Dynamic Search of Train Shortest Routes Within Microscopic Traffic Simulators

ANTONÍN KAVIČKA AND ROMAN DIVIŠ
Faculty of Electrical Engineering and Informatics, University of Pardubice, 532 10 Pardubice, Czech Republic
Corresponding author: Antonín Kavička (antonin.kavicka@upce.cz)

This work was supported by the European Regional Development Fund (ERDF)/European Social Fund (ESF) “Cooperation in Applied Research between the University of Pardubice and companies, in the Field of Positioning, Detection and Simulation Technology for Transport Systems (PosiTrans)” under Grant CZ.02.1.01/0.0/0.0/17_049/0008394.

Abstract

Computer simulations are frequently used for rail traffic optimization. This approach, referred to as simulation-based optimization, typically employs simulation tools – simulators that are designed to examine railway systems at various levels of detail. Microscopic rail traffic simulators find use when examining rail traffic and the rail infrastructure in great detail. Such simulators typically serve to follow the positions and motions of rail vehicles (trains, locomotives, train cars) and their relocation as well as segments of the rail infrastructure (tracks, switches, track crossings). One of the typical problems to be solved by microscopic simulators within a simulation experiment is to determine the realistic (optimal) train and shunting routes (within the currently occupied infrastructure) along which the rail vehicles are moved. This paper describes novel dynamic route searching algorithms applicable to the relocation of rail vehicles within track infrastructure of railway systems. The following main topics are presented in turn: overview of solutions to the problem of finding track routes in the literature, a suitable rail infrastructure model (associated with algorithms that seek admissible routes for the transfer of the relocation objects of given lengths), graph search algorithms computing the shortest track routes (represented by the admissible shortest walks on graphs), illustrative examples of algorithms’ deployment, computational complexity of presented algorithms, comparison with other algorithms and summary of the benefits of newly developed algorithms. The use of the algorithms within the simulation tools (working at the microscopic level of detail) extends the modelling possibilities when searching for realistic track routes (especially for complicated shunting operations), which contributes to better modelling of complex railway traffic (than in the relevant existing rail traffic simulators) and thus to better application of the results of traffic simulations in practice.

Index Terms
Rail traffic, simulation-based optimizations, single-train shortest routes.

I. INTRODUCTION

Traffic simulation models built at a microscopic level of detail (microscopic traffic simulators), serving to examine and optimize rail traffic systems, may be capable of mirroring real (or designed) systems as authentically as possible.

In practice, microscopic traffic simulation models examining various scenarios of simulation experiments are typically used, e.g., to address the following tasks:

■ Detailed examination of the traffic properties of certain track infrastructure parts (within feasibility studies).

The associate editor coordinating the review of this manuscript and approving it for publication was Zhigang Liu.

This type of simulation investigation is mainly applied when assessing the impact of a planned rail infrastructure reconstruction or upgrade on railway traffic. Thus, it is appropriate to compare the characteristics of the relevant traffic, for example, for the existing variant of the infrastructure configuration and for the variant after the reconstruction.

■ Determination of the traffic characteristics when launching new transport technological processes. The introduction of new innovative technological procedures of train handling (e.g., simultaneous shunting technique within railway yards), which can be tested by computer simulation before their deployment, represents a very good potential for increasing the efficiency of railway operation.
Optimization of the decision-making strategies when addressing selected traffic problems (such as the assignment of alternative platform tracks to delayed trains) applying a specific optimization criterion (e.g., associated with the minimization of the sum of delays of all trains during the period examined). Simulation tools are very useful for testing various decision supports for different types of operational problems occurring randomly during the railway traffic under investigation. Innovative supports that show good results are appropriate candidates for application in practice.

Testing the robustness of train schedules/timetables and work schedules for rostering resources concerning the occurrence of random traffic disruptions (such as train delays). Simulation testing of the resilience of timetables (especially for selected critical areas on the rail network) contributes to the elimination of risks associated with the deployment of inappropriate variants of traffic plans.

When setting up the microscopic simulators, the simulation computations must mirror the processes as faithfully as possible. Among the examples of important tasks is the setting of admissible train and shunting routes for the relocation of rail vehicles within a specifically occupied track infrastructure. For well-designed automated computations to be included in a simulator, it is necessary to design and implement: (i) an appropriate track infrastructure model and (ii) efficient algorithms working over that model and computing the route topologies for the specified relocations. And it is the original infrastructure model and the original algorithms that constitute the subject of this paper.

This article follows up the paper [1] addressing the mesoscopic level of investigation, which proved to be useful when designing and implementing both a track infrastructure model and route searching algorithms for rail vehicle relocations. Mesoscopic simulators examine the rail traffic primarily for determining the infrastructure’s traffic capacity rather than for faithful monitoring of all rail vehicle motions.

II. RELATED LITERATURE
Traffic simulations are widely used for rail traffic optimizations (simulation-based optimizations) with respect to specific criteria [2], [3]. For example, one of the global traffic indicators that should be minimized is the sum of the weighted delays of all trains [4] within the railway network segment of interest. A simulation experiment requires many partial optimization problems to be solved during the run (such as the assignment of alternative platform tracks to delayed trains; determining the priorities for entering the line tracks if the timetable is not followed; searching for the shortest admissible routes for rail vehicle relocations within the rail infrastructure; appropriate location of the service resources within the infrastructure; etc.), where different model types can be applied to the different partial problems. The use of partial optimizations facilitates the optimization of the traffic within the specific part of the railway network during the period of interest.

When solving the problem of searching for train routes on a track infrastructure, the route optimization can be based on a minimization of the route length. This problem constitutes a part of a wider problem called the train routing problem, which includes, among other things, the following tasks: route assignment to the trains within extensive railway network areas [5]; coordinated assignment of track routes to multiple trains in the railway stations [6], [7], [8], [9]; identification of the single train shortest route [1], [10], [11], [12]; etc. The layout of the track infrastructure on which the traffic takes place must be taken into account when addressing problems of this type. From this point of view, the specific type of investigation requires an appropriate infrastructure model and efficient algorithms. Among the track infrastructure models that are frequently used is the graph (and its implementations) as a mathematical concept dealt with graph theory [13]. The graph implementations are associated with the field of data structures and algorithms [14]. The models can use both directed graphs and undirected graphs. Many original modifications of the standard graphs are currently used, such as the double vertex graph [15], [16] and the doubly-weighted (edge-weighted and vertex-weighted) digraph [1], where the polar graph concept is loosely applied [10], [17]. Among typical problems addressed by the graph algorithms over the appropriate models is the identification of the admissible (potentially shortest) track routes along which the rail vehicles are relocated. Here, it may be advantageous to use the concept of the initial Dijkstra's algorithm [11], [14], [18], [19], which searches for the single-source shortest paths on edge-weighted directed graphs. Many modifications of this algorithm are also used when examining railway systems [20], [21], [22].

The above-mentioned rail traffic simulations can be applied in conjunction with various specialized simulation tools, which can employ different rail infrastructure sub-models [1], [15], [16], [22]. Examples of relevant simulation tools used for such simulations include OpenTrack [23], [24], RailSys [25], [26], [27], Villon [28], MesoRail [4], NEMO [29], PULSim [30], Simarail [31] etc. Such tools apply the same level of detail of investigation (microscopic, mesoscopic, or macroscopic) within one simulation model. However, approaches also exist where different levels of detail are applied in the different parts of one simulator: such simulations are referred to as multi-scale simulations [32] or hybrid simulations [33], [34].

The extension of the research the results of which were published in [1] and [10] was driven by the need to broaden the modelling capabilities of the MesoRail simulation tool (briefly characterized in the supplementary material to the article [4]). This tool was initially used for mesoscopic rail traffic simulations only. As efforts were made to expand the tool to apply to rail traffic simulations at the microscopic level of detail, it became necessary to implement innovated functions for automated dynamic computations of the track routes along which the relocation objects (trains, locomotives) will be transferred (within the microscopic rail
infrastructure model). To this end, the relevant initial algorithms, which have not been published so far, were re-designed and implemented. Such one-phase algorithms search the shortest admissible walks in the undirected edge-weighted graph representing the track infrastructure model. The walks represent the shortest routes within the track infrastructure. The implementations of the algorithm were successfully tested, integrated, and then tested within the MesoRail tool. The microscopic infrastructure model and the algorithms working over it constitute an original solution which (based on available literature) has not been used elsewhere. Conceptually closest to the above algorithms are approaches published in [11] and [12], where two-phase algorithms are used. The first, preprocessing phase serves to identify those track switches behind which reversals of trains (whose length is L) are feasible. The next step consists in the creation of a transformed model/graph enabling the shortest route to be computed by using original Dijkstra’s algorithm during the second, routing phase. The computation result from this phase can be transformed into the shortest route in the track yard. However, the use of two-phase algorithms within very frequently called operations (searching the shortest route during the simulation experiment) is rather inconvenient from the time aspect because the preprocessing phase must be applied to each switch (regardless of its utilization). In contrast to the approaches applied in [11] and [12], the algorithms described herein compute the feasibility of train motion reversals only for those track switches that were actually reached within the computation spreading.

