define a function to find all paths in a graph that do not cont
def find_paths(graph, start, end, avoid_node):
    paths = []
    visited = set()

    # Define a helper function to recursively find paths
    def dfs(node, path):
        if node == end:
            paths.append(path)
        else:
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited and neighbor != avoid_
                    dfs(neighbor, path + [neighbor])
            visited.remove(node)

    # Call the helper function to find all paths from the start n
    dfs(start, [start])
    return paths
paths = find_paths(graph, start, end, avoid_node)
for path in paths:
```

```
    print(path)


# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"The code block took {elapsed_time} seconds to run.")
```

2. **Finding Shortest and Alternate Paths in a Graph with a Randomized Kruskal's Algorithm and Dijkstra's Algorithm**

The code begins here :

```
import time

# Start the timer
start_time = time.time()

import random
import heapq

start = 'A'
end = 'D'
avoidnode = 'E'

graph = {
  'A': {'B': 2, 'C': 3},
  'B': {'A': 2, 'D': 4, 'E': 3},
  'C': {'A': 3, 'D': 1},
  'D': {'B': 4, 'C': 1},
  'E': {'B': 3}
}


# function to generate random edges
def random_edges(graph):
    edges = []
    for u in graph:
        for v in graph[u]:
            edges.append((u, v, graph[u][v]))
    random.shuffle(edges)
    return edges
```

```
# function to find parent of a node
def find(parent, node):
    if parent[node] == node:
        return node
    parent[node] = find(parent, parent[node])
    return parent[node]

# function to run Kruskal's algorithm with randomized edge select
def kruskals_randomized_algorithm_without_node_r(graph, avoidnode
    parent = {}
    rank = {}
    for node in graph:
        parent[node] = node
        rank[node] = 0
    edges = random_edges(graph)
    mst = []
    for edge in edges:
        u, v, weight = edge
        if u == avoidnode or v == avoidnode:
            continue
        parent_u = find(parent, u)
        parent_v = find(parent, v)
        if parent_u != parent_v:
            mst.append(edge)
            if rank[parent_u] > rank[parent_v]:
                parent[parent_v] = parent_u
            elif rank[parent_u] < rank[parent_v]:
                parent[parent_u] = parent_v
            else:
                parent[parent_v] = parent_u
                rank[parent_u] += 1
    return mst



def dijkstras_algorithm_r(graph, start, end,avoidnode):
```

```
    if avoidnode is not None:
        # remove the specified node and its edges from the graph
        if avoidnode in graph:
            del graph[avoidnode]
        for node in graph:
            if avoidnode in graph[node]:
                del graph[node][avoidnode]

    queue = []
    distance = {}
    for node in graph:
        distance[node] = float('inf')
    distance[start] = 0
    heapq.heappush(queue, (0, start))
    while queue:
        curr_dist, curr_node = heapq.heappop(queue)
        if curr_node == end:
            return distance[end]
        for neighbor in graph[curr_node]:
            new_dist = curr_dist + graph[curr_node][neighbor]
            if new_dist < distance[neighbor]:
                distance[neighbor] = new_dist
                heapq.heappush(queue, (new_dist, neighbor))

    # add back the removed node and its edges to the graph
    if avoidnode is not None:
        graph[avoidnode] = {}
        for node in graph:
            if avoidnode in graph[node]:
                graph[node][avoidnode] = graph[avoidnode][node]

    return distance[end]
# function to find alternate paths using Kruskal's algorithm with
def alternate_paths_r(graph, start, end, avoidnode):
    if avoidnode is not None:
        # remove the specified node and its edges from the graph
        if avoidnode in graph:
```

```
            del graph[avoidnode]
        for node in graph:
            if avoidnode in graph[node]:
                del graph[node][avoidnode]

    # find the minimum spanning tree
    mst = kruskals_randomized_algorithm_without_node_r(graph,avoi

    # find alternate paths for each edge in the minimum spanning
    alternate_paths = []
    for u, v, weight in mst:
        graph[u].pop(v)
        graph[v].pop(u)
        if dijkstras_algorithm_r(graph, start, end,avoidnode) !=
            alternate_paths.append((u, v, weight))
        graph[u][v] = weight
        graph[v][u] = weight

