UNIVERSITY OF MACEDONIA
DEPARTMENT OF APPLIED INFORMATICS
MSC IN ARTIFICIAL INTELLIGENCE AND DATA ANALYTICS

Train-scheduling for railway networks using basic kinematics and A*

A dissertation
by

Dimitris Manolakis

Thessaloniki, February 2024

Train-scheduling for railway networks using basic kinematics and A*

Dimitris Manolakis

Bachelor's Degree, University of Macedonia, 2020

Dissertation

Submitted in partial fulfilment of the requirements for
THE MSC IN ARTIFICIAL INTELLIGENCE AND DATA ANALYTICS

Supervisor Professor
Refanidis Ioannis

Approved by the three-member Examination Committee on dd/mm/yyyy

Full name 1	Full name 2	Full name 3
....	....	....

Dimitris Manolakis

....

Abstract

This thesis tackles the problem of finding time optimal routes for trains over a railway network. The problem is defined as follows: A train has a known length. The position of the train is defined over parts of one or more consecutive track segments. There are a maximum speed, a maximum acceleration and a maximum deceleration capability for the train. Each track segment has a maximum allowed speed for any train being over it. A problem instance is defined by an initial and a goal state, which are two positions accompanied with desired speeds (being usually, but not necessarily, zero). In this study we are interested in minimizing the total duration of reaching the goal state from the initial one; other metrics such as fuel consumption could be considered.

We solve this problem using basic kinematics and A*. We present two algorithms: The first one computes analytically in continuous space the optimal speed profile of the train for a problem defined over a given path. The second algorithm extends the first one over arbitrary graphs. A* empowered with a simple admissible heuristic is employed to find the optimal combination of speed profile and path.

Keywords: Scheduling, Heuristic search, Kinematics

Περίληψη

Η παρούσα διατριβή ασχολείται με το πρόβλημα της εύρεσης χρονικά βέλτιστων διαδρομών για τρένα σε ένα σιδηροδρομικό δίκτυο. Το πρόβλημα ορίζεται ως εξής: Ένα τρένο χαρακτηρίζεται από το μήκος του. Η θέση του τρένου ορίζεται σε τμήματα ενός ή περισσότερων διαδοχικών τμημάτων της διαδρομής. Υπάρχει μια μέγιστη ταχύτητα, μια μέγιστη επιτάχυνση και μια μέγιστη επιβράδυνση για το τρένο. Κάθε τμήμα της διαδρομής έχει μια μέγιστη επιτρεπόμενη ταχύτητα για τις αμαξοστοιχίες που βρίσκονται πάνω σε αυτό. Ένα πρόβλημα ορίζεται από μια αρχική και μια κατάσταση-στόχο, οι οποίες είναι δύο θέσεις που συνοδεύονται από επιθυμητές ταχύτητες (που συνήθως, αλλά όχι απαραίτητα, είναι μηδενικές). Στην παρούσα εργασία μας ενδιαφέρει η ελαχιστοποίηση της συνολικής διάρκειας για την επίτευξη της κατάστασης στόχου από την αρχική. Θα μπορούσαν να εξεταστούν και άλλες μετρήσεις, όπως η κατανάλωση καυσίμων.

Επιλύουμε αυτό το πρόβλημα χρησιμοποιώντας βασική κινηματική και τον αλγόριθμο A*. Παρουσιάζουμε δύο αλγορίθμους: Ο πρώτος υπολογίζει αναλυτικά σε συνεχή χώρο το βέλτιστο προφίλ ταχύτητας του τρένου για ένα πρόβλημα που ορίζεται σε μια δεδομένη διαδρομή. Ο δεύτερος αλγόριθμος επεκτείνει τον πρώτο πάνω σε γραφήματα. Για την εύρεση του βέλτιστου συνδυασμού προφίλ ταχύτητας και διαδρομής, χρησιμοποιείται ο αλγόριθμος A* ενισχυμένος με μια απλή αποδεκτή ευρετική μέθοδος.

Keywords: Χρονοπρογραμματισμός, Ευρετική αναζήτηση, Κινηματική

Acknowledgements

I would like to express my graditute to my advisor, Professor Ioannis Refanidis for providing me with the opportunity to work on this thesis, as well as, for his support and patience throughout my research. Finally, my appreciation extends to my family and friends, whose consistent support, love, and patience have been a source of strength throughout this educational journey.

Contents

List of Figures

List of Tables

List of Algorithms

List of Functions

1 Introduction

1.1 Problem – Importance of the topic

Railways are key components in the transportation systems of many countries around the world, with many European development economists considering a modern rail infrastructure as a significant indicator of a country's economic advancement [1]. Moreover, the European Commission aims for rail networks to absorb the majority of medium-distance passenger transport by 2050 [2]. Increasing the efficiency of transport in the railway infrastructure can greatly improve speed and reliability, reduce costs and energy consumption.

In recent years, research on train scheduling has grown significantly. Companies such as the Swiss Federal Railways invest in research through the Flatland challenge, on tasks which include train schedule optimization (railway timetable generation) [3], as well as applications of multi-agent reinforcement learning on the re-scheduling problem (RSP) [4].

1.2 Aim – Objectives

In this study, we present algorithms to generate time-optimal routes for single trains over arbitrary railway networks, using basic kinematics. The main contribution of this paper is a realistic approach to compute the amount of time a train needs to traverse a certain path, making use of a variety of train and path characteristics, particularly train max speed, acceleration and deceleration; train length; and maximum allowed speed per railway segment, while aiming at minimizing journey duration. Our approach employs continuous domains (a train can be anywhere in the railway network and have any speed and acceleration or deceleration, while satisfying the physical constraints), whereas it uses basic kinematic equations to compute the state of a train over time. The aim is to approximate real-world conditions on railway networks, which, in turn, will improve the optimization of the train-scheduling problem and contribute to the development of an application that can be effectively used on real-world railway networks. The project's structure and originality allows a lot of possible extensions, including more features of a train (e.g., load; engine power; variable maximum acceleration/deceleration depending on speed) or the environment (e.g., slopes of the ground), as well as alternative optimization criteria (e.g., fuel consumption).

1.3 Contribution

This study contributes in two directions: First, we present an algorithm to compute the precise duration needed for the train to arrive at any position over a given path, while respecting the physical constraints of the train and the maximum allowed speed for any segment of the network. Second, we empower the first algorithm with A* search and a common straight line distance admissible heuristic function in order to find the optimal route between a source and a target state over any given railway network.

1.4 Structure of the study

The rest of the dissertation is organized as follows. Section 2 presents a review of the background literature, whereas Section 3 provides the problem formulation. Sections 4 and 5 present the algorithms which are used, in order to compute the optimal speed profile over a given path and the shortest (in terms of time) combination of path and speed profile over a railway network, respectively. Finally, Section 6 concludes the dissertation and poses future challenges.

2 Literature review

In this section, the literature review is presented, organized into three distinct parts. The first part delves into Optimization techniques, while the second section focuses on Heuristic search approaches. Finally, Reinforcement learning techniques in combination with Deep Learning are examined in the third part.

2.1 Optimization techniques

A great amount of effort has been invested in solving the train-scheduling problem for multiple trains on a single or multiple railway tracks, with an emphasis on conflict resolution in the planning phase. Optimization is used in order to generate timetables, with many works [5] [6] [7] [8] [9] [10] presenting surveys on the subject. Specifically, Assad [5] published one of the first surveys on rail transportation including optimization, queuing and simulation approaches. In [6], the authors focused on various optimization models for the most commonly studied railway routing and scheduling problems. Finally, Tornquist [10] examined a total of 48 approaches published between 1973 and 2005, classifying them with respect to problem type, solution mechanism, and type of evaluation.

The train scheduling problem on a single track line has been modeled as a job shop scheduling problem and solved with a branch and bound algorithm [11], while the same algorithm has also been used on railway networks [12]. The patent US20110046827A1 [13] showcases a method for controlling speed in an automatic train operation, making use of kinematic equations in order to estimate and control speed. However, the train's length is not included in the problem formulation reducing the complexity significantly since for example, a train cannot be in multiple segments with overlapping speed limits. In [14], the authors worked on the simultaneous train routing and timetabling problem on the rail network, applying a train-based Lagrangian relaxation decomposition. Finally, linear programming formulations on timetable rescheduling were used in [15], testing on instances from Netherlands Railways.

2.2 Heuristic search approaches

Above all, the train scheduling problem is considered an NP-complete problem [14], rendering mathematical optimization approaches unrealistic in terms of computation times, especially for a large-scale problem. For that reason, many researchers have proposed and evaluated many heuristic search approaches and artificial intelligence techniques. Genetic algorithms (GA) were used in order to further reduce the number of trains (previously assigned individually on routes) on a network basis [15], while in [16] a GA was designed for application on railway scheduling problems, achieving a good performance on real-world instances from the Spanish Manager of Railway Infrastructure (ADIF). Authors of [19] developed a GA-based Particle Swarm Optimization method in order to reschedule high-speed railway timetables while considering delays. The implementation was applied and tested in the Chinese Beijing-Shanghai high-speed railway corridor, managing to produce objective functions values which are at least 15% lower than other methods. Samà et al. [17] formulated the routing selection problem as a linear programming model and solved via an ant colony optimization algorithm. Ant colony optimization algorithms were also attempted to solve a simulated dynamic multiobjective railway rescheduling problem [21]. Bożejko et al. [18] applied the Dijkstra algorithm on railway networks in order to determine the fastest routes for rail freight transportation. In [19], the authors implemented a tabu search scheme for rerouting trains in their real-time traffic management system ROMA, generating solutions in 20 s and in 180 s for small and big instances respectively, a 15% improvement compared to previous versions of the system.

2.3 Reinforcement learning techniques in combination with Deep Learning

The resurgence of neural networks in the modern era of artificial intelligence gave way to the application of deep learning and learning methods in general, on the train-scheduling problem. Reinforcement learning is applied in [20] where deep neural networks are used in order to approximate Q-values, formulating the problem as a Markov decision process. Šemrov et al. [21] proposed a Q-learning algorithm for train rescheduling and compared the implementation with the rescheduling methods that do not rely on learning, showing that the solutions are at least equivalent or superior. In [26], a Q-learning based method was used for bidirectional railway lines (both single- and multi-track), achieving efficient scaling by making the size of the action and the state space indepent of the problem instance size. The applicability of the algorithm was demonstrated on two lines from the Indian railway network, showing the RL algorithm leads to higher solution quality than two prior heuristic-based approaches. Two deep Q-learning methods (a decentralized and a centralized approach) were applied and evaluated on the train dispatching problem [22], showcasing their supremacy over the classical linear Q-learning method. Authors in [23] developed artificial neural networks (ANN) which were trained and tested on real data from the Turkish State Railway, and reproduce the conflict resolutions of train dispatchers. The model was then used in comparison with a GA on the task of train re-scheduling, with the GA outperforming the ANN.

3 Problem Formulation

A railway network graph is modeled as a directed graph $G = (V, E)$, where V is the set of the vertices and E the set of the edges. A track segment, $s = (u, v)$, is an edge of E. Each track segment s is labeled with its length $l(s) \in R^+$, and the maximum allowed speed $vmax(s) \in R^+$ a train can reach while traversing it. A track segment can be connected at its ends with any number of other track segments; no inner connections are allowed. A $path$ is a list of consecutive segments $(s_1, s_2, \dots, s_k)$, $k \geq 1$, such that $s_i = (u_i, v_i)$ and for each i in $\{1\,..\,k-1\}$ we have $v_i = u_{i+1}$. Furthermore, for every $1 \leq i < j \leq k$, $u_i \neq u_j$, that is the sequence of segments do not cross (but potentially, $u_i \neq u_j$, $k > 1$, in which case we have a cycle).

A train T is described by its length $l(T) \in R^+$, its maximum speed $vmax(T) \in R^+$, its maximum acceleration $maxacc(T) \in R^+$, and its maximum deceleration $maxdec(T) \in R^+$. We use the functions $head(T)$ to denote the head point of T, and $tail(T)$ to denote the tail point of T.

The position of a train T, denoted by $pos(T)$, is defined as a pair (S, x), where $S = (s_1, s_2, \dots, s_k)$, with $k \geq 1$, is a path (that is, the train may span over multiple consecutive segments). Furthermore, $tail(T) \in s_1$ but $tail(T) \neq v_1$, and $head(T) \in s_k$ but $head(T) \neq u_k$. x is the distance of $head(T)$ from v_k across s_k. In case $k = 1$, $head(T)$ is considered to be closer to v_1 than $tail(T)$.

The state of a train T, denoted by $state(T)$, is the pair $(pos(T), speed(T))$, where $speed(T)$ is its current speed. We assume that $speed(T) \geq 0$, meaning that the train can move only forwards. We use also the notation $speed(T, t)$ to denote the speed of train T at a particular time point t. A valid state is a state that satisfies the following constraints:

$$0 \leq speed(T) \leq vmax(T)$$

and for each $i \in [1..k]$, where $(s_1, s_2, \dots, s_k)$ is the list of segments over which the train spans,

$$speed(T) \leq vmax(s_i)$$

A $Problem$ is defined by a tuple $(T, G, init, goal)$, where T is a train, with its given attributes, G is a directed graph with speed limits on its segments, $init$ is the initial state and $goal$ is the goal state.

A speed profile $SP_{t_1:t_2}$, or simply SP, is an infinite set of pairs $\{(state, t) :$ for every t such that $t_1 \leq t \leq t_2\}$, for some time points t_1 and t_2, such that $t_1 \leq t_2$. A speed profile $SP_{t_1:t_2}$ is valid if and only if (a) it consists of valid states for every t in $[t_1, t_2]$; (b) for every pair $(state', t')$ and $(state'', t'')$ in $SP_{t_1:t_2}$, if $t' \to t''$ then $state' \to state''$, where convergence between states is defined in the obvious way, that is, the difference in train position and speed between the two states tends to zero; and (c) for every pair $(state', t')$ and $(state'', t'')$ in $SP_{t_1:t_2}$, such that $t' < t''$, if $speed(T, t') < speed(T, t'')$ then $\frac{speed(T,t'')-speed(T,t')}{t''-t'} \leq maxacc(T)$, whereas if $speed(T, t') > speed(T, t'')$ then $\frac{speed(T,t')-speed(T,t'')}{t''-t'} \leq maxdec(T)$. Note that the third constraint subsumes the second one.

The duration of $SP_{t_1:t_2}$ is defined as $t_2 - t_1$. Moreover, we define three functions: $mint(SP_{t_1:t_2})$ that returns the minimum time label (t_1) in the speed profile; $maxt(SP_{t_1:t_2})$ that returns the maximum time label (t_2) in the speed profile; and $stateAt(SP_{t_1:t_2}, t)$ that returns the state $state(T)$ of T at time t, where $t_1 \leq t \leq t_2$.

Given a $Problem$, a pair $(path, SP)$ is a solution to the $Problem$ if (a) the segments in $init$ comprise the prefix of $path$; (b) the segments in $goal$ comprise the suffix of $path$; (c) the speed profile is valid; (d) $init = stateAt(mintime(SP))$; and (e) $goal = stateAt(maxtime(SP))$. The optimal solution to $Problem$ is the pair $(path, SP)$ with the minimum duration.

Moving on, we will make use of two simple examples with the aim of showcasing the operation of the algorithms. It is important to note that our approach makes use of algorithms of low complexity. In more detail, finding an optimal speed profile can be done in time linear in the length of the path, while the process of finding the optimal path has the same worst case algorithmic complexity as the Dijkstra algorithm. For that reason, an extensive experimental study was deemed to be unnecessary.

Example 1. We present an example of an optimal speed profile between an initial and a goal state over a given path (so, we do not have to search over alternative paths). Since the path is given, we are interested only in the speed profile. Table 1 details the path, while Table 2 gives the train T characteristics, as well as the initial and the goal states for this example.

Table 1. Path layout (Example 1)

Segment s	**Length l_s** (m)	$\boldsymbol{vmax(s)}$ (m/s)
s_0	150	20
s_1	150	40
s_2	700	50
s_3	800	80
s_4	1000	85
s_5	200	40
s_6	300	60
s_7	700	50
s_f	150	30

Table 2. Train T characteristics; $init$ and $goal$ states for Example 1

Parameters	**Value**
Length $l(T)$	150 m
Maximum speed $vmax(T)$	75 m/s
Acceleration $maxacc(T)$	1.5 m/s^2
Deceleration $maxdec(T)$	1 m/s^2
$init$	((s_0, 150), 0)
$goal$	((s_f, 0), 0)

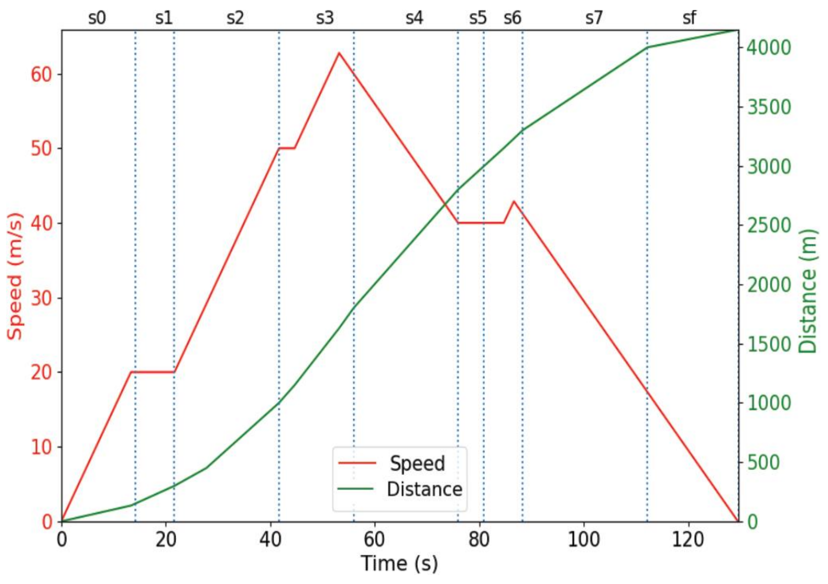


Figure 1. Speed and Positions vs. time graph

Figure 1 presents the optimal speed profile for the particular problem. Note that there are two vertical axes in the figure, the left (red) denoting the speed and the right (green) denoting the distance travelled by the train for any particular time. The distance in this diagram is measured in length units from the initial position of $head(T)$, which (in this example) is at the start of s_0 ($head(T) = u_0$) and is considered the 0 in the green axis. The vertical dotted lines show the time point in which $head(T)$ enters each segment (of course the tail of the train is still in previous segments).