III. TRACK INFRASTRUCTURE MODEL

The track infrastructure model, which represents a part of the traffic simulator’s state space, is represented by a specific edge-weighted undirected graph G. The definition of the graph is presented in Table 1. The edge weights (determined through the function $\omega$) match the physical lengths of the tracks/track segments modelled by the edges. The graph edges represent both tracks (and are referred to as destination edges) and track segments (and are then referred to as connecting edges), which can be parts of track crossings or switches. The terms mirror the assumption that while the tracks serve as the relocation object destinations, switches are the connecting elements between different tracks.

So, when specifying a rail vehicle relocation requirement, both the relocation start, and finish are assumed to be represented by specific tracks (modelled by the appropriate destination edge). Furthermore, the track end over which the vehicle should leave the starting track and the track end over which the vehicle should enter the destination track can also be specified. The track ends are represented by elements of the set $I(G)$.

Function $\kappa$ is introduced for graph G: this function assigns to the two opposite ends of each edge their edge ingress vacancies. The edge ingress vacancy value mirrors the length of the free part of the relevant track from the specific end. If the two edge ingress vacancies of an edge are equal to the weight of that edge, then the track (track segment) is considered free and passable. If this is not so, the track is either occupied by a rail vehicle (vehicles) or else it is fully locked by the railway interlocking system (in which case the ingress vacancies are zero). Specific weights of selected edges and values of the ingress vacancies of selected edges are illustrated in Example 1 below.

| Table 1. Specification of the edge-weighted undirected graph G - the model of track infrastructure. |
|---|
| Symbols | Specifications |
| $G$ | The weighted undirected graph $G=(V,E,\varphi,\omega,\kappa)$ |
| $V(G)$ | The set of vertices of the graph G |
| $E(G)$ | The set of edges of the graph G $E(G)=E_{dest}(G)\cup E_{conn}(G)$, $E_{dest}(G)\cap E_{conn}(G)=\emptyset$ |
| $I(G)$ | The set of edge-vertex elements $I(G)=[(e,v), e\in E(G), v\in V(G), \varphi(v,e)]$ |
| $\omega$ | The edge weight function related to the graph G $\omega: E(G)\rightarrow \mathbb{R}^+ (the set of positive real numbers)$ |
| $\varphi$ | The symbol $\{(v,e)\}$ denotes an unordered pair of vertices (undirected edge) $\forall v\in V(G)$, $\deg(v)\in \{1,2,3\}$, $m\leq \lceil(3/2)n\rceil$ (for n≥4, m=|V(G)|, n=|E(G)|) |
| $\kappa$ | The incidence function related to the graph G $\kappa: V(G)\rightarrow \mathbb{R}^+$ |
| $\kappa([e,v])$ | The ingress vacancy $\kappa([e,v])$, where $e\in E(G)$, $\varphi(e,v)=(v,e)\in E(G)$, $\omega([e,v])$ corresponds to a vacant capacity of a track segment (reflected by the edge $e_0$) via its defined end (reflected by the vertex v) – if $\kappa([e,v])=\omega([e,v])$ the track segment/edge $e_0$ is completely free and is not blocked by the interlocking system |

Track crossings must be discriminated from the various types of switches when setting up track yard models. To this end, a model as a subgraph of graph G is used for each track switch type and track crossing. The mental concept of the construction of such subgraphs is illustrated in Fig. 1.

First, the schematic symbols for track crossings and track switches are shown (as used in track yard layouts). Next, the
initial graph models that intuitively mirror the connections of the elementary track segments by using graph vertices are introduced. Finally, the final graph models whose topology is adequately modified to implement their admissible passages within the calculations of the track searching algorithms are presented. “Forbidden turns” between edge pairs modelling the partial track segments of a switch are graphically represented by grey triangles. This expresses the fact that direct relocations between selected segments belonging to the same switch are impossible.

The structure of the demonstration track yard model is presented in the illustrative example shown graphically in Fig. 2. First, the track yard layout is shown in Fig. 2(a). Next, the initial graph model is set up (Fig. 2(b)) as an intuitive transformation of the track yard layout to an undirected graph where each track is modelled by one edge and the switches are represented by subgraphs interconnected with the edges/tracks. The total length of the tracks and track segments is 3 418 m. Figs. 2(c) and 2(d) show the final track yard model as an edge-weighted undirected graph including the final track switch models (track switches: TS1-TS6) and weights of all edges in the graph.

IV. GRAPH SEARCH ALGORITHMS

At this stage of the introduction of the new algorithms for searching train and shunting routes in track yard models it should be emphasized that the algorithms are designed for computations over the model (edge-weighted undirected graph) specified in the previous section. From this point of view, the algorithms can be classed as graph search algorithms. The optimization performed by the algorithms is based on the searching for the shortest admissible walk in the graph whose topology mirrors the route in the track yard.

The inspirative starting point when designing the algorithms was Dijkstra’s algorithm [18], which is applicable when searching for the shortest routes between two vertices of an edge-weighted graph. Only such applications in which the edge weights are positive were relevant. However, this algorithm had to be fundamentally extended and redesigned to meet the requirements for the route computations in the track yard models. The algorithm was required to take into account the relocation object lengths and to enable inclusion of reversals in the track routes. Once the reversals are considered, the computation results of the new algorithms are interpreted as the shortest walks rather than the shortest paths, in contrast to Dijkstra’s algorithm.

Auxiliary sets, row vectors and subprograms (described in Tables 2-4) are used in the specification of the route search algorithms (in the form of pseudocodes).

In the subprograms, discrimination is made between the input parameters (labelled with the prefix ‘↑’), output parameters (with the prefix ‘↓’), and input-output parameters (with the two-sign prefix ‘↑↓’).

A. PRIMARY ALGORITHM SEARCHING FOR THE SHORTEST WALK

This primary algorithm (referred to as Algorithm 1) is focused on searching for the shortest walk in the track yard model:

- from a specific start edge (track) \( e_x \in E_{dest}(G), \varphi(e_x) = (v_a, v_b), v_a, v_b \in V(G) \) through its specified end vertex \( v_a \); the set \( S \) of start edge-vertex elements is constructed as \( S = \{[e_x, v_a]\} \)
- to a specific finish vertex (track) \( e_y \in E_{dest}(G), \varphi(e_y) = (v_y, v_b), v_y, v_b \in V(G) \) through its defined end vertex \( v_y \); the set \( F \) of finish edge-vertex elements is constructed as \( F = \{[e_y, v_y]\} \)

The shortest walk found mirrors the shortest route in the track yard for the relocation of an object \( O \) whose length is \( L \).

The basic Shortest_Path routine calculates the above-specified shortest walk whose topology it returns in the output parameter Seq. For Algorithm 1, |S|=1 and |F|=1. The algorithm modifications to be introduced later can use variants for \(|S|\in\{1,2\}\) and \(|F|\in\{1,2\}\).

The Shortest_Path routine computation is started with a test (within the Start_Finish_Test subprogram) to find if the route can be computed for the given relocated object’s start and finish positions. The route cannot be sought if \( S = F \) or if the finish edge ingress vacancy is smaller than the relocation object length, \( \kappa([e_y, v_y]) < L \). If the route should be found between the two opposite ends of an edge/track \( e_x = e_y \land [e_x, v_a] \neq [e_y, v_y] \), the ingress vacancy of that edge with respect to the final relocation position is temporarily modified – updated – through the Vacancy_Update function. The updated value \( \kappa([e_y, v_y]) \) mirrors the track (edge \( e_y \)) occupancy after the relation object \( O \) has left.