    # add back the removed node and its edges to the graph
    if avoidnode is not None:
        graph[avoidnode] = {}
        for node in graph:
            if avoidnode in graph[node]:
                graph[node][avoidnode] = graph[avoidnode][node]

    return alternate_paths

# example graph
mst = kruskals_randomized_algorithm_without_node_r(graph,avoidnod
print("Minimum Spanning Tree:")
for edge in mst:
  print(edge)

shortest_path = dijkstras_algorithm_r(graph, start, end,avoidnode
print("\nShortest Path:")
print(shortest_path)
```

```
alternate_paths = alternate_paths_r(graph, start, end, avoidnode)
print("\nAlternate Paths with Node B Removed:")
for path in alternate_paths:
  print(path)

for i in range(1000000):
    pass

# Stop the timer
end_time = time.time()

# Calculate the elapsed time
elapsed_time = end_time - start_time

# Print the elapsed time
print(f"The code block took {elapsed_time} seconds to run.")
```

3. **Comparison of Runtime for Kruskal's Randomized and Deterministic Algorithms on Large Graphs**

The code begins here :

```
import time
import random
import matplotlib.pyplot as plt
# The time vs nodes graph for a larger graph
# function to generate random graph with n nodes
def generate_graph(n):
# Generate a graph with n nodes, where each node is connected to
# with a random weight between 1 and 10.
    graph = {}
    for i in range(n):
        node = str(i)
        edges = {}
        for j in range(n):
            if i != j:
                neighbor = str(j)
                weight = random.randint(1, 10)
                edges[neighbor] = weight
        graph[node] = edges
    # Returns a dictionary representing the graph, where each key
    return graph

# number of nodes to test
nodes = [1, 10, 100, 1000]

runtimes = []
runtimes_r = []
for n in nodes:
    # Generate random graph
    graph = generate_graph(n)

    # Measure runtime of the code
```

```
        start_time = time.time()
        # This code snippet calculates the runtime of two algorithms,
        mst = kruskals_randomized_algorithm_without_node_r(graph,avoi
        shortest_path = dijkstras_algorithm_r(graph, '0', str(n-1),av
        alt_paths_list = alternate_paths_r(graph, '0', str(n-1),avoid
        alt_paths = len(alt_paths_list)
        end_time = time.time()
        runtime = end_time - start_time
        runtimes_r.append(runtime)

        start_time = time.time()
        mst = kruskal_mst_with_removed_node(graph)
        alternate_route = find_alternate_route(mst, '0', str(n-1),clo
        end_time = time.time()
        runtime = end_time - start_time
        runtimes.append(runtime)

# Ploting the graph
fig, ax = plt.subplots()
ax.plot(nodes, runtimes_r, label='Randomized')
ax.plot(nodes, runtimes, label='Deterministic')
ax.legend()
plt.xlabel('Number of Nodes')
plt.ylabel('Runtime (seconds)')
plt.show()
```

4. **Comparison of Runtime for Kruskal's Randomized and Deterministic Algorithms on Small Graphs**

The code begins here :

```
    import time
import random
import matplotlib.pyplot as plt
# The time vs nodes graph for comparing the randomized and determ
# function to generate random graph with n nodes
def generate_graph(n):
    graph = {}
    for i in range(n):
        node = str(i)
        edges = {}
        for j in range(n):
            if i != j:
                neighbor = str(j)
                weight = random.randint(1, 10)
                edges[neighbor] = weight
        graph[node] = edges
    return graph

# number of nodes to test
nodes = [1,10,20,30,40,50,60,70,80,90,100]

runtimes = []
runtimes_r = []
for n in nodes:
    # generate random graph
    graph = generate_graph(n)

    # measure runtime of the code
    start_time = time.time()
    mst = kruskals_randomized_algorithm_without_node_r(graph,avoi
    shortest_path = dijkstras_algorithm_r(graph, '0', str(n-1),av
```

```
        alt_paths_list = alternate_paths_r(graph, '0', str(n-1),avoid
        alt_paths = len(alt_paths_list)
        end_time = time.time()
        runtime = end_time - start_time
        runtimes_r.append(runtime)

        start_time = time.time()
        mst = kruskal_mst_with_removed_node(graph)
        alternate_route = find_alternate_route(mst, '0', str(n-1),clo
        end_time = time.time()
        runtime = end_time - start_time
        runtimes.append(runtime)