It is worth noting that whenever the train leaves a segment with a speed limit that is lower than the speed limits of the segments of its new position, it starts accelerating (if not already accelerating). On the other hand, before the train enters a segment with a speed limit that is lower than its current speed, the train starts decelerating earlier enough, even several segments before the low speed limit segment. The same happens with the goal state, where (in the current example) the train should be idle: the train may start decelerating early enough.

Furthermore, in order for the speed profile to minimize the duration of the journey, the train must continuously maintain the maximum possible speed. This implies that whenever the train accelerates or decelerates, it does so at the maximum possible acceleration and deceleration rates. Thus why in Figure 1 the periods of acceleration have the same ascending slope, and the same happens with the periods of deceleration, which all have the same descending slope.

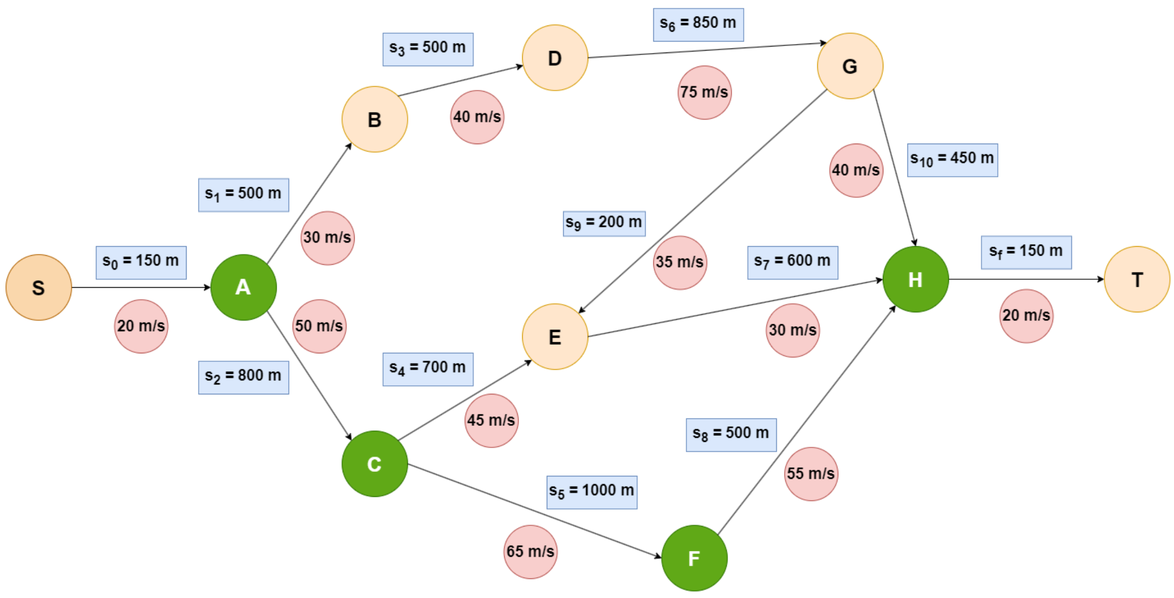


Figure 2. Calculating the shortest path from A to H

Example 2. Figure 2 presents an example railway network, consisting of 10 vertices and 12 edges. Each edge s is labeled with $l(s)$ and $vmax(s)$. The train T characteristics, as well as the goal state, remain the same as in Table 2, while $init = ((s_0, 0), 0)$.

The nodes marked with a green colour form the optimal solution path, with the corresponding optimal speed profile having a total duration of about 91 seconds. In this case, the solution path is not the shortest one in terms of length; indeed, path (S, A, C, E, H, T) is 200 meters shorter than the solution path (S, A, C, F, H, T). We will return to this example in Section 5 .

4 Methodology

This section presents the algorithms that were developed, in order to calculate the optimal speed profile along a designated route and the shortest path (in terms of time) throughout a railway network. Additionally, examples of use cases for both scenarios are provided, demonstrating in detail how each algorithm operates in basic problems.

4.1 Computing optimal speed profile over a given path

In this section, the computation of the optimal speed profile between $init$ and $goal$ is presented, assuming that there is a single path that can be used to reach the goal state, with no junctions across it. Before presenting the details of the computation, we give an overview of it.

There are three possible modes for a train: (a) Maintaining a steady speed; (b) accelerating; and, (c) decelerating. The train maintains a steady speed if it cannot accelerate and there is no need to decelerate. The need for deceleration arises from a subsequent segment (not necessarily the immediate next one) with a lower speed limit than the current speed of the train, or a goal state with a smaller goal speed (usually zero) than the current train speed. Acceleration occurs whenever the train is in a list of segments having all of them higher speed limits that its current speed, provided that there is no need to decelerate. Deceleration has priority over acceleration, which in turn has priority over maintaining steady speed. The computation's crucial part is to determine the point where deceleration should start.

The computation proceeds forwards. For each segment over the given path, it computes the optimum speed profile until $head(T)$ reaches the end of the segment, taking into account the already computed speed profiles over the previous segments of the given path. In case the new segment has a lower speed limit than the train speed when entering the segment, it is assumed that the train enters the new segment with a speed equal to its speed limit and, then, the algorithm computes where and when the train should have started decelerating, in order to satisfy the lower speed limit. This computation has as result changing the speed profile(s) of the previous segment(s).

Example 3. We present an example of the event in which the speed profile is recalculated, since the train reached a segment that has a speed limit lower than its current speed. Table 3 and Table 4 provide the path and the train T characteristics, as well as the $init$ and the $goal$ states for this example.

Table 3. Path layout (Example 3)

Segment s	**Length l_s** (m)	**$vmax(s)$** (m/s)
s_0	150	20
s_1	150	20
s_2	800	50
s_3	700	65
s_4	600	85
s_f	150	10

Table 4. Train T characteristics; $init$ and $goal$ states for Example 3

Parameters	**Value**
Length $l(T)$	150 m
Maximum speed $vmax(T)$	78 m/s
Acceleration $maxacc(T)$	1.5 m/s^2
Deceleration $maxdec(T)$	0.5 m/s^2
$init$	((s_0, 0), 0)
$goal$	((s_f, 0), 0)

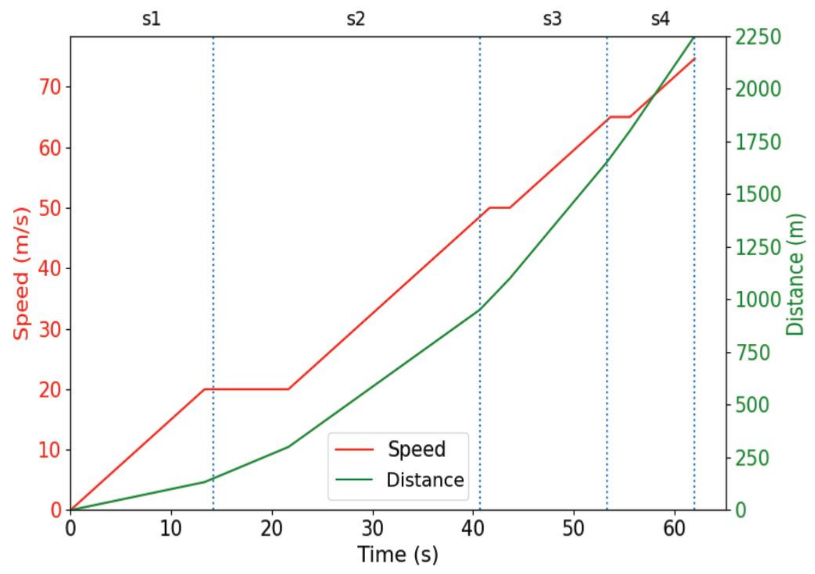


Figure 3. Speed Profile until reaching the end of s_4 (before backtrack)

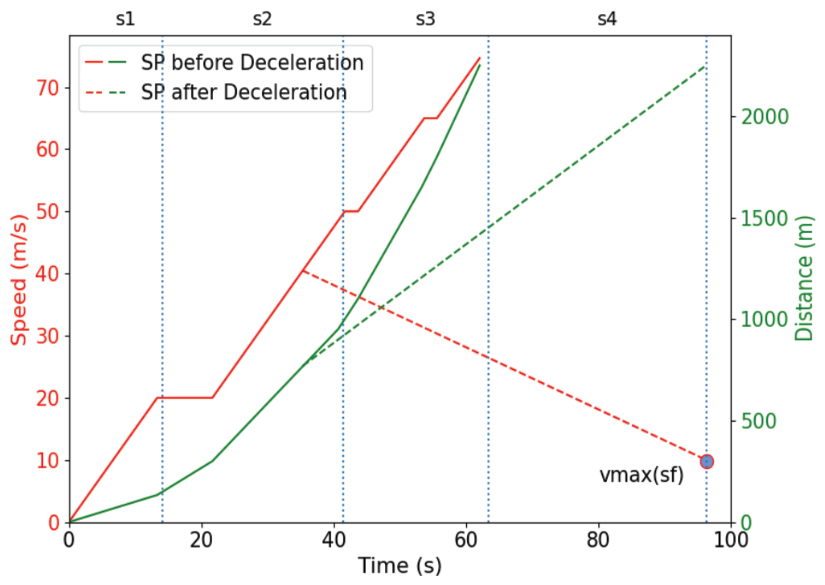


Figure 4. Speed Profile until reaching the end of s_4 (after backtrack)

Figure 3 presents the speed profile at the end of segment s_4; till that point the train mostly accelerates or maintains a steady speed due to the high speed limits of the previous segments. Therefore, according to Figure 3, the train is ready to exit s_4 with a speed over 70 m/s. However, the speed limit for the next segment is 10 m/s hence, the train needs to decelerate significantly. Considering that T enters s_4 with a speed of about 64 m/s, as well as $l(s_4)$ and $maxdec(T)$, deceleration must start earlier enough. Hence, the process is backtracked at previous segments until the deceleration line crosses the speed profile. The result of this procedure is shown in Figure 4. The train's deceleration starts at s_2 in order to reach the speed limit of s_f at the time it enters it, resulting in a slower speed profile. As a result, time expands from the deceleration starting point and onwards.

```
Algorithm 1. Finding the optimal speed profile
Input: $Path = (s_0, .., s_f), T,$
$init = ((s_0, .., s_k), initX), initV),$
$goal = ((s_m, .., s_f), goalX), goalV)$
Output: $SP$
 1: Initialize $currentV, hSgm, tSgm,$
    $hExit, tExit, currentAcc, SP, t_{total}$
 2: $speedLimit \leftarrow getSL(T, (s_{tSgm}, ..., s_{hSgm}))$
 3: $(t_{min}, t_1, t_2, t_3) \leftarrow getMinT(currentAcc, currentV,$
    $tExit, hExit, speedLimit)$
 4: while $SP[-1] \neq goal$ do
 5:     Proceed by $t_{min}$
 6:     $p \leftarrow currentV * t_{min} + (1/2) * currentAcc * t_{min}^2$
 7:     $currentV \leftarrow currentV + t_{min} * currentAcc$
 8:     $hExit \leftarrow hExit - p,\ tExit \leftarrow tExit - p$
 9:     $t_{total} \leftarrow t_{total} + t_{min}$
10:     if $t_2 = t_{min}$ then
11:         $hSgm \leftarrow hSgm + 1,\ hExit \leftarrow l(s_{hSgm})$
12:         $speedLimit \leftarrow getSL(T, (s_{tSgm}, ..., s_{hSgm}))$
13:         if $currentV > speedLimit$ then
14:             Apply Algorithm 2
15:             if Algorithm 2 returns False then
16:                 EXIT
17:             end if
18:         end if
19:     end if
20:     if $t_1 = t_{min}$ then
21:         $tSgm \leftarrow tSgm + 1,\ tExit \leftarrow l(s_{tSgm})$
22:         $speedLimit \leftarrow getSL(T, (s_{tSgm}, ..., s_{hSgm}))$
23:         if $currentV < speedLimit$ then
24:             $currentAcc = maxacc(T)$
25:         end if
26:     end if
27:     if $t_3 = t_{min}$ AND $currentV = speedLimit$ then
28:         $currentAcc = 0$
29:     end if
30:     $SP.append((Path_{tSgm:hSgm}, hExit), currentV, t_{total})$
31:     if $dist(SP[-1]) = dist(goal)$ AND $currentV \neq goalV$ then
32:         EXIT
33:     end if
34:     $(t_{min}, t_1, t_2, t_3) \leftarrow getMinT(currentAcc,$
    $currentV, tExit, hExit, speedLimit)$
35: end while
36: return $SP$
```

Algorithm 1. Finding the optimal speed profile

Algorithm 1 details the main procedure that iteratively detects events during the train's journey. An event happens in three cases: (a) $head(T)$ enters a new segment; (b) $tail(T)$ exits a segment; and (c) the train reaches the effective $speedLimit$ (which is the trains's speed limit or the minimum speed limit of the segments where the train spans). The input to this algorithm includes the train attributes, the given path with its segments' attributes as well as, the $init$ and $goal$ states. The position of train T in $init$ comprises a prefix of the path, whereas the position of train in $goal$ comprises a suffix of the path. The output of the algorithm is the optimal speed profile SP, in terms of journey duration. A speed profile is represented as a finite list of pairs $(state(T), t)$, with the train being in the same mode between subsequent states. Time at $init$ is considered 0. $SP[i]$ denotes the ith node of SP, with $SP[0] = (init, 0)$ and the last node of SP, denoted as $SP[-1]$, being of the form $(goal, t_{total})$ where t_{total} is the duration of the journey. Between $SP[i]$ and $SP[i + 1]$ the train remains in the same mode (acceleration, deceleration or steady speed), which is determined from the difference between the train speeds in the two subsequent nodes.

Concerning $goal$, the $head(T)$ position can be at any point of the last segment. Without loss of generality, we assume that in $goal$ the train must reach the end of the last segment (if this is not the case, we can split the last segment in two ones). Furthermore, we assume an additional segment, after the last one, with a speed limit equal to the desired speed of the train in $goal$. In this way, the $goal$ can be restated as having the train ready to enter this additional segment with a speed equal to its speed limit.

Note that in general the problem may not have a solution. This may happen in two cases: The first one is when the goal speed is too high and the train cannot accelerate early enough (because of either a low initial state speed or low speed limits in the segments before the goal) in order to reach it. The second scenario is when the initial speed is very high and the train cannot decelerate early enough, in order to enter a segment with a low speed limit or to reach the goal state with the desired slow speed.

```
function GETSL($T, (s_{tSgm}, ..., s_{hSgm})$)
    $lowestVmax \leftarrow \min(s_i), i \in [tSgm..hSgm]$
return $\min(lowestVmax, vmax(T))$
end function
```

Function 1. Get the current speed limit of T based on the segments it spans

```
function GETMINT($currentAcc, currentV,$
$tExit, hExit, speedLimit$)
    if $currentAcc > 0$ then
        $t_1 \leftarrow MaxRoot((1/2) * currentAcc, currentV,$
$- tExit)$
        $t_2 \leftarrow MaxRoot((1/2) * currentAcc, currentV,$
$- hExit)$
    else
        $t_1 \leftarrow tExit/currentV$
        $t_2 \leftarrow hExit/currentV$
    end if
    if $currentAcc > 0$ AND $speedLimit > currentV$ then
        $t_3 \leftarrow \frac{speedLimit - currentV}{currentAcc}$
    else
        $t_3 \leftarrow Infinite$
    end if
return $min(t_1, t_2, t_3), t_1, t_2, t_3$
end function
```

Function 2. Compute the time needed for T to reach each event

```
function MAXROOT($a, b, c$)
    $\Delta \leftarrow b^2 - 4ac$
    $t \leftarrow \frac{-b+\sqrt{\Delta}}{2a}$
return $t$
end function
```

Function 3. Compute the positive root of a quadratic equation

Algorithm 1 begins with initiliazing all the necessary variables. $currentV$ tracks the train's current speed and is initialized to be equal to the starting speed $initV$ of the $init$ state. $hSgm$ and $tSgm$ represent the indices of the segments in which $head(T)$ and $tail(T)$ reside respectively. At the $init$ state the train is in $(s_0, .., s_k)$ hence, $hSgm = k$ and $tSgm = 0$. $hExit$ and $tExit$ concern the distance of $head(T)$ and $tail(T)$ from the end of segments $hSgm$ and $tSgm$ respectively. The initial values for $hExit$ is $initX$ which are provided by $init$. If $hExit = 0$, $hSgm$ is incremented by one and $hExit$ is set to be equal to the length of next segment $l(s_{hSgm})$. $tExit$ is initialized using the following equation:

$$tExit\ = l(s_0) - \left(\sum_{i=0}^{k} l(s_i) - hExit - l(T)\right)$$

$currentAcc$ keeps the train's acceleration and is initialized to be equal to $maxacc(T)$. The $init$ state is stored in $SP[0]$. t_{total} is used to keep track of the duration so far and is initialized with 0. For the initialization of the $speedLimit$ that is in effect for the first event, the $getSl()$ function (Function 1) is used, that takes as input the train characteristics and the segments it is currently in. In case $speedLimit$ is equal to $currentV$, $currentAcc$ is set to 0.