The General_Init subprogram performs general initialization operations concerning the:
- vertex-edge marks (contained in the sets stored in the elements of the row vector \( M \)),
- distance marks of vertices (stored in the elements of the row vector \( D \)) and
FIGURE 2. Demonstration track layout models.
TABLE 2. Specifications of auxiliary sets related to the weighted undirected graph G.

| Symbols | Specifications |
|---------|----------------|
| Z       | The set of forbidden turns on the graph $G$  
$Z = \{(c_o,c_v) | c_o,c_v \in E\text{con}(G), c_o \neq c_v, \varphi(c_o) = (v_v,v_o), \varphi(c_v) = (v_v,v_o), v_v,v_o \in V(G)\}$  
The forbidden turn $(c_o,c_v) \in Z$ expresses the technical inadmissibility of a direct train transfer from the track segment/edge $e_o$ to $e_v$ and from $e_v$ to $e_o$. |
| $\text{adj}W(v_o,e_u)$ | The set of adjacent vertices of vertex $v_o$ considering the excluded edge $e_u$  
$\text{adj}W(v_o,e_u) = \{(v_v,v_o) | v_v \in V(G), v_v \neq v_o, \varphi(v_v) = (v_v,v_o), v_v \in E(G), v_v \neq e_u\}$  
$\text{adj}W(v_o,e_u) = \{\text{Wtrans}(v_o,e_u) \cup \text{Wrev}(v_o,e_u) \cup \text{Wtrans}(v_o,e_u) \cap \text{Wrev}(v_o,e_u) = \emptyset\}$  
The set $\text{adj}W(v_o,e_u)$ of transit adjacent vertices of vertex $v_o$ considering the excluded edge $e_u$,  
$\text{adj}W(v_o,e_u) = \{(v_v,v_v) | v_v \in V(G), v_v \neq v_o, \varphi(v_v) = (v_v,v_o), v_v \in E(G), v_v \neq e_u, (v_v,v_o) \in E, (v_v,v_o) \in G\}$,  
$\text{adj}W(v_o,e_u) \in \{0,1,2\}$  
The set $\text{adj}W(v_o,e_u)$ of reverse adjacent vertices of the vertex $v_o$ considering the excluded edge $e_u$,  
$\text{adj}W(v_o,e_u) = \{(v_v,v_o) | v_v \in V(G), v_v \neq v_o, \varphi(v_v) = (v_v,v_o), v_v \in E(G), v_v \neq e_u, (v_v,v_o) \in E, (v_v,v_o) \in G\}$,  
$\text{adj}W(v_o,e_u) \in \{0,1\}$ |
| S       | The set of start edge-vertex elements  
$S \subseteq G, |S| \in \{1,2\}$  
$\forall [s_o,v_i] \in S : s_o \in E\text{con}(G)$  
If $|S| = 2$ then $S = [(c_o,v_i),(c_v,v_o)]$, $\varphi(n) = (v_v,v_o)$ |
| F       | The set of finish edge-vertex elements  
$F \subseteq G, |F| \in \{1,2\}$  
$\forall [s_o,v_i] \in F : e_f \in E\text{con}(G)$  
If $|F| = 2$ then $F = [(c_o,v_i),(c_v,v_o)]$, $\varphi(n) = (v_v,v_o)$ |
| $T_v$   | The dynamic set of temporarily marked vertices  
$T_v \subseteq X$  
$X = \{(v_o,e_u,q) | v_o \in V(G), q \in \mathbb{R}^+, e_u \in E(G), v_o \in \varphi(e_u)\}$  
The ordered triplet $(v_o,e_u,q)$ reflects the vertex $v_o$ (reached via $e_u$ edge) tagged by the relevant marking value $q$ |
| $U_v$   | The dynamic set of ultimately marked finish edge-vertex element  
$U_v \subseteq F, |U_v| \in \{0,1\}$ |

- dynamic sets $T_V$ and $U_V$ and reversal paths beyond all switches.

Hence, the elements of vector $M$ are sets. Set $m_k$ represents the $k$-th element of vector $M$ which is associated with the vertex $v_k \in V(G)$. Cardinality of each set $m_k$, $k \in \{1, \ldots, |V(G)|\}$, equals the degree of vertex $v_k$: $|m_k| = \deg(v_k)$. The elements of the sets $m_k$ are two-component marks $(\text{len}_j,e_j) \in m_k$, $e_j \in \mathbb{Y}(v_k)$, $j = 1, \ldots, \deg(v_k)$. The dynamic length-component $\text{len}_j$ fixes the current minimum length of the walk from the start position to the vertex $v_k$ attained from the “direction” determined by component $e_j$. The static edge-component $e_j \in \mathbb{Y}(v_k)$ contains permanent information about the edge that is incident with vertex $v_k$. The values $d_k$ of the elements of vector $D$ $(d_k \in \mathbb{R}^+, k = 1, \ldots, |V(G)|)$ mirror (for each vertex $v_k$ of the graph) the minimum of the length-components: $d_k = \min(\text{len}_j), (\text{len}_j,e_j) \in m_k, e_j \in \mathbb{Y}(v_k)$. Set $T_V$ saves (during the calculation of Algorithm 1) the temporarily marked vertices of graph $G$. The elements of set $T_V$ are represented by the ordered triplets $(v_i,e_u,q)_k$ where $(v_i,e_u) \in F, e_u \in E(G), q \in \mathbb{R}^+$, which contain the specification of vertex $v_i$ (labelled with mark $q$) attained from the direction/edge $e_u$. Set $U_V$ potentially contains the ultimately marked finish edge-vertex element $(e_u,v_i)$ where $(e_u,v_i)_k \in F, e_u \in E(G), v_i \in V(G), v_i \in \varphi(e_u)$.  

All elements of vector $D$ are initialized to the value $\text{dist}\infty \in \mathbb{R}^+$, a constant which is higher than the highest admissible walk length in the graph $G$. For each set $m_k$ $(k = 1, \ldots, |V(G)|)$ from vector $M$, the length-components of all of its elements are initially set to the value $\text{dist}\infty$. Sets $T_V$ and $U_V$ are initially set empty, the reversal paths beyond all switches are set to the status $\text{notcalc}$ (not yet calculated).

The initialization procedures (performed through the function $\text{Start}_\text{Finish}_\text{Init}$) are finalized by adjustment procedures concerning the start and finish positions of the relocation route being sought. The relocation start vertex $v_a$
### TABLE 3. Specifications of auxiliary sets and row vectors associated with the weighted undirected graph G.

| Symbols | Specifications |
|---------|----------------|
| Y(v)   | Set of the incident edges to the vertex v; Y(v)={(e,Q(e))=(v,v), e∈E(G), v,v∈V(G), v≠v} Y((v))∈{(1,2,3)} |
| M      | Global row vector of the vertex-edge marks M=[[m]]k=1,..., |V(G)| m=[|len(e)|len∈R, e∈Y(v), v∈V(G)] | m_0=deg(v) |
|        | Each vertex v∈P(G) is tagged by a dynamic set of two-component marks, the dynamic length-component len, expresses the length of the current shortest walk from vertex v∈E[v,v]∈E to the vertex v which is directly reached just via the static edge-component e∈Y(v) |
|        | If the above-mentioned walk to the vertex v∈E(G) via e does not exist, then len=dist (dist∈R and it is equal to the value that is greater than the length of the longest admissible walk in the graph G) |
| D      | The global row vector of minimum distance marks D=[[d]]k=1,..., |V(G)| d_0=|len(e)|{len(e)}∈E, e∈Y(v), v∈V(G) | d_0 represents the minimum value from length-component from m_0 (associated with the vertex v∈E(G)) |
| Seq    | Linearly ordered dynamic set containing the found route topology Seq={[[dir,|e|]]k=1,..., |E(G)|, |E(G)|=|Seq|} |
|        | The i-th element of Seq ([|dir,|e|])∈Seq reflects the i-th edge/vertex pair ([|e|,|E(G)|) lying on the modelled train route |
|        | If e corresponds to reaching the track/edge e via vertex v then dir=in; if e mirrors leaving the track/edge e via vertex v then dir=out |
|        | The route end position is modelled by the element [[1,|in,|e|]]∈Seq |

((e, v, a)∈E) is labelled with the mark q=κ((e, v, a)), the value of which expresses the following fact: If the relocation object O stands on track (edge) e (at a certain distance from the end v through which it will leave), then the distance q to the relevant end of track e must be run first. So, the value q in the context of vertex v plays a role in the following settings of the D and M set elements: d_0←q, Set_Mark(e, |m|, e, q). Now, the first element [v, e, q], mirroring the status at the beginning of the route search process, is inserted into set T_V. If the finish track/edge e_f is completely free and the relocation object should enter it through its end/vertex v (i.e., e_f=(v, v)), then the ingress vacancy of that edge/track with respect to the opposite end/vertex v, is temporarily reduced: κ((e, v, a))=(ω(e_f))−ε, ε∈R^+. The aim of this provision is to prevent transit of the edge/track e_f from the “direction” of the track v (the transit cannot occur if κ((e, v, a)))<ω(e_f)).