# plot the graph
fig, ax = plt.subplots()
ax.plot(nodes, runtimes_r, label='Randomized')
ax.plot(nodes, runtimes, label='Deterministic')
ax.legend()
plt.xlabel('Number of Nodes')
plt.ylabel('Runtime (seconds)')
plt.show()
```

XI. Results and Inferences

Results - Deterministic Alorithm :

We gave a graph as input in the algorithm, and we removed a node of the graph considering it as blocked station , we were able to find a MST through this graph using the Kruskal's Algorithm.

Now we find the Shortest path with the removed node using Dijikstra Algorithm by removing the blocked station.

We then generated all the possible alternate routes of the Graph with the removed node.

Results Randomised Algorithm :

We are using randomized edge selection process to find the minimum spanning tree of a given graph.

The we implemented the Dijkistra Algorithm to find the shortest path of graph with removed node.

Alternate Path :

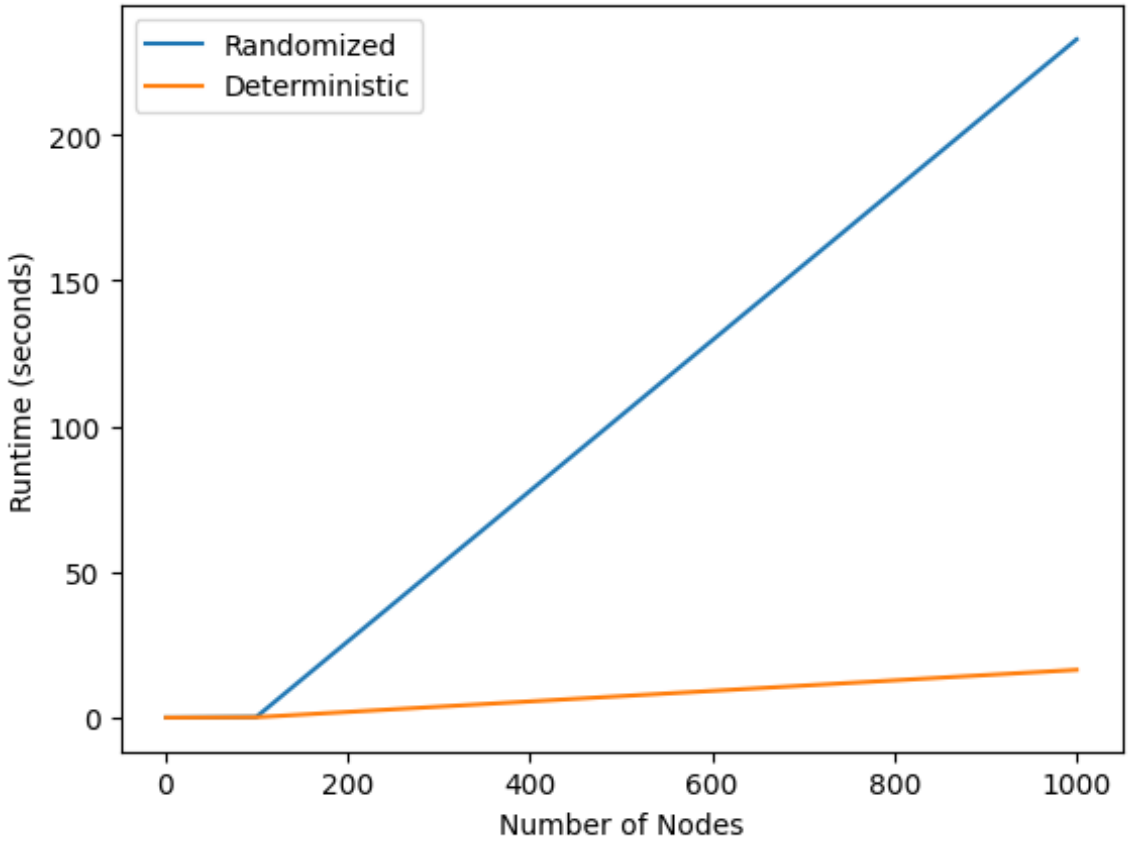
It does so by first removing the avoid node and its edges from the graph,then finding the minimum spanning tree of the remaining graph using Kruskal's randomized algorithm without the avoid node.