Having initialized all variables, $getMinT()$ (Function 2) is called to detect which type of event is upcoming, by computing the time $t_{min} = \min(t_1, t_2, t_3)$ needed to reach each one of them. t_1 is the time needed for the tail to exit its segment; t_2 is the time needed for the head to enter a new segment; and t_3 is the time needed for the train to reach the $speedLimit$, provided that it is in accelerating mode. When the train is accelerating, t_1 and t_2 are computed as the roots of the quadratic equation:

$$\left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - distance = 0$$

$maxRoot()$ (Function 3) returns the positive root (there is always one). t_3 is computed as the time needed in order to accelerate from $currentV$ to the $speedLimit$ with an acceleration of $maxacc(T)$.

Having completed the preliminary computations, the while-loop updates all the relevant variables such as $currentV$, the distance travelled p that is used to update $tExit$ and $hExit$. Based on the type of the event that occurred, either $hSgm$ and $hExit$ or $tSgm$ and $tExit$ or none of them are updated. The train's $currentAcc$

is also updated accordingly; to 0 if the train reached the *speedLimit* and to *maxacc*(T) if *currentV* is lower than the *speedLimit*.

Algorithm 2 Finding the deceleration point

```
for $j \leftarrow length(SP) - 2\ to\ 0$ do
    $v_0 \leftarrow speedLimit$
    $v_1 \leftarrow speed(SP[j]),\ v_2 \leftarrow speed(SP[j+1])$
    if $v_2 \geq v_1$ then
        if $v_2 = v_1$ then
            $a = 0$
        else
            $a = maxacc(T)$
        end if
        $d_1 \leftarrow dist(SP[j]),\ d \leftarrow dist(SP[-1])$
        $x_{dec} \leftarrow \frac{v_0^2 - v_1^2 + 2(ad_1 + bd)}{2(a+b)}$
        $v^{\#} \leftarrow \sqrt{v_1^2 + 2a(x_{dec} - d_1)}$
        if $v^{\#} > v_1$ then
            $SP \leftarrow SP[0:j]$
            $t_1^{\#} \leftarrow \frac{v^{\#} - v_1}{a},\ t_2^{\#} \leftarrow \frac{v^{\#} - v_0}{b}$
            Add two new states in $SP$
            $currentV \leftarrow v_0,\ currentAcc \leftarrow 0$
            Return True
        end if
        if $v^{\#} = v_1$ then
            $SP \leftarrow SP[0:j]$
            if $x_{dec} > d_1$ then
                $t_1^{\#} \leftarrow \frac{x_{dec} - d_1}{v^{\#}},\ t_2^{\#} \leftarrow \frac{v^{\#} - v_0}{b}$
                Add two new states in $SP$
            else
                $t^{\#} \leftarrow \frac{v^{\#} - v_0}{b}$
                Add one new state in $SP$
            end if
            $currentV \leftarrow v_0,\ currentAcc \leftarrow 0$
            Return True
        end if
    end if
end for
return False
```

Algorithm 2. Finding the deceleration point

The most complicated case occurs when *head*(T) is about to enter a new segment ($t_2 = t_{min}$) with *currentV* higher than the new segment's speed limit *vmax*(s_{hSgm}), that is the new *speedLimit*. In this case the speed profile must be recomputed by adding a deceleration point at the latest possible time, when the train should start decelerating in order to reach the new segment with a speed equal to $speedLimit$ (denoted as v_0 in Algorithm 2). Intermediate speed profile entries are removed from SP. Algorithm 2 handles this case, by backtracking in previous states stored in SP, searching for the optimal point to decelerate. In order to access information stored in an entry i of SP, we use $dist(SP[i])$ for the total travelled distance, $speed(SP[i])$ for the train's current speed. Finally, $length(SP)$ denotes the total number of entries in SP.

The algorithm searches for previous speed profile segments during which the train was either accelerating or maintaining its speed, starting from the most recent one. Two consecutive speed profile nodes define a speed profile segment. The train's mode in a speed profile segment is determined by comparing its speed at the two defining speed profile nodes, (denoted by v_1 and v_2). If $v_1 > v_2$, meaning that at this segment the train was already decelerating, the for-loop continues to the previous speed profile segment (deceleration cannot start at a speed profile segment where the train was already decelerating) until an appropriate segment of the speed profile is found. If $v_2 \geq v_1$, d_1 becomes the cumulative travelled distance till $SP[j]$, while d represents the cumulative travelled distance at $SP[-1]$. The next step is to find whether the optimal deceleration point x_{dec} lies between these two positions, as well as computing the train's speed $v^{\#}$ at that point, with $d_1 \leq x_{dec} \leq d$ and $v_1 \leq v^{\#} \leq v_2$. There are two modes for the train's motion in this segment, since it will accelerate or keep steady until it reaches x_{dec} at time $t_1^{\#}$ with a speed of v_1 and then decelerate until d at time $t_2^{\#}$ with a speed of v_0.

If $v^{\#} > v_1$, this means that the deceleration point lies between d_1 and d. Existing SP nodes after $SP[j]$ are removed, while two new SP nodes are added, $SP[j+1]$ at the beginning of deceleration and $SP[j+2]$ at the end of it. In case $v^{\#}$ = v_1, the train maintains steady speed between $SP[j]$ and $SP[j+1]$. In case $x_{dec} =$

d_1, $SP[j+1]$ is removed, so $SP[j+2]$ becomes $SP[j+1]$. Finally, execution returns from Algorithm 2 on Algorithm 1 with the train at the start of the new segment having a speed equal to the new $speedLimit$ and and $currentAcc = 0$. If Algorithm 2 returns False, Algorithm 1 terminates since the train started the path with a speed high enough that a deceleration to $speedLimit$ is not possible.

Algorithm 1 ends by saving the new state in SP, proceeding by t_{min}. It is also checked whether the train has arrived at the position of the $goal$ state but with a speed different than $goalV$, which means that it is impossible to reach it thus, the algorithm terminates.

4.1.1 Use case: Computing the speed profile of Example 3

This section demonstrates the operation of Algorithm 1 and Algorithm 2, displaying the various calculations done as the train is progressing through the segments of the path. Table 3 and Table 4 provide the path and the train T characteristics, as well as the $init$ and the $goal$ states for this example.

Algorithm 1 employs an iterative while-loop mechanism to continuously detect events as the train moves through the segments of the path. For that reason, the computational analysis is presented for each iteration of the algorithm in order to simulate its operation.

Initialization. Before the iterative procedure of detecting events, it is necessary to initialize all the necessary variables. Since the train is idle in the $init$ state, $currentV = initV = 0$. Additionally, all parts of the train reside at s_0, hence $hSgm = tSgm = 0$. Due to the fact that the train is initially at the end of s_0, $init = ((s_0, 0), 0)$ is converted to an equal state $(((s_0, s_1), 150), 0)$ that places the train at the start of s_1. $hSgm$ is incremented by one and $hExit$ is set to be equal to the length of s_1. $tExit$ is initialized using the following equation:

$$tExit = l(s_0) - \left(\sum_{i=0}^{1} l(s_i) - hExit - l(T) \right) = 150 - ((150 + 150) - 150 - 150)$$
$$= 150\ m$$

SP is initialized using the converted $init$ state and t_{total} is set to 0. Since $vmax(s_0) = vmax(s_1) < vmax(T)$, $speedLimit$ is set to $vmax(s_1)$, while $currentAcc = maxacc(T)$, because $currentV < speedLimit$. The subsequent step involves identifying which one of the three types of events will occur next, by computing $t_{min} = \min(t_1, t_2, t_3)$.

t_1 and t_2 concern the time needed for $tail(T)$ to reach the end of s_0 and $head(T)$ to reach the end of s_1, respectively. They are computed as the roots of the quadratic equations:

$$t_1 \colon \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - tExit = 0$$
$$t_2 \colon \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - hExit = 0$$

Since $tExit = hExit = 150$:

$$t_1 = t_2 : \left(\frac{1}{2}\right) * 1.5 * t^2 + 0 * t - 150 = 0 => \frac{3t^2}{4} = 150 => t_{1,2} = \sqrt{200}$$
$$= 14.14\ s$$ [1]

t_3 is the time needed for the train to reach $maxv(T)$ and is calculated as such:

$$t_3 = \frac{speedLimit - currentV}{currentAcc} = \frac{20 - 0}{1.5} = 13.33\ s$$

[1] Note that for the sake of clarity in presentation, all arithmetic values are rounded to two decimal places.

Iteration 1.

Table 5. Values of the parameters at the beginning of Iteration 1

Parameters	**Value**
$SP[-1]$	$(((s_0, s_1), 150), 0)$
$hExit$	150 m
$tExit$	150 m
$hSgm$	1
$tSgm$	0
$currentV$	0 m/s
$currentAcc$	1.5 m/s^2
$speedLimit$	20 m/s
t_{total}	0 s
$t_{min}\ (t_3)$	13.33 s

Table 5 lists all the variable values utilized by Algorithm 1 throughout its operation. Since $t_{min} = t_3$, the algorithm detects that the next event will occur when the train reaches $maxv(T)$. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * {t_{min}}^2 = 0 * 13.33 + \left(\frac{1}{2}\right) * 1.5 * 13.33^2$$
$$= 133.33\ m$$

$currentV$ is set to be equal to $maxv(T)$, while $hExit = tExit = tExit - p = 150 - 133.33 = 16.66\ m$ and $t_{total} = t_{total} + t_3 = 0 + 13.33 = 13.33\ s$. The algorithm updates the train's mode since it reached its $maxv(T)$, by setting $currentAcc = 0$. SP is updated with the new state: $(((s_0, s_1), 16.66), 20)$.

Moving on, in order to detect the next event, t_1, t_2 and t_3 are calculated. Because $currentAcc = 0$, $t_1 = t_2 = \frac{tExit}{currentV} = \frac{hExit}{currentV} = \frac{16.66}{20} = 0.83\ s$, while $t_3 = Infinite$.

Iteration 2.

Table 6. Values of the parameters at the beginning of Iteration 2

Parameters	**Value**
$SP[-1]$	$(((s_0, s_1)$, 16.66), 20)
$hExit$	16.66 m
$tExit$	16.66 m
$hSgm$	1
$tSgm$	0
$currentV$	20 m/s
$currentAcc$	0 m/s^2
$speedLimit$	20 m/s
t_{total}	13.33 s
$t_{min}\ (t_{1,2})$	0.83 s

Table 6 contains the values for the updated parameters at the start of Iteration 2. Considering that $t_1 = t_2$, the next event concerns both $tail(T)$ reaching the end of s_0 and $head(T)$ reaching the end of s_1. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * {t_{min}}^2 = 20 * 0.83 + \left(\frac{1}{2}\right) * 0 * 0.83^2$$
$$= \ 16.66\, m$$

$currentV$ remains the same ($currentAcc = 0$), while $hExit = tExit = tExit - p = 16.66 - 16.66\ = 0\, m$ and $t_{total} = \ t_{total} + t_{1,2} = 13.33 +\ 0.83\ = 14.16\, s$.

Because of the first event ($t_2 = t_{min}$), $hSgm$ is incremented to 2 (the train is about to enter s_2) and $hExit$ is set to $l(s_2) = 800\, m$. At this point, due to the rare cooccurance of two events, $speedLimit$ is updated (line 12, Algorithm 1) even though tSgm is not yet updated. However, this issue is later resolved by reupdating $speedLimit$ after updating tSgm (line 22, Algorithm 1). Finally, since $currentV = speedLimit$, Algorithm 2 is not applied.

Moving on, $tail(T)$ is about to exit s_0 hence, $tSgm$ is set to 1 and $tExit$ is set to $l(s_1) = 150$. The new $speedLimit$ is $\min(vmax(s_1), vmax(s_2), vmax(T)) = \min(20, 50, 78) = 20\ m/s$, that is equal to $currentV$, hence $currentAcc$ remains 0. Finally, SP is updated with the new state: $(((s_1, s_2)$, 800), 20).

As regards the detection of the next event, considering that $currentAcc = 0$, t_3 is set to $infinite$, while t_1 and t_2 are calculated as such:

$$t_1 = \frac{tExit}{currentV} = \frac{150}{20} = 7.5\ s\ ,\ t_2 = \frac{hExit}{currentV} = \frac{800}{20} = 40\ s$$

Iteration 3.

Table 7. Entries of SP at the beginning of Iteration 3

Iteration	State
1	$(((s_0, s_1)$, 150), 0)
2	$(((s_0, s_1)$, 16.66), 20)
3	$(((s_1, s_2)$, 800), 20)

Table 8. Values of the parameters at the beginning of Iteration 3

Parameters	Value
$hExit$	800 m
$tExit$	150 m
$hSgm$	2
$tSgm$	1
$currentV$	20 m/s
$currentAcc$	0 m/s^2
$speedLimit$	20 m/s
t_{total}	14.16 s
$t_{min}\ (t_1)$	7.5 s

Table 7 includes all the entries of SP and Table 8 contains all the values for the updated parameters at the start of Iteration 3. Since $t_{min} = t_1$, the next event occurs when $tail(T)$ exits s_1. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * {t_{min}}^2 = 20 * 7.5 + \left(\frac{1}{2}\right) * 0 * 7.5^2$$
$$= \; 150\, m$$

$currentV$ remains the same ($currentAcc = 0$), while $hExit = hExit - p = 800 - 150 = 650\, m$, $tExit = tExit - p = 150 - 150 = 0\, m$ and $t_{total} = t_{total} + t_1 = 14.16 + 7.5 = 21.66\, s$.

$tail(T)$ is about to exit s_1, hence $tSgm$ is incremented to 2 and $tExit$ is set to $l(s_2) = 800\, m$. The new $speedLimit$ is $\min\big(vmax(s_2), vmax(T)\big) = \min(50, 78) = 50\frac{m}{s} > currentV$, hence $currentAcc = maxacc(T) = 1.5\, m/s^2$. Finally, SP is updated with the new state: ((s_2, 650), 20).

The next step concerns the detection of the next event, with t_1, t_2 and t_3 calculated as such:

$$t_1: \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - tExit = 0$$
$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 20 * t - 800 = 0 => \frac{3}{4}t^2 + 20t - 800 = 0$$
$$\Delta = 20^2 - 4 * \frac{3}{4} * (-800) = 2800$$
$$t_1 = \frac{-20 + \sqrt{2800}}{6/4} = 21.94\, s$$
$$t_2: \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - hExit = 0$$
$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 20 * t - 650 = 0 => \frac{3}{4}t^2 + 20t - 650 = 0$$
$$\Delta = 20^2 - 4 * \frac{3}{4} * (-650) = 2350$$
$$t_2 = \frac{-20 + \sqrt{2350}}{6/4} = 18.98\, s$$
$$t_3 = \frac{speedLimit - currentV}{currentAcc} = \frac{50 - 20}{1.5} = 20\, s$$

Iteration 4.

Table 9. Entries of SP at the beginning of Iteration 4

Iteration	State
1	$(((s_0, s_1), 150), 0)$
2	$(((s_0, s_1), 16.66), 20)$
3	$(((s_1, s_2), 800), 20)$
4	$((s_2, 650), 20)$

Table 10. Values of the parameters at the beginning of Iteration 4

Parameters	Value
$hExit$	650 m
$tExit$	800 m
$hSgm$	2
$tSgm$	2
$currentV$	20 m/s
$currentAcc$	1.5 m/s^2
$speedLimit$	50 m/s
t_{total}	21.66 s
t_{min} (t_2)	18.98 s

Table 9 includes all the entries of SP and Table 10 contains all the values for the updated parameters at the start of Iteration 4. The next event concerns $head(T)$ entering s_3. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * {t_{min}}^2 = 20 * 18.98 + \left(\frac{1}{2}\right) * 1.5 * 18.98^2$$
$$= \quad 650\ m$$

Since the train is accelerating, the updated $currentV = currentV + t_{min} * currentAcc = 20 + 18.98 * 1.5 = 48.47\ m/s,$ while $hExit = hExit - p = 650\ -$

$650 \ = 0\,m,\ \ tExit = tExit - p = 800 \ - 650 \ = 150\,m$ and $t_{total} = t_{total} + t_2 = 21.66 + 18.98 \ = 40.65\,s$.

$head(T)$ is about to enter s_3, hence $hSgm$ is incremented to 3 and $hExit$ is set to $l(s_3) = 700\,m$. The new $speedLimit$ is $\min\big(vmax(s_2), vmax(s_3), vmax(T)\big) = \min(50, 65, 78) = 50\frac{m}{s} > currentV$, hence $currentAcc = maxacc(T) = 1.5\,m/s^2$. Finally, SP is updated with the new state: $(((s_2, s_3)$, 700), 48.47).

In the final step of Iteration 4, the next event is detected by calculating t_1, t_2 and t_3 as such:

$$t_1 \colon \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - tExit = 0$$

$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 48.47 * t - 150 = 0 => \frac{3}{4}t^2 + 48.47t - 150 = 0$$

$$\Delta = \ 48.47^2 - 4 * \frac{3}{4} * (-150) = 2799.34$$

$$t_1 = \frac{-48.47 + \ \sqrt{2799.34}}{6/4} = 2.95\,s$$

$$t_2 \colon \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - hExit = 0$$

$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 48.47 * t - 700 = 0 => \frac{3}{4}t^2 + 48.47t - 700 = 0$$

$$\Delta = \ 48.47^2 - 4 * \frac{3}{4} * (-700) = 4449.34$$

$$t_2 = \frac{-48.47 + \ \sqrt{4449.34}}{6/4} = 12.15\,s$$

$$t_3 = \frac{speedLimit - currentV}{currentAcc} = \frac{50 - 48.47}{1.5} = 1.02\,s$$

Iteration 5.