The algorithm continues by starting the main computation cycle. It first selects (by means of the subprogram VeriSelect using the routine Min_Dist) the vertex v_c in the set T_V to be processed within the next iteration step. The value of the distance mark q of the selected vertex v_c is the lowest among the vertices in the set T_V. Next, the element [v_c, e_c, q]∈T_V, which contains the vertex v_c, is removed from the set T_V. Subsequently, the vertices that are adjacent to the vertex v_c are potentially re-marked (through the Adj_Mark routine), if the routes to them are more advantageous than the existing ones. The computation cycle is terminated if one finish vertex has been ultimately marked (U_V≠∅). The alternative termination condition (T_V=∅) mirrors the fact that the computation cannot propagate along the infrastructure model anymore (because no vertex from which additional improvement of the existing routes could be sought is available).

If an adjacent vertex v∈adj_W(v_c, e_c) exists, then it can be reached from vertex v_c via the track/edge e (i.e., v∈E[v_c, v]) provided that the latter is currently passable (κ((e, v, a))=ω(e_c)). Vertex v_t can be re-marked by means of the Try_Change_Transit subprogram (using the Set_Mark function) to a new distance value newdist←(q+ω(e_c)) if the relation newdist<currdist, [currdist, e_c]∈m_0 holds true. The newdist value expresses the fact that a distance that is equal to the weight of edge e (i.e., ω(e_c)) must be run to reach the end (vertex v_t) of the track/edge e. Once vertex v_t has been re-marked, the appropriate element [v_t, e_t, newdist] is inserted into the set T_V.

If an adjacent vertex v∈adj_W(v_c, e_c) exists, then it can be reached via edge e (i.e., v∈E[v_c, v]) provided that the
| Subprograms/functions | Specifications |
|-----------------------|---------------|
| ShortestWalk(4S,4F,4L,4Seq) | Calculation of the shortest walk on the edge-weighted undirected graph G  
S – set of the start edge-vertex elements  
F – set of the finish edge-vertex elements  
L – relocation object length  
Seq – the topology of the shortest walk found |
| StartFinishTest(4S,4F,4L,4Iokay) | Test of correctness of start and finish elements  
S – set of the start edge-vertex elements  
F – set of the finish edge-vertex elements  
L – relocation object length  
okay – the test result |
| GeneralInit(4L) | Execution of the general initialisation activities  
Initialisations of the sets T_v, U_v and the row vectors M, D  
Reversal paths (of the length L) beyond all switches (e_o,e_i)ÎZ are set to the status notcalc (not yet calculated) |
| StartFinishInit(4S,4F) | Initialisation activities reflecting the start elements and finish elements  
S – set of the start edge-vertex elements  
F – set of the finish edge-vertex elements  
First insertion into set T_v, the first modifications of vectors M, D, potential modification of the value κ({e_o,v_i}), [e_o,v_i]ÎF |
| VertSelect(4F,4v_o,4e_o,4Iq) | Selection of vertex v_o (reached via edge e_o) associated with the distance mark q for the processing in the next step of the algorithm  
F – set of the finish edge-vertex elements  
The element [v_o,e_o,q] is removed from set T_v, set U_v is potentially updated |
| UploadPath(4v_o,4e_o,4e_i,4Parroute) | Saving the topology (Parroute) related to the path found (by the DFS-algorithm) for the train reversal  
The k-th path element is defined as follows: [k,nk,[e_o,v_i]]ÎParroute, k=|Parroute|,  
Y(v_i)={e_o,e_i}, (e_o,e_i)ÎZ, ϕ(e_o)={v_i,v_o}, ν_i,ν_o Îϕ(e_o) |
| DownloadPath(4v_o,4e_o,4e_i,4Parroute) | Delivery of the topology (Parroute) related to the previously saved path (by the function UploadPath(4v_o,4e_o,4e_i,4Parroute)) for the train reversal |
| GetPathStatus(4e_o,4e_i,4L,4status) | Auxiliary function providing the status of a reversal path (of the length L) beyond the switch (e_o,e_i)ÎZ  
status∈{notcalc,exist,notexist}  
notcalc – the path not calculated yet  
exist – the path exists  
notexist – the path does not exist |
| SetPathStatus(4e_o,4e_i,4L,4status) | Auxiliary function setting the status of a reversal path (of the length L) beyond the switch (e_o,e_i)ÎZ to a given value  
status∈{notcalc,exist,notexist} |
| AdjMark(4v_o,4e_o,4Iq,4L) | Potential rewriting of the marks belonging to the adjacent vertices of vertex v_o (reached via edge e_o)  
L – relocation object length  
q – distance mark associated with vertex v_o (from the element [v_o,e_o,q] just removed from set T_v) |
TABLE 4. (Continued.) Specifications of subprograms/functions.

| Subprogram | Specification |
|------------|--------------|
| Get_Walk([i]Seq) | Delivery of the topology (Seq) related to the shortest walk found |
| Min_Dist([Tv,Te,v0,q]) | Auxiliary function identifying vertex v0 reached via edge e0 within set Tv tagged with the minimum value of the relevant mark q |
| Try_Change_Reverse([iv,ie,v0,q,iv,il]) | Auxiliary function potentially rewriting the mark belonging to the reverse adjacent vertex νi∈E\W_rel(νi,v0) of vertex νv (with regard to its incident edge e0 and its mark value q) |
| Try_Change_Transit([iv,ie,v0,q,iv,w]) | Potential update of set Tv and the row vectors M, D |
| Get_Mark([im,v0,q]) | The reversal path for the train (whose length is L) is potentially calculated (by the DFS-algorithm) |
| Set_Mark([im,v0,q]) | Auxiliary function providing the value q reflecting the minimal length of the walk to vertex νv via the edge e0, [q,e0]∈Em |
| Depth_First_Search([iv,ie,v0,q,ParTroute,il]) | Auxiliary function (DFS-algorithm) searching the first available path (ParTroute of the length L) providing the place for train reversal behind the switch (e0,e0)∈E, e0∈φ(νs) V, νs∈φ(νs) L V, νs∈φ(νs) V |
| Get_Index([iv,w]) | The k-th path element is defined as follows: [k,ιn([e0,v0])]∈ParTroute, k=|ParTroute|, Y(νv)=e0,e0,e0, (e0,e0)∈E, φ(νv)=νv, νs∈φ(νs) V, νs∈φ(νs) V |
| Vacancy_Update([i]k([e0,v0])) | Auxiliary function temporarily updating (just for the current run of the Shortest Walk function) the ingress vacancy of a defined edge/track e0 with its incident vertex v0; it reflects the case where the edge e0 represents both the starting and finish tracks of the walk sought |
| Select_Element([i]Seq,ii,[i,νv]) | Auxiliary function selecting the pair [e,νv] from the i-th element of set Seq, [i,dir,[i,νv]]∈Seq |
| Pull_Out([iU],1[i,v]) | Auxiliary function removing one element ([i,v]) from the set U, |U|=1 |
| Insert_Reversal([iParTroute,ii,dir,i]Seq) | Auxiliary function inserting the topology of the partial walk (constructed on the path ParTroute) into the currently formed final walk Seq |
| Pred_Edge([i][e,v0],[e0]) | Auxiliary function delivering edge e0 representing the predecessor of the edge-vertex pair [e0,v0] lying on the shortest walk between two defined tracks/edges |
| Pred_Vertex([i][e,v0],[νv]) | The relevant predecessor is identified by evaluating distance the marks from set m0 belonging to vertex νv and distance marks from the relevant sets m0 belonging to all corresponding adjacent vertices νv∈E\W(νv,e0) |

The latter is currently passable (κ([e0,v0])=ω(e0)). In addition, it holds for the edges e0 and e0 that ([e0,e0])∈E. Those edges represent a “forbidden turn”, i.e., they mirror two track segments of the same switch between which direct relocation of rail vehicles is technically impossible. The following two assumptions must be met for vertex νv to be re-markable in the Try_Change_Reverse subprogram:

- First, the relation new\dist<currdist, [currdist,e0]∈m0, must apply to enable vertex νv to be re-marked to a new distance value new\dist←(q+ω(e0)+L) where L mirrors the
Algorithm 1 Computation of the shortest walk from $S = [e_x, v_a]$ to $F = [e_y, v_g]$