It then iterates over each edge in the minimum spanning tree, temporarily removes the edge from the graph, and checks if there is a path between the start and end nodes in the modified graph using Dijkstra's algorithm. If a path exists, the edge is added to the list of alternate paths. The function then adds back the removed node and its edges to the graph and returns the list of alternate paths.

Final Results :

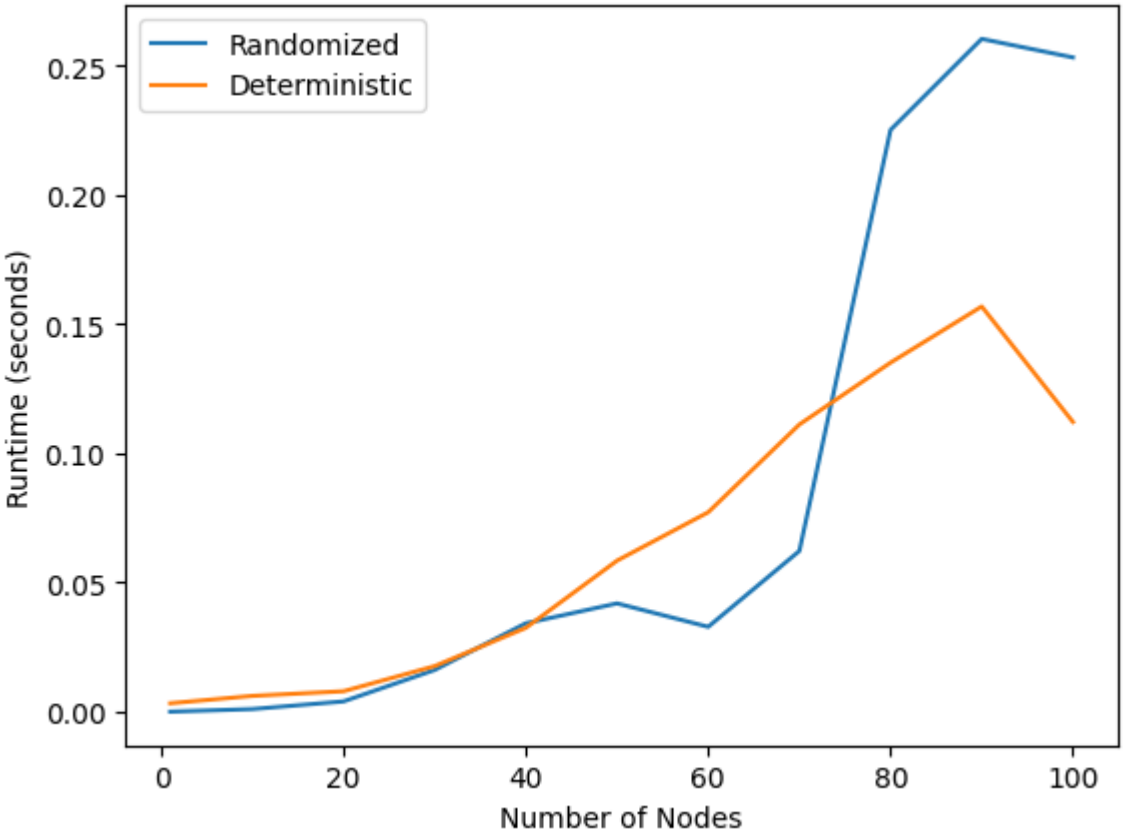
When we tested the Randomised and Deterministic Algorithm for smaller inputs we were able to draw that for smaller inputs the randomised algorithms run much faster than deterministic approach.

When we tested both the codes for larger inputs we were able to get that Randomised Code grows very faster than Deterministic Algorithms.



(a) fig 1

Figure 1: Plot for alternative path



(a) fig 2

Figure 2: Zoomed in view

XII. References

1. **RMST Notes (Campuswire) :** https://bit.ly/3zM4wJY

2. https://files.campuswire.com/477259e7-8684-40ee-b62c-4cbfe1d025c5
   6d4fceb9-0d9c-4639-98d4-163aeebd17a6/Report_linear_time_
   mst.pdf

3. **Kruskal's Algorithm :** https://www.youtube.com/watch?v=b2-vAkasvD0&
   list=PLyqSpQzTE6M9DKhN7z2fOpKTJWu-639_P&index=30

4. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/
   2022-2023/Makalah2022/Makalah-Matdis-2022%20(96).pdf

5. https://siddhigate.hashnode.dev/social-network-and-recommendation

6. https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/
   04GreedyAlgorithmsII.pdf