Table 11. Entries of SP at the beginning of Iteration 5

Iteration	State
1	$(((s_0, s_1), 150), 0)$
2	$(((s_0, s_1), 16.66), 20)$
3	$(((s_1, s_2), 800), 20)$
4	$((s_2, 650), 20)$
5	$(((s_2, s_3), 700), 48.47)$

Table 12. Values of the parameters at the beginning of Iteration 5

Parameters	Value
$hExit$	700 m
$tExit$	150 m
$hSgm$	3
$tSgm$	2
$currentV$	48.47 m/s
$currentAcc$	1.5 m/s^2
$speedLimit$	50 m/s
t_{total}	40.65 s
t_{min} (t_3)	1.02 s

The next event occurs when the train reaches $maxv(T)$. The distance p until this event is: $p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * {t_{min}}^2 = 48.47 * 1.02 + \left(\frac{1}{2}\right) * 1.5 * 1.02^2 = 50\ m.$

Since the train is accelerating, the updated $currentV = currentV + t_{min} * currentAcc = 48.47 + 1.02 * 1.5 = 50\ m/s,$ while $hExit = hExit - p = 700\ -$

$50 = 650\ m,\quad tExit = tExit - p = 150\ - 50 = 100\ m$ and $t_{total} = t_{total} + t_3 = 40.65 + 1.02\ = 41.67\ s.$

Since the train reached its maximum speed, $currentAcc = 0$. Additionally, SP is updated with the new state: $(((s_2, s_3)$, 650), 50). The final step of Iteration 5 concerns the detection of the next step by calculating t_1 and t_2 (t_3 is set to $infinite$) as such: $t_1 = \frac{tExit}{currentV} = \frac{100}{50} = 2\ s\ , t_2 = \frac{hExit}{currentV} = \frac{650}{50} = 13\ s.$

Iteration 6.

Table 13. Entries of SP at the beginning of Iteration 6

Iteration	State
1	$(((s_0, s_1)$, 150), 0)
2	$(((s_0, s_1)$, 16.66), 20)
3	$(((s_1, s_2)$, 800), 20)
4	$((s_2$, 650), 20)
5	$(((s_2, s_3)$, 700), 48.47)
6	$(((s_2, s_3)$, 650), 50)

Table 14. Values of the parameters at the beginning of Iteration 6

Parameters	Value
$hExit$	650 m
$tExit$	100 m
$hSgm$	3
$tSgm$	2
$currentV$	50 m/s
$currentAcc$	0 m/s^2
$speedLimit$	50 m/s
t_{total}	41.67 s
$t_{min}\ (t_1)$	2 s

Since $t_{min} = t_1$, the next event concerns $tail(T)$ exiting s_2. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * {t_{min}}^2 = 50 * 2 + \left(\frac{1}{2}\right) * 1.5 * 2^2$$
$$= 100\ m$$

$currentAcc = 0$, hence $currentV$ remains the same, while $hExit = hExit - p = 650 - 100 = 550\ m,$ $tExit = tExit - p = 100 - 100 = 0\ m$ and $t_{total} = t_{total} + t_1 = 41.67 + 2 = 43.67\ s.$

$tail(T)$ is about to exit s_2, hence $tSgm$ is incremented to 3 and $tExit$ is set to $l(s_3) = 700\ m.$ The new $speedLimit$ is $\min\big(vmax(s_3), vmax(T)\big) = \min(65\,, 78) = 65\frac{m}{s} > currentV,$ hence $currentAcc = maxacc(T) = 1.5\ m/s^2.$ Finally, SP is updated with the new state: ((s_3, 550), 50).

In the final step of Iteration 6, the next event is detected by calculating t_1, t_2 and t_3 as such:

$$t_1 : \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - tExit = 0$$
$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 50 * t - 700 = 0 => \frac{3}{4}t^2 + 50t - 700 = 0$$
$$\Delta = 50^2 - 4 * \frac{3}{4} * (-700) = 4600$$
$$t_1 = \frac{-50 + \sqrt{4600}}{6/4} = 11.88\ s$$
$$t_2 : \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - hExit = 0$$
$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 50 * t - 550 = 0 => \frac{3}{4}t^2 + 50t - 550 = 0$$
$$\Delta = 50^2 - 4 * \frac{3}{4} * (-550) = 4150$$
$$t_2 = \frac{-50 + \sqrt{4150}}{6/4} = 9.61\ s$$
$$t_3 = \frac{speedLimit - currentV}{currentAcc} = \frac{65 - 50}{1.5} = 10\ s$$

Iteration 7.

Table 15. Entries of SP at the beginning of Iteration 7

Iteration	**State**
1	$(((s_0, s_1), 150), 0)$
2	$(((s_0, s_1), 16.66), 20)$
3	$(((s_1, s_2), 800), 20)$
4	$((s_2, 650), 20)$
5	$(((s_2, s_3), 700), 48.47)$
6	$(((s_2, s_3), 650), 50)$
7	$((s_3, 550), 50)$

Table 16. Values of the parameters at the beginning of Iteration 7

Parameters	**Value**
$hExit$	550 m
$tExit$	700 m
$hSgm$	3
$tSgm$	3
$currentV$	50 m/s
$currentAcc$	1.5 m/s^2
$speedLimit$	65 m/s
t_{total}	43.67 s
$t_{min}\ (t_2)$	9.61 s

The next event occurs when $head(T)$ enters s_4. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * {t_{min}}^2 = 50 * 9.61 + \left(\frac{1}{2}\right) * 1.5 * 9.61^2$$
$$= 550\ m$$

Since the train is in accelerating mode, $currentV = currentV + t_{min} * currentAcc = 50 + 9.61 * 1.5 = 64.42\, m/s,$ while $hExit = hExit - p = 550 - 550 = 0\, m,$ $tExit = tExit - p = 700 - 550 = 150\, m$ and $t_{total} = t_{total} + t_2 = 43.67 + 9.61 = 53.28\, s$.

$head(T)$ is about to enter s_4, hence $hSgm$ is incremented to 4 and $hExit$ is set to $l(s_4) = 600\, m$. The new $speedLimit$ is $\min\big(vmax(s_3), vmax(s_4), vmax(T)\big) = \min(65, 85, 78) = 65 \frac{m}{s} > currentV,$ hence $currentAcc = maxacc(T) = 1.5\, m/s^2$. Finally, SP is updated with the new state: $(((s_3, s_4), 600), 64.42)$.

Iteration 7 concludes by calculating t_1, t_2 and t_3 as such:

$$t_1 : \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - tExit = 0$$

$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 64.42 * t - 150 = 0 => \frac{3}{4} t^2 + 64.42t - 150 = 0$$

$$\Delta = 64.42^2 - 4 * \frac{3}{4} * (-150) = 4599.93$$

$$t_1 = \frac{-64.42 + \sqrt{4599.93}}{6/4} = 2.26\, s$$

$$t_2 : \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - tExit = 0$$

$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 64.42 * t - 600 = 0 => \frac{3}{4} t^2 + 64.42t - 600 = 0$$

$$\Delta = 64.42^2 - 4 * \frac{3}{4} * (-600) = 5949.93$$

$$t_2 = \frac{-64.42 + \sqrt{5949.93}}{6/4} = 8.47\, s$$

$$t_3 = \frac{speedLimit - currentV}{currentAcc} = \frac{65 - 64.42}{1.5} = 0.38\, s$$

Iteration 8.

Table 17. Entries of SP at the beginning of Iteration 8

Iteration	**State**
1	$(((s_0, s_1), 150), 0)$
2	$(((s_0, s_1), 16.66), 20)$
3	$(((s_1, s_2), 800), 20)$
4	$((s_2, 650), 20)$
5	$(((s_2, s_3), 700), 48.47)$
6	$(((s_2, s_3), 650), 50)$
7	$((s_3, 550), 50)$
8	$(((s_3, s_4), 600), 64.42)$

Table 18. Values of the parameters at the beginning of Iteration 8

Parameters	**Value**
$hExit$	600 m
$tExit$	150 m
$hSgm$	4
$tSgm$	3
$currentV$	64.42 m/s
$currentAcc$	1.5 m/s^2
$speedLimit$	65 m/s
t_{total}	53.28 s
t_{min} (t_3)	0.38 s

The next event occurs when the train reaches $maxv(T)$. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * {t_{min}}^2 = 64.42 * 0.38 + \left(\frac{1}{2}\right) * 1.5 * 0.38^2$$
$$= 25\, m$$

Since the train is in accelerating mode, $currentV = currentV + t_{min} * currentAcc = 64.42 + 0.38 * 1.5 = 65\, m/s,$ while $hExit = hExit - p = 600 - 25 = 575\, m,$ $tExit = tExit - p = 150 - 25 = 125\, m$ and $t_{total} = t_{total} + t_3 = 53.28 + 0.38 = 53.66\, s$. The algorithm updates the train's mode since it reached its $maxv(T)$, by setting $currentAcc = 0$. SP is updated with the new state: (((s_3, s_4), 575), 65).

As regards the detection of the next event, considering that $currentAcc = 0$, t_3 is set to $infinite$, while t_1 and t_2 are calculated as such:

$$t_1 = \frac{tExit}{currentV} = \frac{125}{65} = 1.92\, s$$
$$t_2 = \frac{hExit}{currentV} = \frac{575}{65} = 8.84\, s$$

Iteration 9.

Table 19. Entries of SP at the beginning of Iteration 9

Iteration	State
1	(((s_0, s_1), 150), 0)
2	(((s_0, s_1), 16.66), 20)
3	(((s_1, s_2), 800), 20)
4	((s_2, 650), 20)
5	(((s_2, s_3), 700), 48.47)
6	(((s_2, s_3), 650), 50)
7	((s_3, 550), 50)
8	(((s_3, s_4), 600), 64.42)
9	(((s_3, s_4), 575), 65)

Table 20. Values of the parameters at the beginning of Iteration 9

Parameters	**Value**
$hExit$	575 m
$tExit$	125 m
$hSgm$	4
$tSgm$	3
$currentV$	65 m/s
$currentAcc$	0 m/s^2
$speedLimit$	65 m/s
t_{total}	53.66 s
t_{min} (t_1)	1.92 s

The next event occurs when $tail(T)$ exits s_3. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * {t_{min}}^2 = 65 * 1.92 + \left(\frac{1}{2}\right) * 0 * 1.92^2$$
$$= 125\ m$$

The train is not accelerating, hence $currentV$ remains steady, while $hExit = hExit - p = 575 - 125 = 450\ m,$ $tExit = tExit - p = 125 - 125 = 0\ m$ and $t_{total} = t_{total} + t_1 = 53.66 + 1.92 = 55.58\ s$.

$tail(T)$ is about to exit s_3, hence $tSgm$ is incremented to 4 and t$Exit$ is set to $l(s_4) = 600\ m$. The new $speedLimit$ is $\min\big(vmax(s_4), vmax(T)\big) = \min(85, 78) = 78\frac{m}{s} > currentV,$ hence $currentAcc = maxacc(T) = 1.5\ m/s^2$. Finally, SP is updated with the new state: ((s_4, 450), 65).

Iteration 9 concludes by calculating t_1, t_2 and t_3 as such:

$$t_1: \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - tExit = 0$$

$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 65 * t - 600 = 0 => \frac{3}{4}t^2 + 65t - 600 = 0$$

$$\Delta = 65^2 - 4 * \frac{3}{4} * (-600) = 6025$$

$$t_1 = \frac{-65 + \sqrt{6025}}{6/4} = 8.41\ s$$

$$t_2 : \left(\frac{1}{2}\right) * currentAcc * t^2 + currentV * t - tExit = 0$$

$$=> \left(\frac{1}{2}\right) * 1.5 * t^2 + 65 * t - 450 = 0 => \frac{3}{4}t^2 + 65t - 450 = 0$$

$$\Delta = 65^2 - 4 * \frac{3}{4} * (-450) = 5575$$

$$t_2 = \frac{-65 + \sqrt{5575}}{6/4} = 6.44\ s$$

$$t_3 = \frac{speedLimit - currentV}{currentAcc} = \frac{78 - 65}{1.5} = 8.66\ s$$

Iteration 10.

Table 21. Entries of SP at the beginning of Iteration 10

Iteration	State
1	$(((s_0, s_1), 150), 0)$
2	$(((s_0, s_1), 16.66), 20)$
3	$(((s_1, s_2), 800), 20)$
4	$((s_2, 650), 20)$
5	$(((s_2, s_3), 700), 48.47)$
6	$(((s_2, s_3), 650), 50)$
7	$((s_3, 550), 50)$
8	$(((s_3, s_4), 600), 64.42)$
9	$(((s_3, s_4), 575), 65)$
10	$((s_4, 450), 65)$

Table 22. Values of the parameters at the beginning of Iteration 10

Parameters	**Value**
$hExit$	450 m
$tExit$	600 m
$hSgm$	4
$tSgm$	4
$currentV$	65 m/s
$currentAcc$	1.5 m/s^2
$speedLimit$	78 m/s
t_{total}	55.58 s
t_{min} (t_2)	6.44 s

The next event occurs when $head(T)$ enters s_f. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * t_{min}{}^2 = 65 * 6.44 + \left(\frac{1}{2}\right) * 1.5 * 6.44^{\,2}$$
$$= \ 450\, m$$

Since the train is in accelerating mode, $currentV = currentV + t_{min} * currentAcc = 65 + 6.44 * 1.5 = 74.66\, m/s,$ while $hExit = hExit - p = 450 - 450 = 575\, m,$ $tExit = tExit - p = 600 - 450 = 150\, m$ and $t_{total} = t_{total} + t_2 = 55.58 + 6.44 = 62.03\, s.$

$head(T)$ is about to enter s_f, hence $hSgm$ is incremented to 5 and h$Exit$ is set to $l(s_5) = 150\, m.$ The new $speedLimit$ is $\min\left(vmax(s_4), vmax(s_f), vmax(T)\right) = \min(85, 10, 78) = 10\,\frac{m}{s}$. SP is updated with the new state: ((s_f, 150), 74.66). At this point, it is checked whether Algorithm 2 must be applied or not, by comparing the effective $speedLimit$ and $currentV$. Due to the fact that $currentV$ is higher than the $speedLimit$ (the train is about to enter a segment with a speed limit of 10 m/s having a speed of 74.66 m/s), Algorithm 2 is applied.

Algorithm 2 starts searching through previous speed profile segments (defined by two consecutive speed profile nodes), in order to find the optimal point to decelerate. v_0 represents the desired speed for the train at the start of s_f, that is $vmax(s_f) = 10\ m/s$. Finally, j is used as an index of the SP list, initialized as $length(SP) - 2 = 11 - 2 = 9$.

Algorithm 2 Iteration 1. The process begins by examining the last two entries of SP: $SP[j] = \big((s_4, 450), 65\big)$, $SP[j+1] = \Big(\big(s_f, 150\big), 74.66\Big)$. v_1 and v_2 are set to 65 and 74.66 m/s, respectively. Since $v_2 > v_1$, the train was in accelerating mode and the algorithm continues, by setting d_1 (cumulative travelled distance until $SP[j]$) to 1950 m and d (cumulative travelled distance at $SP[j+1]$) to 2400 m. Next, based on the train's mode in the current speed profile segment, $currentAcc = maxacc(t)$ and the deceleration point x_{dec} is computed as such:

$$x_{dec} = \frac{{v_0}^2 - {v_1}^2 + 2 * (currentAcc * d_1 + maxdec(T) * d)}{2 * (currentAcc + maxdec(T))}$$

$$= \frac{10^2 - 65^2 + 2 * (1.5 * 1950 + 0.5 * 2400)}{2 * (1.5 + 0.5)} = \frac{4125}{4} = 1031.25\ m$$

The speed of the train at x_{dec}, that is $v^{\#}$, is computed as such:

$$v^{\#} = \sqrt{v_1^2 + 2 * currentAcc * (x_{dec} - d_1)} = \sqrt{65^2 + 2 * 1.5 * (1031.25 - 1950)}$$

$$= 38.32\ m/s$$

Since $v^{\#} < v_1$, the train's deceleration should have started at previous speed profile segments, hence Algorithm 2 continues its iteration. Table 23 contains the calculations of Algorithm 2 for all previous speed profile segments until the optimal deceleration point is found. The procedure lasted a total of 7 iterations, the algorithm found the optimal deceleration point at s_2, thus completing the backtracking procedure. In that speed profile segment, instead of accelerating until reaching a speed of 48.47 m/s, the train will now accelerate to a speed of 40.46 m/s and then immediately start decelerating, until it reaches the start of s_f, with the desired speed of 10 m/s. As mentioned previously, Figure 3 presents the speed profile at the end of segment s_4, as the train is about to enter s_f and violate its speed limit. The result of the backtracking procedure is shown in Figure 4.

Table 23. Values of the backtracking variables computed by Algorithm 2

Iteration	**j**	$\boldsymbol{SP[j]}$	$\boldsymbol{SP[j+1]}$	$\boldsymbol{d_1}$	$\boldsymbol{x_{dec}}$	$\boldsymbol{v^\#}$
2	8	$(((s_3, s_4), 575), 65)$	$((s_4, 450), 65)$	1825	-1725	65
3	7	$(((s_3, s_4), 600), 64.42)$	$(((s_3, s_4), 575), 65)$	1800	937.5	39.52
4	6	$((s_3, 550), 50)$	$(((s_3, s_4), 600), 64.42)$	1250	937.5	39.52
5	5	$(((s_2, s_3), 650), 50)$	$((s_3, 550), 50)$	1150	0	50
6	4	$(((s_2, s_3), 700), 48.47)$	$(((s_2, s_3), 650), 50)$	1100	862.5	40.46
7	**3**	$\boldsymbol{((s_2, 650), 20)}$	$\boldsymbol{(((s_2, s_3), 700), 48.47)}$	**450**	**862.5**	**40.46**

The final step of Algorithm 2 concerns updating SP with the changes produced by the backtracking procedure. Firstly, all entries of SP after j are removed. Similarly, t_{total} is reverted back to the value it had at state $SP[3]$, that is $21.66\, s$. Moving on, since $v^\# > v_1$, two new states will be added to SP, one after the train's acceleration until reaching x_{dec} and one after the train's deceleration until the start of s_f. The train accelerates for $t_1^\# = \frac{v^\# - v_1}{maxacc(T)} = \frac{40.46-20}{1.5} = 13.64\, s$, travelling $currentV * t_1^\# + \frac{1}{2} * currentAcc * (t_1^\#)^2 = 20 * 13.64 + \frac{1}{2} * 1.5 * 13.64^2 = 412.5\, m$, which means that at the deceleration point, $hExit = 650 - 412.5 = 237.5\, m$, leading to the first new state: $((s_2, 237.5), 40.46)$. Afterwards, the train decelerates for $t_2^\# = \frac{v^\# - v_0}{maxdec(T)} = \frac{40.46-10}{0.5} = 60.93\, s$, travelling $currentV * t_2^\# - \frac{1}{2} * currentAcc * (t_2^\#)^2 = 40.46 * 60.93 - \frac{1}{2} * 0.5 * 60.93^2 = 1537.5\, m$, until it reaches the start of s_f, leading to the second new state: (((s_4, s_f), 150), 10). Finally, $t_{total} = t_{total} + t_1^{\#} + t_2^{\#} = 21.66 + 13.64 + 60.93 = 96.23\ s$, $currentV$ takes the value of v_0, $currentAcc = 0$ and execution returns from Algorithm 2 to Algorithm 1.