01 function Shortest_Walk(↓ S, ↓ F, ↓ L, ↑ Seq)
02 Seq ← ∅
03 correct ← true
04 Start_Finish_Test(↓ S, ↓ F, ↓ L, ↑ correct) // admissibility of the start/finish vertices
05 if correct then
06 General_Init(↓ L) // setting initial marks of all vertices
07 Start_Finish_Init(↓ S, ↓ F) // initialisations related to the start/finish vertices
08 repeat
09 if $T_V ≠ ∅$ then
10 Vert_Select(↓ F, ↑ v_c, ↑ e_u, ↑ q) // selection of a new current vertex $v_c$
11 if $U_V = ∅$ then
12 Adj_Mark(↓ v_c, ↓ e_u, ↓ q, ↓ L) // potential marking of adjacent vertices of $v_c$
13 end
14 end
15 until $T_V = ∅$ or $U_V ≠ ∅$ // algorithm termination testing
16 if $U_V ≠ ∅$ then
17 Get_Walk(↑ Seq) // getting a topology of the shortest walk
18 end
19 end
20 end
21 function Start_Finish_Test(↓ S, ↓ F, ↓ L, ↑ okay)
22 if $S = ∅$ or $F = ∅$ then
23 okay ← false
24 exit
25 end
26 if $[e_x, v_a] = [e_y, v_g]$ or $κ([e_y, v_g]) < L$ then // $[e_x, v_a] ∈ S$, $[e_y, v_g] ∈ F$
27 okay ← false // inadmissible combination of the start and finish edges-vertices
28 exit
29 end
30 if $e_x = e_y$ and $[e_x, v_a] ≠ [e_y, v_g]$ then // $[e_x, v_a] ∈ S$, $[e_y, v_g] ∈ F$
31 Vacancy_Update(↑ $(e_x, e_y)$) // an update of edge-vertex element vacancy
32 end
33 end
34 function General_Init(↓ L)
35 for $i ← 1$ to $n$ do // $n = |V(G)|$
36 $m_i ← ∅$
37 $d_i ←$ dist $∞$
38 for each $e_j ∈ Y(v_i)$ do
39 $m_i ← m_i ∪$ dist $∞, e_j$
40 end
41 end
42 for each $(e_u, e_s) ∈ Z$ do
43 Set_Path_Status(↓ e_u, ↓ e_s, ↓ L, notcalc)
44 end
45 $T_V ← ∅$
46 $U_V ← ∅$
47 end

The newdist value expresses the fact that for reaching the end (vertex $v_r$) of the track segment (edge) $e_s$, the object $O$ must first pass through the track segment (edge) $e_u$ and be completely relocated “behind” the appropriate switch (“behind” vertex $v_c$). Then the object $O$ motion direction must be changed to continue the relocation operation to the track segment (edge) $e_s$. This reversal is associated with a trajectory the length of which is $L$. 

90172
Algorithm 1 (Continued). Computation of the shortest walk from S=[e_s,v_a] to F=[e_y,v_g]

```plaintext
function Start_Finish_Init(↓ S, ↓ F)
  Get_Index(↓ v_a, ↑ a)
  q ← κ([e_s,v_a]) // [e_s,v_a] ∈ S
  Set_Mark(↓ m_s, ↓ e_s, ↓ q) // initial marking of the start vertex
  d_s ← q
  T_V ← T_V ∪ [v_s,e_s,q] // initial insertion of the start vertex into the set T_V
  if e_s ≠ e_c and κ([e_y,v_g]) = ω(e_y) then // [e_s,v_a] ∈ S, [e_y,v_g] ∈ F, ϕ(e_y) = (v_g,v_h)
    κ([e_s,v_g]) ← ω(e_y) − ε / ε ∈ R^+ , negligible small value compared to the weights of edges
  end
end

function Vert_Select(↓ F, ↑ v_c, ↑ e_u, ↑ q)
  Min_Dist(↓ T_V, ↑ v_c, ↑ e_u, ↑ q) // selection of a vertex v_c with the lowest distance mark q
  T_V ← T_V ∩ [v_c,e_u,q]
  if v_c ∈ ϕ(e_y) then // [e_y,v_c] ∈ F
    U_V ← U_V ∪ [e_y,v_c]
  end
end

function Adj_Mark(↓ v_c, ↓ e_u, ↓ q, ↓ L)
  if adj W_{trans}(v_c, e_u) ≠ ∅ then
    for each v_t ∈ adj W_{trans}(v_c, e_u)
      Try_Change_Transit(↓ v_c, ↓ e_u, ↓ q, ↓ v_t) // potential remarking of v_t
    end
  end
  if adj W_{rev}(v_c, e_u) ≠ ∅ then
    for each v_t ∈ adj W_{rev}(v_c, e_u)
      Try_Change_Reverse(↓ v_c, ↓ e_u, ↓ q, ↓ v_t, ↓ L) // potential remarking of v_t
    end
  end
end

function Try_Change_Transit(↓ v_c, ↓ e_u, ↓ q, ↓ v_t)
  Get_Index(↓ v_c, ↑ c)
  Get_Index(↓ v_t, ↑ t)
  if κ([e_s,v_c]) = ω(e_s) then // if the edge is completely vacant; ϕ(e_s) = (v_c,v_t)
    Get_Mark(↓ m_s, ↓ e_s, ↑ currdist)
    newdist ← (q + ω(e_s))
    if newdist < currdist then
      if newdist < d_t then
        d_t ← newdist
      end
      Set_Mark(↓ m_s, ↓ e_s, ↓ newdist)
      T_V ← T_V ∪ {[v_t,e_s,newdist]}
    end
  end
end
```

- Second, free infrastructure must be available “behind” the switch, which means that there must be available a partial track route whose length is no shorter than L, to which the object O will be temporarily relocated (in the track yard model, this route is represented by a vacant path in the graph). The route is searched by the Depth_First_Search routine using a depth-first search (DFS) algorithm. If the route is found, its topology is saved (by means of the Upload_Path function) for later partial use when setting up the topology of the finish shortest walk.

Once the vertex v_c has been re-marked (by means of the Set_Mark function), the appropriate element [v_t,e_s,newdist] is inserted into the set T_V.
A few facts are noteworthy as regards the *DFS-algorithm* computing strategy. This algorithm uses the auxiliary *stack* data structure, applying the *LIFO (Last In First Out)* strategy for the order of the vertices processed during the computation progress. The ongoing results of the *DFS-algorithm* computations are saved in local auxiliary row vectors (\( \mathbb{D} \) and \( \mathbb{P} \)). Additional auxiliary subprograms are listed in Table 5. During the algorithm computations, vector \( \mathbb{D} \) stores the distance mark values of the vertices of graph \( G \), mirroring the current lengths of the paths to the vertices from the starting vertex \( v_c \).

Each element \( \mathbb{P}_k, k=1, \ldots , |V(G)| \) of vector \( \mathbb{P} \) saves information on the predecessor of vertex \( v_k \in V(G) \) within the topology of the currently found path from the start vertex \( v_c \) to vertex \( v_k \). The main computation cycle is based on a stepwise withdrawal of elements from the stack and potential re-marking of their adjacent transit vertices if they can be reached through paths that are shorter than the currently found ones. The *DFS-algorithm* computation is terminated once it has found the first path from vertex \( v_c \) whose Path-Length meets the condition \( \text{PathLength}\geq L \). The topology of this path is made available through the *Get_Path* function, which makes reverse browsing of the path found (by using the marks-predecessors from vector \( \mathbb{P} \)). The computation will also be terminated if it finds that no path having the required length exists within the given (currently occupied) track yard model.

The final part of the main *Shortest_Walk* routine consists in the topology computation of the shortest walk found within graph \( G \). This computation uses the *Get_Walk* function, making reverse browsing from the finish element \( [e_c,v_c] \) reached (which has the property \( [e_c,v_c] \in F \land [e_c,v_c] \in U_c \)) as far as the start element of the walk \( [e_s,v_s] \in S \). The elements of the walk topology are inserted one by one into the linearly ordered dynamic set \( \text{Seq} \). The elements of set \( \text{Seq} \) are ordered triplets: \( [i,\text{dir},[e,v]] \), \( i=1, \ldots , k \), \( \text{dir}\in \{\text{in, out}\} \), \( [e,v] \in \text{I}(G) \), \( k=|\text{Seq}| \). The dir component expresses the direction associated with the passage of edge \( e \) through vertex \( v \) \( ([e,v]) \). If \( \text{dir} = \text{in} \), the \( i \)-th element of the walk is interpreted as entering the edge \( e \) through its side vertex \( v \). The opposite case \( (\text{dir} = \text{out}) \) is interpreted as leaving edge \( e \) through its side vertex \( v \). The finish position of the walk is mirrored by the element \( [1,\text{in},[e,v]]\in \text{Seq} \) and the start position of the walk is mirrored by the element \( [k,\text{out},[e,v]]\in \text{Seq} \). The auxiliary functions *Pred_Edge* and *Pred_Vertex* are used for the stepwise identification of the