As regards the detection of the next event, considering that $currentAcc = 0$, t_3 is set to $infinite$, while t_1 and t_2 are calculated as such:

$$t_1 = \frac{tExit}{currentV} = \frac{150}{10} = 15\ s\ ,\ t_2 = \frac{hExit}{currentV} = \frac{150}{10} = 15\ s$$

Iteration 11.

Table 24. Entries of SP at the beginning of Iteration 11

Iteration	State
1	(((s_0, s_1), 150), 0)
2	(((s_0, s_1), 16.66), 20)
3	(((s_1, s_2), 800), 20)
4	((s_2, 650), 20)
10	((s_2, 237.5), 40.46).
10	(((s_4, s_f), 150), 10)

Table 25. Values of the parameters at the beginning of Iteration 11

Parameters	Value
$hExit$	150 m
$tExit$	300 m
$hSgm$	5
$tSgm$	4
$currentV$	10 m/s
$currentAcc$	0 m/s^2
$speedLimit$	10 m/s
t_{total}	96.23 s
t_{min} ($t_{1,2}$)	15 s

The next events occur simultaneously when $head(T)$ reaches the end of s_f and $tail(T)$ reaches the start of s_f. The distance p until this event is:

$$p = currentV * t_{min} + \left(\frac{1}{2}\right) * currentAcc * t_{min}{}^2 = 10 * 15 + \left(\frac{1}{2}\right) * 0 * 15^{\ 2}$$
$$= \ 150\, m$$

$currentV$ remains at 10 m/s since $currentAcc = 0$, while the new $speedLimit = goalV = 0\frac{m}{s} < currentV$. Hence, Algorithm 2 is applied once again since the train is about to reach the $goal$ state with a speed of 10 m/s, that is higher than $goalV = 0\, m/s$.

At this case, the first speed profile segment examined by Algorithm 2 can accommodate the train's deceleration. Table 26 contains the result of Algorithm 2. $v^{\#} = v_1$ and $x_{dec} > d_1$, hence after removing $SP[j+1]$, two new states are added, one after the train reaches x_{dec} with a steady speed and another when it finishes its deceleration at the end of s_f. Similarly, t_{total} is reverted back to the value it had at the state $SP[j]$, that is 96.23 s.

Table 26. Values of the backtracking variables computed by Algorithm 2

Iteration	j	$SP[j]$	$SP[j+1]$	d_1	x_{dec}	$v^{\#}$
1	4	$(((s_4, s_f),\ 150), 10)$	$(((s_f),\ 0), 10)$	2400	2450	10

The train will maintain its speed until x_{dec}, for $t_1^{\#} = \frac{x_{dec} - d_1}{currentV} = \frac{2450-2400}{10} = 5\, s$, travelling $currentV * t_1^{\#} = 50$ m and leading to the first new state: $((s_f, 100), 10)$. Moving on, the train will start decelerating for $t_2^{\#} = \frac{v^{\#} - v_0}{maxdec(T)} = \frac{10-0}{0.5} = 20\, s$, travelling $currentV * t_2^{\#} - \frac{1}{2} * currentAcc * (t_2^{\#})^2 = 10 * 20 - \frac{1}{2} * 0.5 * 20^2 = 100\, m$, until it reaches the start of s_f, leading to the second new state: $((s_f,\ 0), 0)$, that is the $goal$ state. Finally, $t_{total} = 96.23 + 5 + 20 = 121.24\, s, currentV$ takes the value of v_0, $currentAcc = 0$ and execution returns from Algorithm 2 to Algorithm 1. Since $SP[-1] = goal$, the while-loop of Algorithm 1 terminates.

Table 27 includes the result of Algorithm 1, that is the optimal speed profile for Example 3.

Table 27. The final entries of SP at the end of Algorithm 1

Iteration	State
1	$(((s_0, s_1), 150), 0)$
2	$(((s_0, s_1), 16.66), 20)$
3	$(((s_1, s_2), 800), 20)$
4	$((s_2, 650), 20)$
10	$((s_2, 237.5), 40.46)$.
10	$(((s_4, s_f), 150), 10)$
11	$((s_f, 100), 10)$
11	$((s_f, 0), 0)$

4.2 Shortest Path Finding

```
Algorithm 3 Shortest path with A*
Input: $T, G, init = ((s_0, .., s_k), initX), initV)$,
$goal = ((s_m, .., s_f), goalX), goalV)$
Output: $SP_{dict}[min(openDict)], min(openDict)$
1: Initialize $start$, $openDict$.
2: $openDict \leftarrow generatePaths(G, start)$
3: $openDict \leftarrow expandPaths(G, openDict)$
4: For $path \in openDict$, $openDict[path] = f(path)$
5: Update $SP_{dict}$.
6: while $openDict$ is not empty do
7:     $current \leftarrow min(openDict)$.
8:     $openDict \leftarrow openDict - current$
9:     $current \leftarrow generatePaths(G, current)$
10:    $successors \leftarrow expandPaths(G, current)$
11:    For $path \in successors$, $successors[path] = f(path)$
12:    Update $SP_{dict}$.
13:    $openDict \leftarrow openDict \cup successors$
14:    $lastState \leftarrow (SP_{dict}[min(successors)])[-1]$
15:    if $lastState = goal$ then
16:        return $SP_{dict}[min(successors)], min(successors)$.
17:    end if
18: end while
```

Algorithm 3. Shortest path with A*

In this section we employ the A* algorithm to compute the shortest path in terms of time for the general form of the problem, that is over an arbitrary graph. Algorithm 3 takes as input a *Problem* (T, G, init, goal) and outputs the solution, that is the pair (*path*, *SP*) (if it exists). *openDict* contains all the discovered paths and is initialized to include only the *start* path, that is a sequence of nodes $(u_0, u_1, \dots, u_k, s_k)$, defined by the segments $(s_0, .., s_k)$ in *init*. For each path in *openDict*, *generatePaths*() (Function 4) iterates through the outgoing edges (of the last vertex of the path), generating new paths (which are extensions of the current path, with a single or more segments added to it) that are stored temporarily in *neighbors*, before entering *openDict*. In the example of Figure 2, $s_k = s_0$ with $s_0 = (S, A)$ hence, the *start* path is (S, A). Using the neighboring nodes B and C, extended paths (S, A, B) and (S, A, C) are created.

```
function GENERATEPATHS($G, path$)
    $node \leftarrow path[-1]$
    for $(u, v)$ $in$ $G.edges$ do
        if $u = node$ then
            $path \leftarrow (node, v)$
            $neighbors.insert(path)$
        end if
    end for
return $neighbors$
end function
```

Function 4. Connect the last vertex of *path* with its neighboring nodes

```
function EXPANDPATHS($G, paths$)
    for $path$ $in$ $paths$ do
        $v \leftarrow path[-1]$
        while $v$ not a junction AND $v \neq v_f$ do
            $extension \leftarrow generatePaths(G, v)$
            $v \leftarrow extension[-1]$
            $path \leftarrow path \cup v$
        end while
        $expandedPaths.insert(path)$
    end for
return $expandedPaths$
end function
```

Function 5. Expand all paths in *paths* until the first junction

In the next step, these paths are further expanded until the first junction, by applying the $expandPaths()$ function (Function 5). This function takes as input a list of $paths$ and for each $path$ (saved as a sequence), it iterativily expands it by applying the $generatePaths()$ function on its last node v. If v has a single outgoing edge, it has a single neighbor and the $generatePaths()$ function returns only one path (of two nodes) that is stored in the set $extension$. The last element of $extension$ is the new node, that is added to the $path$. Moreover, if v has multiple outgoing edges or it is the $destination$ node (the end of segment s_f, that is v_f), the path is not expanded. The $expandedPaths$ are stored in $openDict$. In our example this means that the path (S, A, B) is expanded into (S, A, B, D, G) while path (S, A, C) remains unchanged, since node C has more than one outgoing edges.

As mentioned previously, Algorithm 1 stores train states in the list SP. In contrast, Algorithm 3 handles multiple paths, each one having its own SP list. Whenever a path is expanded, its SP is used as the starting point. This functionality is implemented by using the dictionary SP_{dict} that stores the SP arraylist for each path. $openDict$ is a another dictionary as well; the keys identify all the discovered paths, while the values concern the f values for a particular path.

A* scores all paths in $openDict$ with their f value. The g function makes use of Algorithm 1, in order to calculate the traversal time for each path, reaching the end of the path with the maximum possible speed, irrelevant to the next segments. In our example, the g values for paths (S, A, B, D, G) and (S, A, C) are 59.16 and 33.15 seconds respectively. The f function values for each path are computed in the usual way using an admissible heuristic function h that computes the time needed to travel the straight line distance between the last node of a path and the $destination$ node using $maxv(T)$.

The $successors$ dictionary contains the expansions of the current path with neighboring nodes. In our example this means that path (S, A, C) generates the paths (S, A, C, E) and (S, A, C, F). Path (S, A, C, F) is expanded to path (S, A, C, F, H) and path (S, A, C, E) is expanded to path (S, A, C, E, H) since there is no junction at nodes F and E. After the calculations of the f, g and h values, SP_{dict} is updated with the new paths and their SP arraylists, while $openDict$ and $successors$ are merged. Note that $\min(successors)$ denotes the key of $successors$ with the lowest f value. Moving on, the shortest path in $successors$ is selected, setting its final state as $lastState$. If $lastState = goal$, a solution is found, the algorithm terminates and returns the speed profile of the solution path as well as, the optimal $path$.

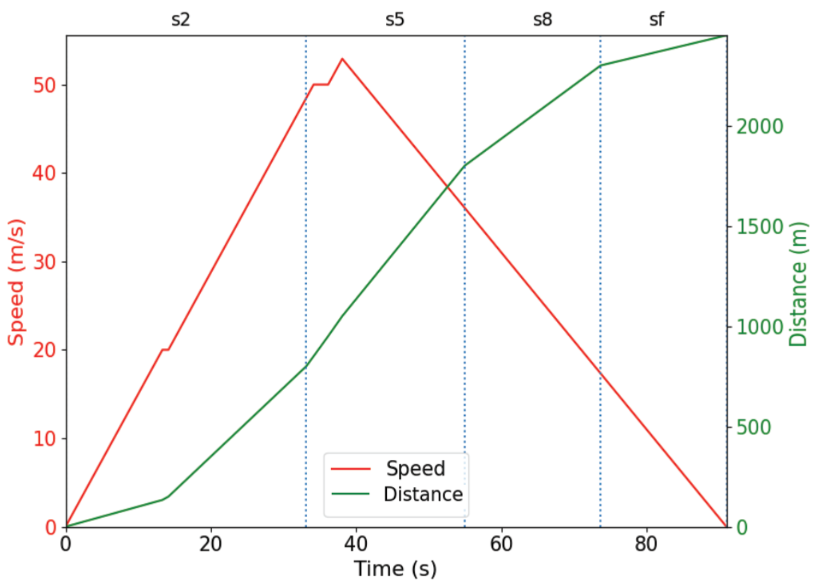


Figure 5. Speed and Position vs. time graph for the optimal path

Figure 5 presents the optimal speed profile for the shortest path (S, A, C, F, H) that has a duration of 91.02 seconds.

4.2.1 Use case: Finding the shortest path of Example 2

In this section, the operation of Algorithm 3 is showcased, when computing the shortest path in terms of time in a railway network graph, specifically that of Example 2. Figure 2 presents the railway network, consisting of 10 vertices and 12 edges. Each edge s is labeled with $l(s)$ and $vmax(s)$. The train T characteristics, as well as the goal state are included in Table 2, while $init = ((s_0\ , 0), 0)$. Algorithm 3 employs an iterative while-loop mechanism to examine and evaluate nodes of the network graph. For that reason, the computational analysis is presented for each iteration of the algorithm in order to simulate its operation.

Initialization. The $start$ path is defined by the segments (s_0,.., s_k) in $init$. For this example, s_k=s_0 with $s_0 = (S, A)$ hence, the $start$ path is (S, A). $openDict$ is initialized using the $start$ path: $openDict = \{(S, A)\text{: } 0\}$.

Moving on, the first path is generated by connecting the last node of $start$, that is A, with its neighboring nodes, which are B and C, using the $generatePaths()$ function. The paths generated by this function are stored in

$openDict = \{(S, A, B): 0, (S, A, C): 0\}$. Next, each generated path is extended by adding the neighbor of its final node until a junction node (leads to multiple nodes) is met. For the current example, this means that path (S, A, C) is not extended since C is already a junction node, while path (S, A, B) is extended into path (S, A, B, D, G). $openDict$ is updated with the extended paths: $openDict = \{(S, A, B, D, G): 0, (S, A, C): 0\}$. Figure 6 illustrates the first available paths detected by Algorithm 3, with their nodes marked with a blue color.

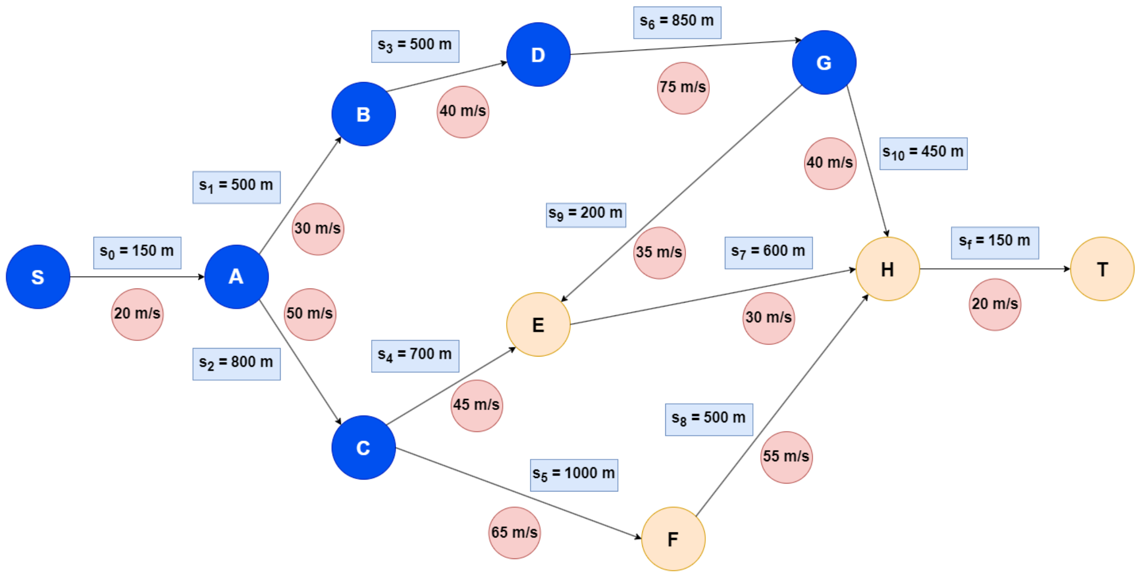


Figure 6. The first available paths of Example 2

At this point, for every path in $openDict$, the f values are calculated. Firstly, the g values for each path are computed using Algorithm 1, which calculates the traversal time, reaching the end of the path with $maxv(T)$ (75 m/s for Example 2), irrelevant to the next segments. These values are stored in $openDict = \{(S, A, B, D, G): 59.16, (S, A, C): 33.15\}$. Next, the f values are computed, by adding the heuristic function values of each path to their corresponding g values. Each node in the railway network G is given a straight-line distance estimation with respect to the $destination$ node T, the values of which are included in Table 28.

Table 28. Straight-line distance estimations for Example 2

Node	Straight-line distance
A	2000
B	1400
C	1200
D	900
E	900
F	500
G	500
H	50
T	0

The h and f values for each path are calculated as such:

$$h(S,A,B,D,G) = \frac{Straight\ line\ distance\ of\ node\ G}{maxv(T)} = \frac{500}{75} = 6.66$$

$$h(S,A,C) = \frac{Straight\ line\ distance\ of\ node\ C}{maxv(T)} = \frac{1200}{75} = 16$$

$$f(S,A,B,D,G) = g(S,A,B,D,G) + h(S,A,B,D,G) = 59.16 + 6.66 = 65.82$$

$$f(S,A,C) = g(S,A,C) + h(S,A,C) = 33.15 + 16 = 49.15$$

$openDict$ is updated with the f values, while the computed speed profiles for each path are stored in SP_{dict}. Table 29 includes the entries of SP_{dict} after the computation of the speed profiles.