---

**Algorithm 1 (Continued).** Computation of the shortest walk from \( S=[e_s,v_s] \) to \( F=[e_y,v_y] \)

```plaintext
092 function Try_Change.Reverse(↓v_c, ↓e_u, ↓q, ↓v_r, ↓L)
093 | Get_Index(↓v_c, ↑c)
094 | Get_Index(↓v_r, ↑r)
095 | if k([e_s,v_r])=ω(e_s) then // if the edge is completely vacant: \( \varphi(e_s)=(v_c,v_r) \), \((u_s,e_s)\in Z\)
096 | Get_Mark(↓m_v, ↓e_s, ↑currdist)
097 | newdist←(q+ω(e_s)+L)
098 | if newdist<currdist then
099 | Get_Path_Status(↓e_u, ↓e_s, ↑status)
100 | if status = notcalc then // if the reversal path is not known yet, try to find it
101 | Partroute ← ∅
102 | Depth_First_Search(↓v_c, ↓e_u, ↓e_s, ↓Partroute, ↓L)
103 | if Partroute=∅ then
104 | | Set_Path_Status(↓e_u, ↓e_s, ↓L, notexist)
105 | | exit
106 | | end
107 | | Upload_Path(↓v_c, ↓e_u, ↓e_s, Partroute)
108 | | Set_Path_Status(↓e_u, ↓e_s, ↓L, exist)
109 | else
110 | | if status = notexist then
111 | | exit
112 | | end
113 | end
114 | if newdist<d_e then
115 | | d_e ← newdist
116 | end
117 | Set_Mark(↓m_r, ↓e_s, ↑newdist)
118 | T_v ← T_v ∪ [ν_r, e_s, newdist]
119 | end
120 end
121 end
```
Algorithm 1 (Continued). Computation of the shortest walk from $S=[e_x,v_x]$ to $F=[e_y,v_y]$.

```plaintext
function Depth_First_Search($\downarrow v_c, \downarrow e_u, \downarrow e_v, \uparrow \text{Partroute}, \downarrow L$)
for i ← 1 to n do // n=|V(G)|
    $D_k$ ← dist$^\infty$
    $P_i$ ← none
end
Stack ← ∅
PathLength ← 0
Get_Index($\downarrow v_c, \uparrow c$)
$D_k$ ← 0
$P_k$ ← $v_u$ // $v_u \in \varphi(e_u)$, $\varphi(e_u) = (v_c, v_u)$, $v_u \neq v_c$
Push($\downarrow \uparrow$ Stack, $\downarrow v_c$) // insertion of a first vertex into the stack
while PathLength < L and Stack ≠ ∅ do
    Pop ($\downarrow \uparrow$ Stack, $\downarrow v_k$) // removal of the top vertex from the stack
    Get_Index($\downarrow v_k, \uparrow k$)
    PathLength ← $D_k$
    for each $v_t \in \text{adj} W_{\text{trans}}(v_k, e_k)$ do // $\varphi(e_k) = (v_k, P_k)$
        Get_Index($\downarrow v_t, \uparrow t$)
        if $D_k < D_k + \omega(e_u)$ then // $\varphi(e_u) = (v_k, v_t)$
            if $k([e_u, v_k]) + D_k > \text{PathLength}$ then // if the edge is completely vacant
                $D_k$ ← $D_k + \omega(e_u)$
                $P_k$ ← $v_k$
                Push($\downarrow \uparrow$ Stack, $\downarrow v_t$)
            end
        end
    end
    if $k([e_u, v_k]) + D_k > \text{PathLength}$ then // if found the new longest reversal path
        PathLength ← $k([e_u, v_k]) + D_k$
    end
end
if PathLength ≥ L then
    Get_Path($\uparrow \text{Partroute}$)
else
    Partroute ← ∅
end
end
```

predecessors for the walk elements. These functions evaluate (with respect to the current vertex $v_c$) the distance marks from those elements (of the set $m_k$) that are associated with the vertices $v_k \in \varphi(W_{\text{trans}}(v_c, e_c))$. Based on those evaluations, the vertex-predecessor or edge-predecessor is identified for each currently processed walk element (this procedure is illustrated in Example 1). A specific procedure is used if the reverse browsing finds that two consecutive walk elements should contain edges $e_c$ and $e_p$ for which $(e_c, e_p) \in Z$. These edges mirror the track segments of a certain track switch (TS) and are in the “forbidden turn” interrelation. It holds that the partial walk topology must be inserted “between” those elements. Data for the computation of that partial walk (represented by the path topology behind the track switch TS) were precalculated before by the Depth_First_Search subroutine and are made available by means of the Download_Path routine (through the Partroute output parameter).

The partial walk in question is built up and inserted on an ongoing basis into the currently constructed main walk by means of the Insert_Reversal subprogram. This subprogram first performs an “egress” traversal and then an “ingress” traversal of the route whose topology was provided by the Partroute parameter.

B. MODEL PROPERTIES AND IMPLEMENTATION COMMENTS

As regards the above primary algorithm, additional information concerning the implementation as well as the track infrastructure model properties can be presented:

- With respect to the properties of the graph $G$:

  $\forall v_i \in V(G): \deg(v_i) \in 1, 2, 3, m \leq \lfloor (3/2) n \rfloor, n \geq 4,$

  $n = |V(G)|, m = |E(G)|$
Algorithm 1 (Continued). Computation of the shortest walk from \(S=[e_x, v_a]\) to \(F=[e_y, v_g]\)

157 function Get_Walk(\(\downarrow\uparrow\)Seq)
158 Stop \(\leftarrow\) false
159 \(i \leftarrow 0\)
160 dir \(\leftarrow\) out
161 Pull_Out(\(\downarrow U_v, \uparrow [e_c, v_c]\)) // selecting the finish vertex
162 repeat
163 \(i \leftarrow i + 1\)
164 if dir = out then
165 \(\text{dir} \leftarrow\) in
166 else
167 \(\text{dir} \leftarrow\) out
168 end
169 Seq \(\leftarrow\) Seq \(\cup\) [\(i, \text{dir}, [e_c, v_c]\)]
170 if \([e_c, v_c] \in S\) then // if the start vertex is reached
171 Stop \(\leftarrow\) true
172 else
173 if dir = in then
174 \(\text{Pred}_\text{Edge}(\downarrow [e_c, v_c], \uparrow e_p)\)
175 \(e_c \leftarrow e_p\)
176 if \((e_c, e_p) \in Z\) then // if a reversal is supposed to be constructed
177 \(\text{Download}_\text{Path}(\downarrow v_c, \downarrow e_c, \downarrow e_p, \uparrow \text{Partroute})\)
178 \(\text{Insert}_\text{Reversal}(\downarrow \text{Partroute}, \downarrow \uparrow i, \downarrow \uparrow \text{dir}, \downarrow \uparrow \text{Seq})\)
179 end
180 else
181 \(\text{Pred}_\text{Vertex}(\downarrow [e_c, v_c], \uparrow v_p)\)
182 \(v_c \leftarrow v_p\)
183 end
184 until Stop
185 end
186 function Insert_Reversal(\(\downarrow\)Partroute, \(\downarrow\)i, \(\downarrow\)dir, \(\downarrow\)Seq) \(\rightarrow\)
187 k \(\leftarrow\) |Partroute| // the first cycle starts at the last Partroute element and goes back to the first element
188 first \(\leftarrow\) k
189 last \(\leftarrow 1\)
190 cycle \(\leftarrow 1\)
191 repeat
192 for \(j \leftarrow\) first to last do
193 \(\text{Select}_\text{Element}(\downarrow\)Partroute, \(\downarrow j, \uparrow [e, v])\)
194 \(i \leftarrow i + 1\)
195 if dir = out then
196 \(\text{dir} \leftarrow\) in
197 else
198 \(\text{dir} \leftarrow\) out
199 end
200 Seq \(\leftarrow\) Seq \(\cup\) [i, dir, [e, v]]
201 end
202 if cycle = 1 then
203 first \(\leftarrow 1\) // the second cycle starts at the first element and goes back to the last element
204 last \(\leftarrow k\)
205 end
206 cycle \(\leftarrow\) cycle + 1
207 until cycle > 2 // the function is terminated after finishing the first and the second cycle
208 end
TABLE 5. Specifications of auxiliary vectors, sets and functions associated with the depth-first search algorithm.