Table 29. Speed profiles stored in SP_{dict} for each discovered path

Path	Speed Profile
(S,A,C) = (s_0, s_2)	(((s_0, s_2), 800), 0), (((s_0, s_2), 666.66), 20.0), (((s_2, 650.0), 20.0), (((s_2, 0), 48.47)

$(S, A, B, D, G) = (s_0, s_1, s_3, s_6)$	$(((s_0,\ s_1), 500), 0),$ $(((s_0,\ s_1), 366.66), 20.0),$ $((s_1,\ 350.0), 20.0),$ $((s_1,\ 183.33), 30.0),$ $(((s_1,\ s_3), 500), 30.0),$ $((s_3,\ 350.0), 30.0),$ $((s_3,\ 116.66), 40.0),$ $(((s_3,\ s_6), 850), 40.0),$ $((s_6,\ 700.0), 40.0),$ $((s_6,\ 0), 60.82)$

Iteration 1. The first step concerns selecting the $current$ path, that is the path with lowest f value in $openDict$ hence, $current = (S, A, C)$. The $current$ path is removed from $openDict$. Moving on, the $generatePaths()$ is applied on the $current$ path, connecting node C with its neighbors. In that way, paths (S, A, C, E) and (S, A, C, F) are created and then extended using the $extendPaths()$ function, leading to paths (S, A, C, E, H, T) and (S, A, C, F, H, T), that are stored in $successors$.

In the next step, the g values are calculated for each path in $successors$. Note that since the speed profile is already computed for path (S, A, C), Algorithm 1 is only applied exclusively on the extensions of the path, which are (C, E, H, T) and (C, F, H, T). This is possible because the speed profiles for each path are stored in SP_{dict}, hence speed profile computations for path extensions can continue from the final state of each path. However, instead of using the final state, the penultimate one is used in order to allow for Algorithm 1 to check whether a deceleration is needed before starting the computation of the speed profile of the path's extension. The reason behind is the fact that the speed limit of the extension's first segment is not known beforehand. Hence, it is quite possible that the train is not in a state (in terms of its current speed) to enter the new segment and a deceleration is needed in earlier points of the path's speed profile. For example, in the final state of the (S, A, C) speed profile, the train has a speed of 48.47 m/s. When the path is extended to (S, A, C, E, H, T), the speed limit of the esxtension's first segment is $vmax(s_4) = 45\frac{m}{s}$, that is lower than the train's current speed. For this reason, the speed profile computation begins from ((s_2, 650.0), 20.0), allowing the train to accelerate for 585 meters, leading to the new state (((s_2, 65), 46.42), and then immediately start decelerating, in order to reach s_4 with a speed of 45 m/s, creating the new state (((s_2, s_4), 700), 45.0).

As regards the g values for paths (S, A, C, E, H, T) and (S, A, C, F, H, T), they are 91.28 and 91.02 seconds, respectively. Since T is the destination node, the h values are 0 and the f values are equal to the g values, for both paths. Table 30 includes the f values for all discovered paths up until this point. $openDict$ and SP_{dict} are updated with the new f values and speed profiles, respectively. Path (S, A, C, F, H, T) has the lowest f value in $successors$, hence the final state of its speed profile $((s_f, 0), 0)$ is the new $lastState$. Due to the fact that $lastState = goal$, the A* algorithm terminates.

Table 30. f values of all discovered paths at the end of Iteration 1

Path	f value
(S, A, B, D, G)	65.82
(S, A, C, E, H, T)	91.28
(S, A, C, F, H, T)	91.02

5 Conclusion

At this section, the main points of the dissertation are summarized.

5.1 Summary and conclusions

Our work resulted in the development and the implementation of two algorithms for the tasks of computing the optimal speed profile over a given path, as well as, that of computing the shortest path in terms of journey duration, over arbitrary directed graphs. We provided a realistic problem formulation that makes use of train characteristics, including its length, maximum speed, acceleration and deceleration capabilities, while also including path characteristics, such as the speed limits that apply in the path segments.

5.2 Future extensions

Future challenges include the extension of our work in order to handle more physical characteristics of the problem, such as railway machine power profile, train load, ground slopes, optimizing other metrics, like fuel consumption. Offline and online scheduling of multiple trains, in order to satisfy particular transportation needs, is another significant, while optimizing the aforementioned metrics, is another significant future challenge. Finally, integrating our work with an information system with real data about railway networks, such as openrailwaymap.org, is needed to have a fully functional product.

6 References

[1] Nicolas J. Firzli, ‘Transportation infrastructure and country attractiveness’, Revue Analyse Financière, (Jul 2013).

[2] European-Comission, White paper on transport : Roadmap to a Single European transport area - Towards a competitive and resource-efficient transport system, Publications Office of the European Union, Luxembourg, 2011.

[3] AIcrowd. Train schedule optimisation challenge, 2018.

[4] AIcrowd. Flatland 3, 2021.

[5] Arjang A. Assad, ‘Models for rail transportation’, Transportation Research Part A: General, 14(3), 205–220, (Jun 1980).

[6] Jean-François Cordeau, Paolo Toth, and Daniele Vigo, ‘A survey of optimization models for train routing and scheduling’, Transportation Scence, 32(4), 380–404, (Nov 1998).

[7] Wei Fang, Shengxiang Yang, and Xin Yao, ‘A survey on problem models and solution approaches to rescheduling in railway networks’, IEEE Transactions on Intelligent Transportation Systems, 16(6), 2997–3016, (Dec 2015).

[8] Dennis Huisman, Leo G. Kroon, Ramon M. Lentink, and Michiel J. C. M. Vromans, ‘Operations research in passenger railway transportation’, Statistica Neerlandica, 59(4), 467–497, (Oct 2005).

[9] Richard M. Lusby, Jesper Larsen, Matthias Ehrgott, and David Ryan,‘Railway track allocation: models and methods’, OR Spectrum, 33(4),843–883, (Dec 2009).

[10] Johanna Törnquist. Computer-based decision support for railway traffic scheduling and dispatching: A review of models and algorithms, 2006.

[11] B Szpigel, ‘Optimal train scheduling on a single line railway’, Operations research, 72, 343–352, (1973).

[12] Andrea D'Ariano, Dario Pacciarelli, and Marco Pranzo, 'A branch and bound algorithm for scheduling trains in a railway network', European Journal of Operational Research, 183(2), 643–657, (Dec 2007).

[13] Young Hwan Yoon and Yung Bang. Apparatus and method for controlling speed in automatic train operation, Jul 2010.

[14] Zhou, Wenliang, and Hualiang Teng. "Simultaneous Passenger Train Routing and Timetabling Using an Efficient Train-Based Lagrangian Relaxation Decomposition." Transportation Research Part B: Methodological, vol. 94, Dec. 2016, pp. 409–439,, https://doi.org/10.1016/j.trb.2016.10.010.

[15] Louwerse, Ilse, and Dennis Huisman. "Adjusting a Railway Timetable in Case of Partial or Complete Blockades." European Journal of Operational Research, vol. 235, no. 3, 16 June 2014, pp. 583–593, www.sciencedirect.com/science/article/pii/S0377221713010047.

[16] Alessandro Mascis and Dario Pacciarelli, 'Job-shop scheduling with blocking and no-wait constraints', European Journal of Operational Research, 143(3), 498–517, (Dec 2002).

[17] G Nirmala and D Ramprasad, 'A genetic algorithm based railway scheduling model', International Journal of Science and Research (IJSR), (2014).

[18] P. Tormos, A. Lova, F. Barber, L. Ingolotti, M. Abril, and M. A. Salido, 'A genetic algorithm for railway scheduling problems', Metaheuristics for Scheduling in Industrial and Manufacturing Applications, 255–276, (2008).

[19] Wang, Mingming, et al. "Genetic Algorithm-Based Particle Swarm Optimization Approach to Reschedule High-Speed Railway Timetables: A Case Study in China." Journal of Advanced Transportation, vol. 2019, 20 Mar. 2019, pp. 1–12, https://doi.org/10.1155/2019/6090742..

[20] Marcella Samà, Paola Pellegrini, Andrea D'Ariano, Joaquin Rodriguez, and Dario Pacciarelli, 'Ant colony optimization for the real-time train routing selection problem', Transportation Research Part B: Methodological, 85, 89–108, (2016).

[21] Eaton, Jayne, et al. “Ant Colony Optimization for Simulated Dynamic Multi-Objective Railway Junction Rescheduling.” IEEE Transactions on Intelligent Transportation Systems, vol. 18, no. 11, 1 Nov. 2017, pp. 2980–2992, https://doi.org/10.1109/TITS.2017.2665042. .

[22] Wojciech Bo ˙zejko, Radosław Grymin, and Jarosław Pempera, ‘Scheduling and routing algorithms for rail freight transportation’, Procedia Engineering, 178, 206–212, (2017). RelStat-2016 9-22, 2016. Transport and Telecommunication Institute, Riga, Latvia.

[23] Francesco Corman, Andrea D’Ariano, Dario Pacciarelli, and Marco Pranzo, ‘A tabu search algorithm for rerouting trains during rail operations’, Transportation Research Part B: Methodological, 44(1), 175–192, (2010).

[24] Intaek Gong, Sukmun Oh, and Yunhong Min, ‘Train scheduling with deep q-network: A feasibility test’, Applied Sciences, 10(23), 8367, (Nov 2020).

[25] D. Šemrov, R. Marsetiˇc, M. Žura, L. Todorovski, and A. Srdic, ‘Reinforcement learning approach for train rescheduling on a single-track railway’, Transportation Research Part B: Methodological, 86, 250–267, (2016).

[26] Khadilkar, Harshad. “A Scalable Reinforcement Learning Algorithm for Scheduling Railway Lines.” IEEE Transactions on Intelligent Transportation Systems, vol. 20, no. 2, 1 Feb. 2019, pp. 727–736, ieeexplore.ieee.org/abstract/document/8357909/, https://doi.org/10.1109/TITS.2018.2829165.

[27] Valerio Agasucci, Giorgio Grani, and Leonardo Lamorgese. Solving the single-track train scheduling problem via deep reinforcement learning, 2020.

[28] Selim Dündar and ˙Ismail ¸Sahin, ‘Train re-scheduling with genetic algorithms and artificial neural networks for single-track railways’, Transportation Research Part C: Emerging Technologies, 27, 1–15, (2013).

Annex A – Executing the source code

This section provides information regarding the execution of the application that implements the source code, as well as the formatting of the input data. The code was implemented using Python 3.8.10. The source code and the input files of various problems can be found at GitHub[2]. The code is organized into two separate files:

- File *OptimalPathSpeedProfile.py* can be used to compute the speed profile of any provided path. It includes the implementations of Algorithm 1 and Algorithm 2, as well as the necessary functions needed for the algorithms' operation and for parsing the input data.
- File *ShortestPath.py* extends the functionality of the first file, since it import its code, in order to apply Algorithm 1 and Algorithm 2 on the paths of a railway network graph through Algorithm 3.

A.1. Computing the optimal speed profile of a path

The *OptimalPathSpeedProfile.py* script utilizes three Python modules to perform its functions, which are presented in Table 31. These modules are included in the Python Standard Library and it should not be necessary to install them.

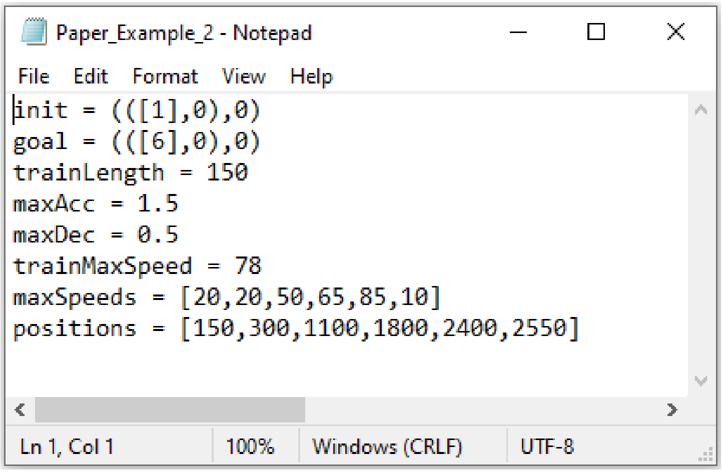
Table 31. Libraries used in the *OptimalPathSpeedProfile.py* script

Library	Description
math	A Python build-in module that is used for mathematical computations.

[2] Link for the GitHub repository: https://github.com/Dimimano/Finding-Time-Optimal-Routes-for-Trains-Using-Basic-Kinematics-and-A-

argparse	A Python build-in module, essential for the script's command-line interface (CLI), enabling it to accept and process input parameters from users.
ast	A Python build-in module that is used in order for the program to parse input data from text files into Python data structures.

The input data necessary for the execution of the script concern the path and train characteristics, as well as the *init* and the *goal* states, which are provided to the application through a text file. Figure 7 illustrates the appropriate format of the input data in the text file of Example 2. The list of segments in which the train spans in the *init* and *goal* states must be provided a list of integers. In Example 2, the train in *init* is at the end of s_0, that is the first segment of the path, while in *goal*, the train is at the end of s_f, that is the sixth segment of the path. Hence, the lists of integers are [1] and [6] for *init* and *goal*, respectively.



```
init = (([1],0),0)
goal = (([6],0),0)
trainLength = 150
maxAcc = 1.5
maxDec = 0.5
trainMaxSpeed = 78
maxSpeeds = [20,20,50,65,85,10]
positions = [150,300,1100,1800,2400,2550]
```

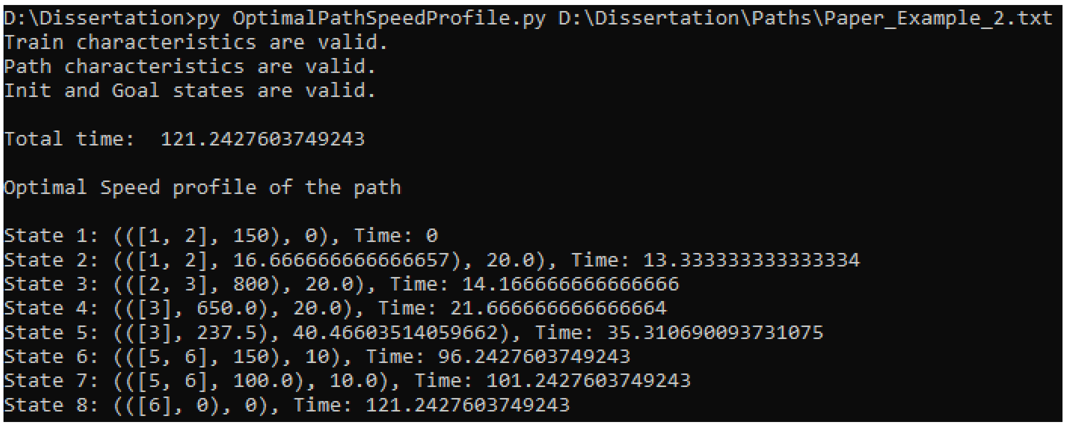
Figure 7. Input data format in a text file for Example 2

All input data are validated before the script's execution. For instance, all train characteristics must be non-negative float numbers. Similarly, all entries of the *maxSpeeds* and *positions* lists must be positive float numbers. Additionally, the train's length, as well as the acceleration and deceleration capabilities are limited from the global variables TRAIN_LENGTH_LIMIT, MAX_ACC_LIMIT and MIN_ACC_LIMIT, which are defined in the first lines of the script. Table 32 includes the values of these parameters for all the examples presented in this study.

Table 32. Default values for the global variables

Global Variable	Value
TRAIN_LENGTH_LIMIT	4000
MAX_ACC_LIMIT	5
MIN_ACC_LIMIT	5

Moving on, the code can be executed using the command-line interface and more specifically, using the command: py OptimalPathSpeedProfile.py <path_to_the_file.txt>. Figure 8 showcases the script's execution for Example 2. The script provides information regarding the validation of the input data and then, it computes the optimal speed profile, printing the total time it takes for the train to traverse it, as well as the states of the speed profile.

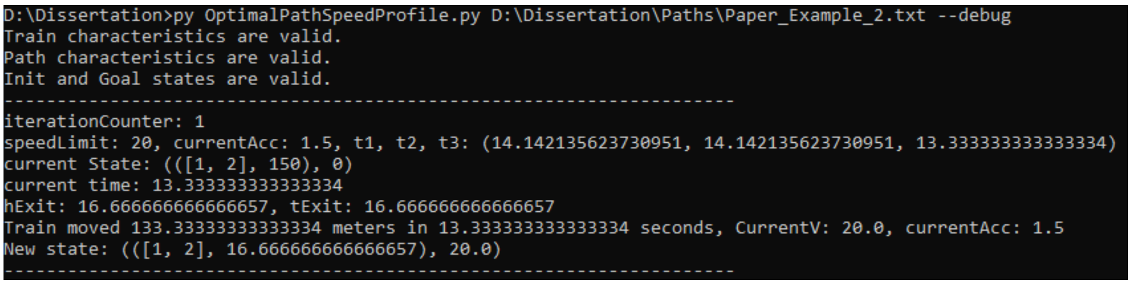


```
D:\Dissertation>py OptimalPathSpeedProfile.py D:\Dissertation\Paths\Paper_Example_2.txt
Train characteristics are valid.
Path characteristics are valid.
Init and Goal states are valid.

Total time:  121.2427603749243

Optimal Speed profile of the path

State 1: (([1, 2], 150), 0), Time: 0
State 2: (([1, 2], 16.666666666666657), 20.0), Time: 13.333333333333334
State 3: (([2, 3], 800), 20.0), Time: 14.166666666666666
State 4: (([3], 650.0), 20.0), Time: 21.666666666666664
State 5: (([3], 237.5), 40.46603514059662), Time: 35.310690093731075
State 6: (([5, 6], 150), 10), Time: 96.2427603749243
State 7: (([5, 6], 100.0), 10.0), Time: 101.2427603749243
State 8: (([6], 0), 0), Time: 121.2427603749243
```

Figure 8. Execution of the *OptimalPathSpeedProfile.py* script in the CLI

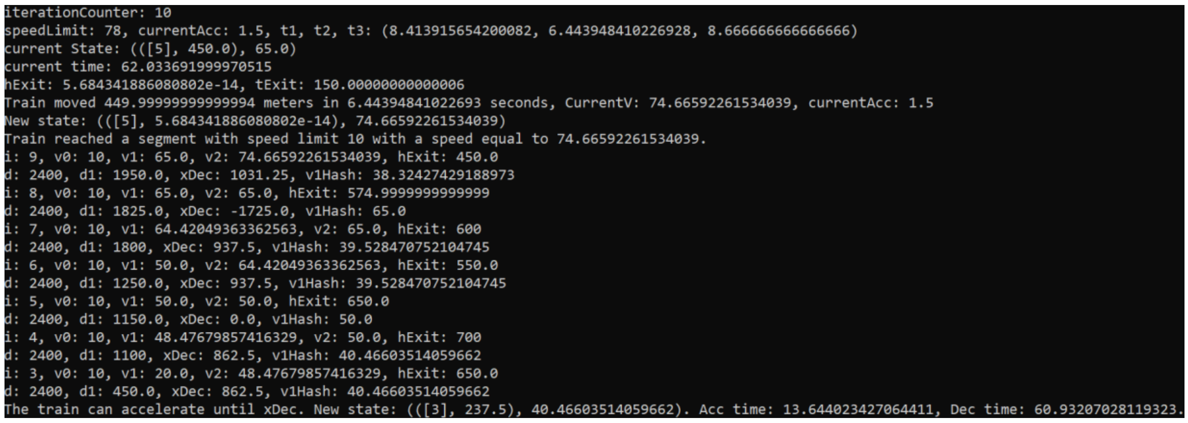
In case the user prefers a more detailed output regarding the program's execution, the –debug parameter can be added to the execution command. Figure 9 illustrates the output of the program for the first iteration of Algorithm 1, while

Figure 10 shows the program's output when Algorithm 2 is applied, if the debug parameter is used.



```
D:\Dissertation>py OptimalPathSpeedProfile.py D:\Dissertation\Paths\Paper_Example_2.txt --debug
Train characteristics are valid.
Path characteristics are valid.
Init and Goal states are valid.
--------------------------------------------------------------------
iterationCounter: 1
speedLimit: 20, currentAcc: 1.5, t1, t2, t3: (14.142135623730951, 14.142135623730951, 13.333333333333334)
current State: (([1, 2], 150), 0)
current time: 13.333333333333334
hExit: 16.666666666666657, tExit: 16.666666666666657
Train moved 133.33333333333334 meters in 13.333333333333334 seconds, CurrentV: 20.0, currentAcc: 1.5
New state: (([1, 2], 16.666666666666657), 20.0)
--------------------------------------------------------------------
```

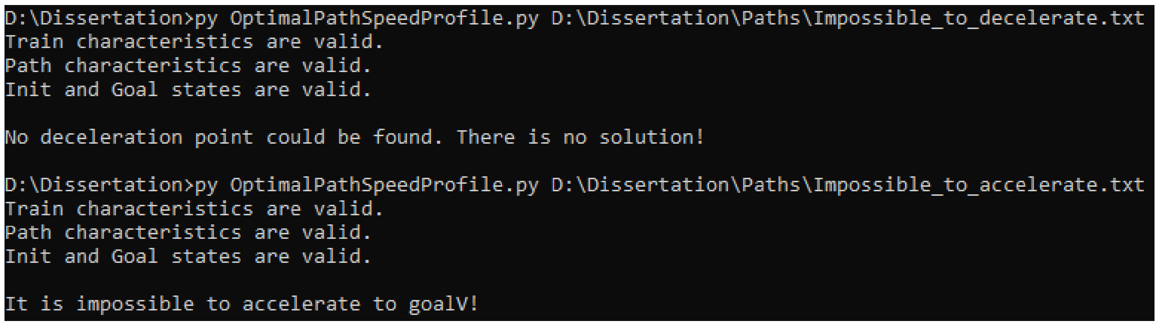
Figure 9. Output of Algorithm 1 when the debug parameter is used



```
iterationCounter: 10
speedLimit: 78, currentAcc: 1.5, t1, t2, t3: (8.413915654200082, 6.443948410226928, 8.666666666666666)
current State: (([5], 450.0), 65.0)
current time: 62.033691999970515
hExit: 5.684341886080802e-14, tExit: 150.00000000000006
Train moved 449.99999999999994 meters in 6.44394841022693 seconds, CurrentV: 74.66592261534039, currentAcc: 1.5
New state: (([5], 5.684341886080802e-14), 74.66592261534039)
Train reached a segment with speed limit 10 with a speed equal to 74.66592261534039.
i: 9, v0: 10, v1: 65.0, v2: 74.66592261534039, hExit: 450.0
d: 2400, d1: 1950.0, xDec: 1031.25, v1Hash: 38.32427429188973
i: 8, v0: 10, v1: 65.0, v2: 65.0, hExit: 574.9999999999999
d: 2400, d1: 1825.0, xDec: -1725.0, v1Hash: 65.0
i: 7, v0: 10, v1: 64.42049363362563, v2: 65.0, hExit: 600
d: 2400, d1: 1800, xDec: 937.5, v1Hash: 39.528470752104745
i: 6, v0: 10, v1: 50.0, v2: 64.42049363362563, hExit: 550.0
d: 2400, d1: 1250.0, xDec: 937.5, v1Hash: 39.528470752104745
i: 5, v0: 10, v1: 50.0, v2: 50.0, hExit: 650.0
d: 2400, d1: 1150.0, xDec: 0.0, v1Hash: 50.0
i: 4, v0: 10, v1: 48.47679857416329, v2: 50.0, hExit: 700
d: 2400, d1: 1100, xDec: 862.5, v1Hash: 40.46603514059662
i: 3, v0: 10, v1: 20.0, v2: 48.47679857416329, hExit: 650.0
d: 2400, d1: 450.0, xDec: 862.5, v1Hash: 40.46603514059662
The train can accelerate until xDec. New state: (([3], 237.5), 40.46603514059662). Acc time: 13.644023427064411, Dec time: 60.93207028119323.
```

Figure 10. Output of Algorithm 2 when the debug parameter is used

Finally, as mentioned in section 4.1 it is possible that no solutions are found, in cases where the train cannot decelerate early enough until the speed limit of a segment it is about to enter (or $goalV$) or the train cannot accelerate enough, in order to reach $goalV$. Figure 11 shows the script's execution and output on two example paths, in which a solution cannot be found.



```
D:\Dissertation>py OptimalPathSpeedProfile.py D:\Dissertation\Paths\Impossible_to_decelerate.txt
Train characteristics are valid.
Path characteristics are valid.
Init and Goal states are valid.

No deceleration point could be found. There is no solution!

D:\Dissertation>py OptimalPathSpeedProfile.py D:\Dissertation\Paths\Impossible_to_accelerate.txt
Train characteristics are valid.
Path characteristics are valid.
Init and Goal states are valid.

It is impossible to accelerate to goalV!
```

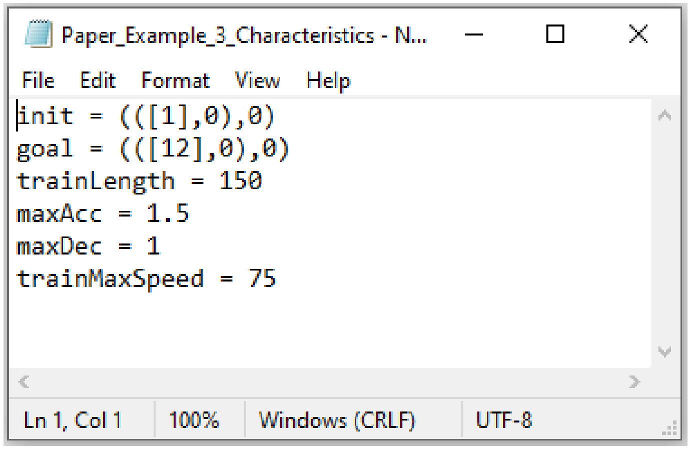
Figure 11. Output of the script when a solution cannot be found

A.2. Finding the shortest path of a railway network

The *ShortestPath.py* script utilizes all the Python modules presented in the previous section, as well as another two Python modules, which are presented in Table 33. Module *copy* is included in the Python Standard Library and it is not necessary to install, while module *networkx* and *numpy* must be installed. Note that in order for this script to run, it must be in the same folder as the *OptimalPathSpeedProfile.py* script.

Table 33. Libraries used in the *ShortestPath*.py script

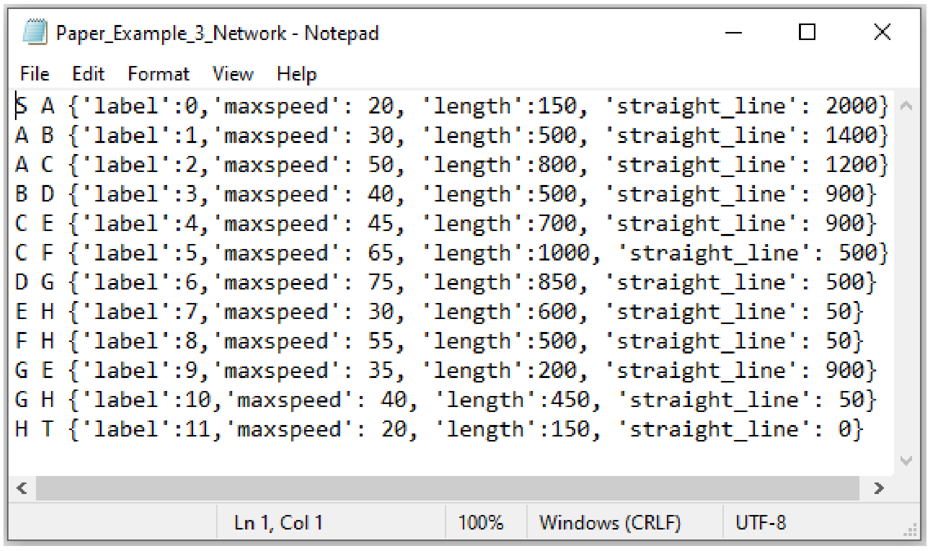
Library	Description
copy	A Python build-in module that is used in order to create (deep) copies of Python data structures.
networkx	A Python module, essential for creating and processing network graphs.
numpy	A Python library that is primarily used for working with arrays.


Paper_Example_3_Characteristics - N...
File Edit Format View Help
Ln 1, Col 1 100% Windows (CRLF) UTF-8

```
init = (([1],0),0)
goal = (([12],0),0)
trainLength = 150
maxAcc = 1.5
maxDec = 1
trainMaxSpeed = 75
```

Figure 12. Format of the states and characteristics in a text file for Example 3

In the case of *ShortestPath.py*, the input data concern the train characteristics, the *init* and the *goal* states (as before), as well as, the railway network graph G edgelist. Two separate text files are used; the first one includes the train characteristics, the *init* and the *goal* states (as in Figure 12 for Example 3), while the second one contains the network data, as illustrated in Figure 13.



```
S A {'label':0,'maxspeed': 20, 'length':150, 'straight_line': 2000}
A B {'label':1,'maxspeed': 30, 'length':500, 'straight_line': 1400}
A C {'label':2,'maxspeed': 50, 'length':800, 'straight_line': 1200}
B D {'label':3,'maxspeed': 40, 'length':500, 'straight_line': 900}
C E {'label':4,'maxspeed': 45, 'length':700, 'straight_line': 900}
C F {'label':5,'maxspeed': 65, 'length':1000, 'straight_line': 500}
D G {'label':6,'maxspeed': 75, 'length':850, 'straight_line': 500}
E H {'label':7,'maxspeed': 30, 'length':600, 'straight_line': 50}
F H {'label':8,'maxspeed': 55, 'length':500, 'straight_line': 50}
G E {'label':9,'maxspeed': 35, 'length':200, 'straight_line': 900}
G H {'label':10,'maxspeed': 40, 'length':450, 'straight_line': 50}
H T {'label':11,'maxspeed': 20, 'length':150, 'straight_line': 0}
```

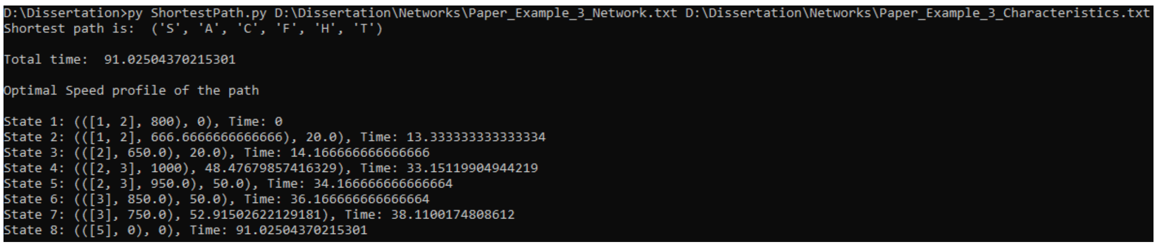
Figure 13. Format of the network data in a text file for Example 3

Each line of the network data concern a specific edge of the network, with the first two elements defining the nodes. Additionally, each edge has four attributes; label concerns the index of each edge (for instance, segment s_0 is given the label 0), while maxspeed and length define the segment characteristics. The attribute straight_line characterizes the ending node of each segment and is used for the computation of *h* values for paths, having that node as the final one.

As mentioned previously, the list of segments in which the train spans in the *init* and *goal* states must be provided as list of integers. The labels of each segment must be used, in order to index the starting and the ending segments. Note that in Example 3, the labels are numbered from 0 to 11, while the indices from 1 to 12. Finally, it is important to note that the train can start from any point of the network, but at the *goal* state, it must span inside of one segment only, that is the destination segment.

Moving on, the code can be executed using the command-line interface and more specifically, using the command: py ShortestPath.py <path_to_the_network_data_file.txt> <path_to_the_characteristics_file.txt>.

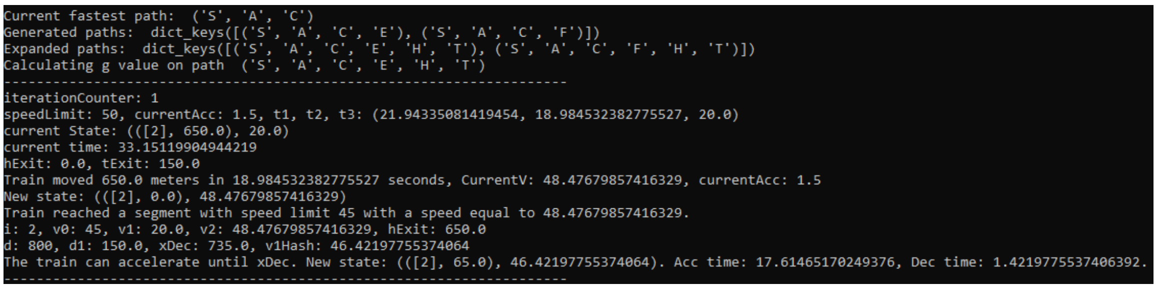
Figure 14 showcases the script's execution for Example 3. The scripts prints the nodes of the shortest path, the total time the train needs to traverse it, as well as, the states of the speed profile for the shortest path.



```
D:\Dissertation>py ShortestPath.py D:\Dissertation\Networks\Paper_Example_3_Network.txt D:\Dissertation\Networks\Paper_Example_3_Characteristics.txt
Shortest path is:  ('S', 'A', 'C', 'F', 'H', 'T')

Total time:  91.02504370215301

Optimal Speed profile of the path

State 1: (([1, 2], 800), 0), Time: 0
State 2: (([1, 2], 666.6666666666666), 20.0), Time: 13.333333333333334
State 3: (([2], 650.0), 20.0), Time: 14.166666666666666
State 4: (([2, 3], 1000), 48.47679857416329), Time: 33.15119904944219
State 5: (([2, 3], 950.0), 50.0), Time: 34.166666666666664
State 6: (([3], 850.0), 50.0), Time: 36.166666666666664
State 7: (([3], 750.0), 52.91502622129181), Time: 38.1100174808612
State 8: (([5], 0), 0), Time: 91.02504370215301
```

Figure 14. Execution of the *ShortestPath.py* script in the CLI

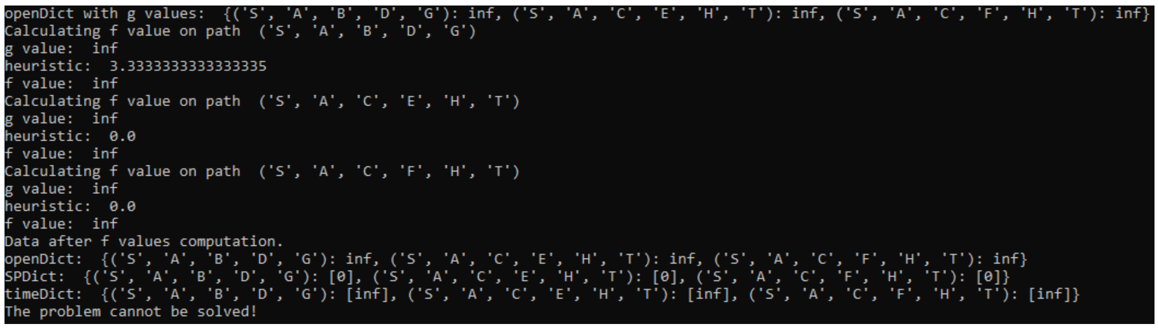
In case the user prefers a more detailed output regarding the program's execution, the –debug parameter can be added to the execution command, which will enable all algorithms to print information as they are executed. Figure 15 illustrates the output of the program for the first iteration of Algorithm 3, while it is calculating the g value of path (S, A, C, E, H, T).



```
Current fastest path:  ('S', 'A', 'C')
Generated paths:  dict_keys([('S', 'A', 'C', 'E'), ('S', 'A', 'C', 'F')])
Expanded paths:  dict_keys([('S', 'A', 'C', 'E', 'H', 'T'), ('S', 'A', 'C', 'F', 'H', 'T')])
Calculating g value on path  ('S', 'A', 'C', 'E', 'H', 'T')
-----------------------------------------------------------------
iterationCounter: 1
speedLimit: 50, currentAcc: 1.5, t1, t2, t3: (21.94335081419454, 18.984532382775527, 20.0)
current State: (([2], 650.0), 20.0)
current time: 33.15119904944219
hExit: 0.0, tExit: 150.0
Train moved 650.0 meters in 18.984532382775527 seconds, CurrentV: 48.47679857416329, currentAcc: 1.5
New state: (([2], 0.0), 48.47679857416329)
Train reached a segment with speed limit 45 with a speed equal to 48.47679857416329.
i: 2, v0: 45, v1: 20.0, v2: 48.47679857416329, hExit: 650.0
d: 800, d1: 150.0, xDec: 735.0, v1Hash: 46.42197755374064
The train can accelerate until xDec. New state: (([2], 65.0), 46.42197755374064). Acc time: 17.61465170249376, Dec time: 1.4219775537406392.
-----------------------------------------------------------------
```