| Symbols/functions | Specifications |
|-------------------|---------------|
| $\mathbf{P}$      | The global row vector of auxiliary marks-predecessors used by DFS-algorithm |
|                   | $\mathbf{P}=|[p_v]|, p_v \in V(G) \cup \{\text{none}\}, v=1, ..., |V(G)|$ |
|                   | Each vertex $v \in V(G)$ is tagged by a mark $p_v$, which corresponds to the predecessor of vertex $v$ on the current path from vertex $v_o$ to vertex $v$. |
|                   | The start vertex $v_0$ has the following properties: $v_0 \in \phi(e_o)$, $v_0 \in \phi(e_o)$, $\phi(e_o)=(v_0,v_o)$, $\phi(e_o)=(v_0,v_o)$, $(e_o,e_o) \not\in E$. |
|                   | The first edge $e_0$ on the path is defined as follows: $\phi(e_0)=(v_0,v_o)$, $v_0 \in W_{trans}(v_0,v_o) \land v_0 \in W_{trans}(v_0,v_o)$, $p_v=v_o$. |
|                   | If the path to vertex $v \in V(G)$ does not exist, then $p_v=\text{none}$ (the symbol “none” expresses the nonexistence of the relevant predecessor regarding vertex $v_0$). |
| $\mathbf{D}$      | The global row vector of the auxiliary distance marks used by DFS-algorithm |
|                   | $\mathbf{D}=|[d_v]|, d_v \in \mathbb{R}^+, v=1, ..., |V(G)|$ |
|                   | Each vertex $v \in V(G)$ is tagged by a mark $d_v$ reflecting the length of the current path from the start vertex $v_o$ to vertex $v$ (computed by DFS-algorithm). |
|                   | If the path to vertex $v \in V(G)$ does not exist, then $d_v=\text{dist}^\infty$ (the symbol dist$^\infty$ expresses the “infinite length” of the path). |
| Stack             | Dynamic set of temporarily marked vertices used by DFS-algorithm |
|                   | Stack$\subset V(G)$ |
|                   | The linearly ordered dynamic set Stack applies the LIFO (Last In First Out) strategy with regard to inserting and removing elements. |
| Push($\downarrow$Stack,$iv_0$) | Auxiliary function inserting element-vertex $v_i$ into the set Stack |
| Pop($\uparrow$Stack,$iv_0$)  | Auxiliary function removing element-vertex $v_i$ from the set Stack |
| GetPath($\Upsilon$Path) | Delivery of the topology related to the path found (Path) based on reverse path browsing (using marks-predecessors) |

This is a sparse graph [13] where the relation between the number of its edges ($m$) and the number of its vertices ($n$) can be expressed by a linear function. For the infrastructure model described (graph $G$), the computations of Algorithm 1 do not spread massively to the width (using the basic concept of Dijkstra’s algorithm) because it is possible to continue in no more than 2 directions from each vertex attained. This can be illustrated on the demonstration track yard model shown in Fig. 2(d), for which we have $n=22$, $m=24$, as well as on the model of the real rail infrastructure (partially shown in Fig. 3), where $n=677$ and $m=710$. From the practical point of view, this means that the number of routes/walks (leading to the given relocation finish positions) that must be examined is considerably lower than in the case of dense (undirected) graphs. For such graphs, the relation between $m$ and $n$ can be expressed by a quadratic function [13].

For example, the maximum number of edges in a simple undirected graph is $n((n−1)/2)$. The undirected graph $G$ can also be characterized by using the indicator $d(G)$ called edge density [13]: $d(G)=(2m)/(n(n−1))$, $d(G)\in(0,1)$. This parameter mirrors the graph sparsity/density rate expressed by the ratio of the actual-to-maximum possible number of edges in graph $G$. A graph is cruelly classified as sparse if its $d(G)<1/2$, dense if its $d(G)>1/2$, and indifferent if its $d(G)=1/2$. The graph shown in Fig. 2(d) has $d(G)=0.103$ and the graph shown in Fig. 3 has $d(G)=0.003$.

- Graph $G$ was implemented by using an aggregate data structure composed of two hash tables, making their elements accessible based on keys. The one table stores the graph vertices, the other, the graph edges. Each element-vertex encapsulates an array of references to the appropriate incident edge (of which there can be up to 3). Each element-edge fixes 2 references to the relevant incident vertices. The references associated to the elements in the two tables can be used for efficient traversing between the graph vertices and edges. A Fibonacci heap [14] was used for the implementation of set $\mathcal{T}_V$, whereby very efficient implementation of the priority queue is attained. The priorities of its elements $(v_i,e_o,q)$, $v_i \in V(G)$, $q \in \mathbb{R}^+_{\geq 0}$, $e_o \in E(G)$, $v_i \in \phi(e_o)$, are determined by the $q$-component value (the element priority is the higher the lower the value of its distance mark $q$).

- The track infrastructure can be defined within a specific configuration file (in XML format), which uses templates...
A. Kavička, R. Diviš: Dynamic Search of Train Shortest Routes Within Microscopic Traffic Simulators

FIGURE 3. Part of the track layout related to the station Pardubice hl.n. and its surroundings.

inspired by the railML standard [35], [36]. The infrastructure models can be constructed (e.g., for the MesoRail simulator [4]), within the TrackEd simulator [37] for instance.

- For the Shortest_Walk function (performing calculation within Algorithm 1), the finish relocation position ([e_y, v_g] ∈ F) within the track yard model (graph G) is represented by track/edge e_y, reached through its particular end (vertex v_g). In reality, this means that once the relocation motion is over, object O will stand precisely at the respective end of the finish track. So, if object O is required to stand at a certain distance Δ from the above track/edge end (vertex v_g), Δ ≥ L ∧ Δ ≤ κ([e_y, v_g]), the total distance travelled by object O must be increased by Δ.

V. EXAMPLES OF PRIMARY ALGORITHM DEPLOYMENT

To illustrate the use of Algorithm 1, examples of its application when computing different routes within the track yard demonstration model are presented below.

A. EXAMPLE 1—TRAIN SHUNTING

Example 1 describes the computation of the route for train shunting between two selected tracks. Fig. 4 illustrates the position of relocation object (train) O_1 with a length L within the model (graph G) of the demonstration rail infrastructure (mirroring a realistic track yard). Object O_1 stands on track/edge e_5(at its end v_12) and is required to be relocated to track/edge e_4 through its end v_11. The following settings apply when computing the track route (by the Shortest_Walk routine):

\[ S = \{[e_5, v_{12}], F = [e_{4,11}]\}, \quad L = 120 \]

The weights and ingress vacancies for selected edges in graph G are as follows:

\[ \omega(e_4) = 280, \quad \kappa([e_4, v_{11}]) = \kappa([e_4, v_9]) = 280, \quad e_4 \in \text{E}_{\text{dest}}(G), \quad v_9, v_{11} \in \text{V}(G) \]

\[ \omega(e_5) = 259, \quad \kappa([e_5, v_{12}]) = 0, \quad \kappa([e_5, v_{10}]) = 139, \quad e_5 \in \text{E}_{\text{dest}}(G), \quad v_{10}, v_{12} \in \text{V}(G) \]

In one case, the shortest path was sought (by the Depth_First_Search subprogram) “behind” the track switch \((e_{19}, e_{19}) \in Z\). Table 6 lists the contents of the auxiliary vectors \( \mathbf{D} \) and \( \mathbf{P} \) (the symbols ‘–’ in the table substitute the dist∞ values or ‘none’ elements) after termination of the above subprogram, whereby the following path was found:

\[ \text{Partroute} = [[1, \text{in}, [e_9, v_{16}]], \quad [2, \text{out}, [e_{21}, v_{16}]], \quad [3, \text{in}, [e_{21}, v_{14}]], \quad [4, \text{out}, [e_7, v_{14}]], \quad [5, \text{in}, [e_7, v_{13}]]] \]

The contents of the auxiliary vectors M and D when the Shortest_Walk routine is over is illustrated in Table 7 and the resulting walk Seq is presented in Table 8. For graphic reasons, each element of the walk Seq is saved on one table column and the partial components of the various elements are saved in different rows.