Figure 15. Output of Algorithm 3 when the debug parameter is used

Moving on, in cases where Algorithm 1 cannot find a solution for a particular path, the path is converted to a 'dead end' by setting its g value to $infinite$. In that way the path will not be expanded upon in future iterations of Algorithm 3. Additionally, it is possible that all the available paths of network have no solution. Figure 16 showcases the output of the script is such cases (files *No_solution_Network.txt* and *No_solution_Characteristics*). Note that the g values of all discovered paths in this example are $infinite$.



```
openDict with g values:  {('S', 'A', 'B', 'D', 'G'): inf, ('S', 'A', 'C', 'E', 'H', 'T'): inf, ('S', 'A', 'C', 'F', 'H', 'T'): inf}
Calculating f value on path  ('S', 'A', 'B', 'D', 'G')
g value:  inf
heuristic:  3.3333333333333335
f value:  inf
Calculating f value on path  ('S', 'A', 'C', 'E', 'H', 'T')
g value:  inf
heuristic:  0.0
f value:  inf
Calculating f value on path  ('S', 'A', 'C', 'F', 'H', 'T')
g value:  inf
heuristic:  0.0
f value:  inf
Data after f values computation.
openDict:  {('S', 'A', 'B', 'D', 'G'): inf, ('S', 'A', 'C', 'E', 'H', 'T'): inf, ('S', 'A', 'C', 'F', 'H', 'T'): inf}
SPDict:  {('S', 'A', 'B', 'D', 'G'): [0], ('S', 'A', 'C', 'E', 'H', 'T'): [0], ('S', 'A', 'C', 'F', 'H', 'T'): [0]}
timeDict:  {('S', 'A', 'B', 'D', 'G'): [inf], ('S', 'A', 'C', 'E', 'H', 'T'): [inf], ('S', 'A', 'C', 'F', 'H', 'T'): [inf]}
The problem cannot be solved!
```

Figure 16. Output of the script when no solution can be found in a network

Annex B – Source code of the implementation

In this section, a significant portion of the source code is presented, particularly focusing on the algorithms and functions discussed in previous sections. As mentioned previously, all of the code can be found at dissertation's GitHub repository[3].

B.1. Computing the optimal speed profile of a path

Implementation of Algorithm 1:

```
def computeOptimalSP(goal,pathChar,trainChar,variables,debug,intermediate):
     '''ALGORITHM1: Computation of the optimal speed profile between an initial state init and a
goal state goal, assuming that there is a single path that can be used to reach the goal state,
with no junctions across it. Iteratively detects events during the train's journey.'''

    #Extract train characteristics.
    trainLength, Acc, Dec, trainMaxSpeed = trainChar

    #Extract path characteristics.
    maxSpeeds, positions, lengths, segmentIndices = pathChar

    #Extract variables.
  currentV,currentAcc,time,tSgm,hSgm,hExit,tExit,SP,flag,init,speedLimit,tMin,t1,t2,t3 = variables
    goalV = speed(goal)

    iterationCounter = 0

    while(not equalStates(SP[-1],goal)): #while current state != goal

        iterationCounter += 1

        p = currentV*tMin + (1/2)*currentAcc*math.pow(tMin,2) #Update position.
        currentV = currentV + tMin*currentAcc #Update speed.
        tExit, hExit = tExit - p, hExit - p #Update tExit, hExit.
        time.append(time[-1] + tMin) #Update total time.
```

[3] Link for the GitHub repository: https://github.com/Dimimano/Finding-Time-Optimal-Routes-for-Trains-Using-Basic-Kinematics-and-A-

```
        #The train's head is about to enter a new segment.
        #Deceleration might be needed.
        if equals(t2, tMin):

            hSgm += 1 #The train's head is in the new segment.

            try:
                #Set hExit equal to the length of the new segment.
                hExit = lengths[hSgm-1]
                #Calculate the new speed limit.
                speedLimit = getSpeedLimit(trainMaxSpeed, maxSpeeds, tSgm, hSgm)

            #This covers the case when the train is about to finish the last segment.
            except:
                t2Flag, t1Flag = True, False
                hExit, hSgm, t2Flag, t1Flag, speedLimit =
                 finalSegment(pathChar, trainChar, tSgm, hSgm, t2Flag, t1Flag, speedLimit)

            if currentV > speedLimit: #The train will need to decelerate.

               #Store the achieved state before backtracking.
               SP.append((([indexforindexinrange(tSgm,hSgm+1)],hExit),currentV))

               #Apply the algorithm that handles deceleration.
                decFlag, SP, time, currentV, currentAcc =
                decelerate(trainChar, pathChar, SP, time, currentV, currentAcc, speedLimit, debug)

                if not decFlag:
                    print('\nNo deceleration point could be found. There is no solution!')
                    return None, None

        #The train's tail is about to exit a previous segment. Acceleration might be needed.
        if equals(t1, tMin):

            tSgm += 1 #The train's tail exited the previous segment.

            try:
                #Set hExit equal to the length of the next segment.
                tExit = lengths[tSgm-1]
                #Calculate the new speed limit.
                speedLimit = getSpeedLimit(trainMaxSpeed, maxSpeeds, tSgm, hSgm)
            #This covers the case when the train is about to finish the last segment.
            except:
                t2Flag, t1Flag = False, True
                tExit, tSgm, t2Flag, t1Flag =
                finalSegment(pathChar, trainChar, tSgm, hSgm, t2Flag, t1Flag, speedLimit)
            if currentV < speedLimit: currentAcc = Acc #Acceleration
        #The train reached the speed limit. A steady speed must be maintained.
        if equals(t3, tMin) and currentV == speedLimit: currentAcc = 0

          #Update SP
          SP.append((([index for index in range(tSgm,hSgm+1)],hExit),currentV))

        #At this point it is checked whether the train reached the goal state.
        #Note that in the case of a network, the algorithm might work on smaller/intermediate paths
        #that do not include the goal state. In that case, the train does not need to have a speed
        #equal to goalV when reaching the end of the path.
        if not intermediate:
        #If the train reaches the goal position with a lower speed than goalV,
        #then there is no solution.
          if pos(SP[-1]) == pos(goal)
          AND abs(distance(SP[-1],positions)-distance(goal,positions))<=MIN_DIFFERENCE
          AND currentV != goalV:
                print('\nIt is impossible to accelerate to goalV!')
                return None, None
        #If the path is an intermediate one, the train does not need to have a speed equal to goalV.
        else:
          if pos(SP[-1]) == pos(goal)
          AND abs(distance(SP[-1],positions)-distance(goal,positions))<=MIN_DIFFERENCE:
                break

        if speedLimit == currentV: currentAcc = 0 #Update train's mode.
        #Calculate tMin.
        if currentV != 0: tMin, t1, t2, t3 = getMinT(currentAcc, currentV, tExit, hExit, speedLimit)

    return SP, time
```

Implementation of Algorithm 2:

```
def decelerate(trainChar, pathChar, SP, time, currentV, currentAcc, speedLimit, debug):
    '''ALGORITHM2: Backtracks in previous states stored in SP, searching for the optimal point to
decelerate. The most complicated case occurs when head(T) is about to enter a new segment (t2 =
tmin) with currentV higher than the new segment's speed limit. In this case the speed profile
must be recomputed by adding a deceleration point at the latest possible time, when the train
should start decelerating in order to reach the new segment with a speed equal to speedLimit.'''

    #Extract train characteristics.
    trainLength, Acc, Dec, trainMaxSpeed = trainChar

    #Extract path characteristics.
    maxSpeeds, positions, lengths, segmentIndices = pathChar

    #If ALGORITHM2 can not find a deceleration point, the problem has no solution.
    decFlag = False
```

```
#Total distance at the start of the new segment (last state).
d = distance(SP[-1],positions)

#Iteratively backtrack to previous states (starting from the second to last).
for i in range(len(SP)-2,-1,-1):

    #The algorithm searches for previous parts of the speed profile during which the train
    #was either accelerating or maintaining its speed, starting from the most recent on.

    #v0 is the max speed of the new segment (speed limit).
    #v1 is the speed of the train at the start of the SP part.
    #v2 is the speed of the train at the end of the SP part.
    v0, v1, v2 = speedLimit, speed(SP[i]), speed(SP[i+1])

    #Extract hExit for this part.
    hExit = pos(SP[i])[1]

    #We are interested only in SP parts with a mode of acceleration or steady speed.
    if v2 >= v1:
        #Determine the train's mode during the SP part.
        if v2==v1: currentAcc = 0
        else: currentAcc = Acc

        #Total distance at the start of the SP part.
        d1 = distance(SP[i],positions)
        #Calculate the optimal deceleration point between d1 and d.
        xDec =
        (math.pow(v0,2) - math.pow(v1,2) + 2*(currentAcc*d1 + Dec*d))/(2*(currentAcc + Dec))

        #Calculate the speed at that point.
        v1Hash = math.sqrt(math.pow(v1,2) + 2*currentAcc*(xDec - d1))

        if v1Hash > v1: #The train can accelerate until xDec.

            #Add two states. One before and one after the deceleration.
            i, SP, time, v1Hash, trainChar, pathChar, xDec, hExit, v0, Dec, currentV,
            currentAcc, decFlag, t1Hash, t2Hash =
            addTwoStates(i, SP, time, v1Hash, currentAcc, trainChar, pathChar, xDec, hExit,
            v0, v1, Dec, d1)

            return decFlag, SP, time, currentV, currentAcc

        #The train must have steady speed until xDec (or decelerate immediately).
        if v1Hash == v1:

            if xDec > d1: #The train must have steady speed until xDec.
```

```
                    #Add two states. One before and one after the deceleration.
                    i, SP, time, v1Hash, trainChar, pathChar, xDec, hExit, v0, Dec, currentV,
                    currentAcc, decFlag, t1Hash, t2Hash =
                    addTwoStates(i, SP, time, v1Hash, currentAcc, trainChar, pathChar,xDec,hExit,
                    v0, v1, Dec, d1)

                    return decFlag, SP, time, currentV, currentAcc

                elif xDec == d1: #The train must decelerate immediately.

                    #Add one state. One after the deceleration.
                    i, SP, time, v1Hash, v0, Dec, currentV, currentAcc, decFlag, tHash =
                    addOneState(i, SP, time, v1Hash, v0, Dec)

                    return decFlag, SP, time, currentV, currentAcc

    return decFlag, SP, time, currentV, currentAcc
```

Implementation of Function 1:

```
def getSpeedLimit(trainMaxSpeed, maxSpeeds, tSgm, hSgm):
    '''Takes as input the train characteristics and the segments it is currently in, in order to
calculate the speed limit.'''

    lowestVmax = min(maxSpeeds[tSgm-1:hSgm])

    return min(trainMaxSpeed,lowestVmax)
```

Implementation of Function 2:

```
def getMinT(currentAcc, currentV, tExit, hExit, speedLimit):
    '''Detects which type of event is upcoming, by computing the time tMin = min(t1, t2, t3)
needed to reach each one of them.'''

    #Time needed for the tail to exit its segment.
    if currentAcc>0: t1 = MaxRoot((1/2)*currentAcc, currentV, -tExit)
    else: t1 = tExit/currentV
    #Time needed for the head to enter a new segment.
    if currentAcc>0: t2 = MaxRoot((1/2)*currentAcc, currentV, -hExit)
    else: t2 = hExit/currentV
    #Time needed for the train to reach the speedLimit, provided that it is in accelerating mode.
    if currentAcc > 0 and speedLimit > currentV: t3 = (speedLimit - currentV)/currentAcc
    else: t3 = float("inf")

    return min(t1,t2,t3), t1, t2, t3
```

Implementation of Function 3:

```
def MaxRoot(a,b,c):
    '''Returns the positive root.'''

    Delta = math.pow(b,2) - 4*a*c #discriminant
    t = (-b + math.sqrt(Delta))/(2*a) #positive root

    return t
```

B.2. Finding the shortest path of a railway network

Implementation of Algorithm 3:

```
def computeShortestPath(G, variables, debug):
    '''ALGORITHM3: Using the problem's description provided by the createInputData function, as
well as the network G, this algorithm provides the shortest path (time-wise) and its speed
profile.'''

    #Initialization
    SPDict, timeDict = {}, {}
    mapDict = createMap(G)

    #Extract init and goal states, as well as the trainChar (trainLength etc.).
    _, initPos, init, goalV, goalPos, _, trainChar, _ = storeVariables(variables)

        #Extract train max speed.
    _, _, _, trainMaxSpeed = trainChar

    #Set the hyperparameter maxStraightLineSpeed.
    maxStraightLineSpeed = trainMaxSpeed

    #Initiliaze openDict and start path, find the final node.
    openDict, destinationNode, start = extractStartDestination(G, initPos, goalPos)

    #Generate the first paths using the start path.
    neighbors = generatePaths(G, start)

    #Update openDict using the generated paths in neighbors.
    openDict = updateOpenDict(neighbors, openDict, start)

    #Expand the paths of openDict.
    openDict, finalPath = expandPaths(G, destinationNode, openDict)

    #Compute g values for all paths in openDict.
```

```
openDict, SPDict, timeDict = computeGValues(openDict, init, initPos, trainChar, SPDict,
timeDict, destinationNode, goalV, goalPos, finalPath, False, debug)

#Compute f values for all paths in openDict.
openDict_fValues = computeFValues(openDict, mapDict, maxStraightLineSpeed, debug)

while(openDict): #while openDict not empty

    #Find the path with lowest f value.
    current = min(openDict_fValues, key=openDict_fValues.get)

    oldPath = current #Store it before expanding it.

    #Generate new paths using the neighbors of its final nodes.
    current = generatePaths(G, current)

    #Create the successors dictionary.
    successors = createSuccessors(current, openDict, oldPath)

    #Expand the paths in successors.
    successors, finalPath = expandPaths(G, destinationNode, successors)

    #Compute g values for all paths in successors.
    successors, SPDict, timeDict = computeGValues(successors, init, initPos, trainChar,
    SPDict, timeDict, destinationNode, goalV, goalPos, finalPath, True, debug, oldPath)

    #Update openDict, SPDict and timeDict using successors.
    openDict = updateOpenDict(successors, openDict, oldPath, timeDict)
    SPDict, timeDict = updateSPandTimeDict(SPDict, timeDict, oldPath)

    #Compute f values for all paths in openDict.
    openDict_fValues = computeFValues(openDict, mapDict, maxStraightLineSpeed, debug)

    #If a path containing the destination node is found, return the solution.
    if finalPath:
        current = min(successors, key=successors.get)
        return openDict, SPDict, current, timeDict
```

Implementation of Function 4:

```python
def generatePaths(G, path):
    '''generatePaths() iterates through the outgoing edges (of the last vertex of the path),
generating new paths (which are extensions of the current path, with a single or more segments
added to it) that are stored temporarily in neighbors, before entering openDict.

    :param G: A networkx Graph in which the problem is defined on.
    :param path: List, contains the nodes of the path.
    :return: neighbors: List, containts the nodes of each path that was generated through the
outgoing edges.'''

    neighbors = []
    node = path[-1] #last vertex of the path

    for u,v,e in G.edges(data=True):
        #Iterate through the outgoing edges and generate paths
        if u == node: neighbors.append(path + (v,))

    return neighbors
```

Implementation of Function 5:

```python
def expandPaths(G, destinationNode, dict):
    '''After a path is generated (its last node is connected with each one of its N neighbors
creating N paths), it is expanded by adding the neighbor of its final node until a junction node
(leads to multiple nodes) is met.

    :param G: A networkx Graph in which the problem is defined on.
    :param destinationNode: Char, the name of the final node (at the end of final segment).
    :param dict: Dictionary (openDict or successors) (key -> list of characters/nodes, value ->
float), contains the path of openDict which will be expanded.
    :return: dict: Dictionary (key -> list of characters/nodes, value -> float), contains the
expanded versions of paths in dict.
    :return: finalPath: Boolean, is True when one of the paths in paths is final (contains
destinationNode).
    '''

    paths = list(dict.keys()) #Extract the list of paths from dict.

    finalPath = False #Initialize

    for path in paths:

        oldPath = path #Store the path before expansion, in order to update openDict.

        while(1): #Expand paths until a junction node is found
        junctions = 0
        #Iterate through the network, searching for nodes (v) to expand the path (u)
        for u,v,e in G.edges(data=True):
           #If u and v are connected, u is not the final node and v is not already in the
           #path (cycle)...
            if path[-1]==u and path[-1]!=destinationNode and v not in path:
              nextNode = v #Store v as the nextNode.
              junctions += 1 #Increment the junction counter.
        #If u is connected to many nodes, it is junction and the process terminates.
        if junctions != 1:
            break
        else: #Else update the path.
            path = path + (nextNode,)

    dict[path] = dict.pop(oldPath) #Update dict with the expanded path.

    if destinationNode in path: finalPath = True #If path is final set finalPath to True.

return dict, finalPath
```