As mentioned above, the topology of the walk Seq is computed by the Get_Walk function, and the computation development (related to object O_1) proceeds from its finish position to its start position. The process can be illustrated
as follows. The walk finish track/edge is $e_4$ and the finish vertex is $v_{11}$ (with the distance mark $d_{11} = 194$). Element $[e_4,v_{11}]$ is initially inserted into the walk Seq. Furthermore, it is found that $v_{11}$ was reached through edge $e_{18}$ (this information can be gained from $m_{11}$), hence, element $[e_{18},v_{11}]$ is inserted into the walk. The predecessor $v_p$ of vertex $v_{11}$ is identified first. Based on the property of the edge $e_{18}$ it can be determined from the appropriate data structure that $v_p \in \phi(e_{18}) \land v_p \neq v_{11}$, and hence, $v_p = v_{13}$ (element $[e_{18},v_{13}]$ is inserted into the walk Seq). The predecessor of vertex $v_{13}$ is identified in the next step. It holds for that vertex that $d_{13} = 40$ and it was reached through edge $e_{19}$ (as can be found from $m_{13}$). However, since it is indicated that $(e_{18},e_{19}) \in Z$ and $d_{11} = d_{13} + \omega(e_{18}) + L$ (194 = 40+34+120), the partial walk PartSeq must first be inserted into Seq (by the Insert_Reversal function).

The walk PartSeq is stepwise built up (and, at the same time, inserted into Seq) over the Partroute path topology by means of the Insert_Reversal function. The Partroute path is obtained by using the Download_Path($\downarrow v_{13}$, $\downarrow e_{19}$, $\downarrow e_{18}$, $\uparrow$Partroute) function. The Partroute path is traversed twice when building up the PartSeq walk. First it is passed via the ingress vertex $v_{13}$ (from edge/track $e_7$) as far as the edge/track $e_9$ (via $v_{16}$). Subsequently this path is traversed back.

Once the complete PartSeq has been inserted into Seq, the element $[e_{19},v_{13}]$ is added to the walk. Now, the predecessor of vertex $v_{13}$ (reached via edge $e_{19}$) can be determined, it is vertex $v_{12}$ (labelled with the mark $d_{12} = 0$).

For illustration, Table 9 contains the reduced distance matrix (RDM) (for $L=120$) containing the relocation lengths between all the relevant destination edge pairs in graph $G$. The RDM was calculated by Algorithm 1 (with differently defined sets $S$ and $F$). The computations were made assuming that except for object $O_1$ (each time standing on a different start vertex)
TABLE 7: Vectors D, M after computations of \textit{Shortest_Walk}((I,S,i,F,i,L,\text{Seq}), S=\{e_5,v_{12}\}, F=\{e_4,v_{11}\}, L=120.

| k | d_k | m_k |
|---|-----|-----|
| 1 | - | [-,e_1] |
| 2 | - | [-,e_2] |
| 3 | - | [-,e_3] | [-,e_{13}] |
| 4 | - | [-,e_2] | [-,e_{13}] | [-,e_4] |
| 6 | - | [-,e_3] | [-,e_4] |
| 7 | - | [-,e_3] | [-,e_{13}] | [-,e_4] |
| 8 | - | [-,e_3] | [-,e_{13}] | [-,e_4] |
| 9 | - | [-,e_3] | [-,e_4] |
| 10 | - | [-,e_3] | [-,e_{13}] |
| 11 | 194 | [-,e_4] | [194,e_{13}] |
| 12 | 0 | [0,e_5] | [-,e_{13}] |
| 13 | 40 | [-,e_1] | [-,e_{13}] | [40,e_4] |
| 14 | 80 | [80,e_1] | [-,e_{20}] | [-,e_{21}] |
| 15 | 118 | [-,e_3] | [118,e_{20}] |
| 16 | 121 | [-,e_3] | [121,e_{21}] |
| 17 | 423 | [423,e_4] | [-,e_{22}] |
| 18 | - | [-,e_4] | [-,e_{22}] |
| 19 | - | [-,e_1] | [-,e_{22}] | [-,e_{23}] |
| 20 | 257 | [257,e_4] | [-,e_{24}] | [-,e_{23}] |
| 21 | - | [-,e_3] | [-,e_{24}] |
| 22 | - | [-,e_3] | [-,e_{25}] |
| 23 | - | [-,e_3] | [-,e_{26}] |

objects: object/train O_2 100 m long stands on track/edge e_5; object O_1 - a regional multiple-unit train 50 m long – stands on track/edge e_{10}; and object/train O_3 50 m long stands on track/edge e_4. The task is to relocate object O_1 to track/edge e_4 via track/edge end v_9.

The following settings apply to the track route computation (by the \textit{Shortest_Walk} routine):

\[ S = ([e_{10}, v_{21}], F = [e_4, v_9]), L = 50 \]

The weights and ingress vacancies of certain edges in graph G are specified below:

\[
\begin{align*}
\omega(e_4) &= 280, \kappa([e_4, v_{11}]) = 0, \kappa([e_4, v_9]) \\
\omega(e_5) &= 260, e_4 \in E_{\text{dest}}(G), v_9, v_{11} \in V(G) \\
\omega(e_5) &= 259, \kappa([e_5, v_{12}]) = 159, \kappa([e_5, v_{10}]) \\
\omega(e_{10}) &= 236, \kappa([e_{10}, v_{21}]) = 0, \kappa([e_{10}, v_{23}]) \\
\omega(e_{10}) &= 186, e_{10} \in E_{\text{dest}}(G), v_{21}, v_{23} \in V(G)
\end{align*}
\]

The contents of the auxiliary vectors M and D when the \textit{Shortest_Walk} subprogram is over are listed in Table 10 and the resulting walk Seq is shown in Table 11 (this walk includes 2 reversals: v_4 \leftarrow e_2 \rightarrow v_4 and v_20 \leftarrow e_{20} \rightarrow v_20).

The topology of the ultimate track route is as follows:

\[ e_4(1752) \rightarrow v_9(1070) \rightarrow e_{16} \rightarrow v_{8}(1668) \rightarrow e_3 \rightarrow v_6(1648) \rightarrow e_4(1518) \rightarrow e_2 \rightarrow v_4(1518) \rightarrow e_{13} \rightarrow v_7(1433) \rightarrow e_6 \rightarrow v_{18}(706) \rightarrow e_{23} \rightarrow v_{19}(639) \rightarrow e_{25} \rightarrow v_{20} \rightarrow e_{20} \rightarrow v_{20}(33) \rightarrow e_{24} \rightarrow v_{21}(1) \rightarrow e_{10} \]

The total length of the trajectory travelled by object O_1 is 1752 m.

This example can be used to demonstrate the irreplaceable role of the distance marks from vector M in this situation where the distance marks from vector D are not sufficient (in contrast to the traditional Dijkstra’s algorithm). This case is associated with vertex v_19, the marking of which can be as follows during the Algorithm 1 computation:

\[
d_{19} = 639, m_{19} = ([666, e_{11}], [639, e_{22}], [\text{dist}^\infty, e_{23}])
\]

The following cases occurred during the re-marking of vertex v_{18} from the adjacent vertex v_{19} (via edge e_{23}), for which we have \( \omega(e_{23})=\kappa([e_{23}, v_{19}])=40 \):

(a) \( d_{18}=729, m_{18}=[\text{dist}^\infty, e_6], [729, e_{23}] \)
(b) \( d_{18}=706, m_{18}=[\text{dist}^\infty, e_6], [706, e_{23}] \)

In case (a), vertex v_{19} was attained via edge e_{22} and reversal after the track switch \( (e_{22}, e_{23}) \in Z \) had to be taken into account. This implies that the relocation length \( (L=50) \) of object O_1 beyond the switch had to be included. Vertex v_{18} \( \in W_{\text{trans}}(v_{19}, e_{22}) \), attained by edge e_{23}, is from the set of its adjacent reversal vertices. As a result of the reversal, the distance mark of vertex v_{18} was changed to 729=639+L+\omega(e_{23}). The re-markings could be made where the condition: currdist>729, [currdist,e_{23}] \in m_{18}, was met.

In case (b), vertex v_{19} was attained via edge e_{11}. Vertex v_{18} \( \in W_{\text{trans}}(v_{19}, e_{11}) \), attained via edge e_{23}, is an element of track), the infrastructure is completely empty. The RDM is obtained by simple reduction of the complete distance matrix (CDM). The reduction is based on the fact that the RDM does not include those CDM rows and columns all elements of which contain the dist^\infty value (indicating the non-existence of the walk specifying the object O_1 relocation).

B. EXAMPLE 2—SHUNTING LOCOMOTIVE RELOCATION

The example demonstrates the computation results for the relocation trajectory along which a shunting locomotive is moved to a specific end of a particular train set. Fig. 5 shows the model of the track yard, now occupied by 3 relocations
TABLE 8. The shortest walk topology (Seq) computed by Shortest_Walk, $S=\{e_5,v_{12}\}$, $F=\{e_4,v_{11}\}$, $L=120$.

| dir | in | out | in | out | in | out | in | out |
|-----|----|-----|----|-----|----|-----|----|-----|
| $[e,v]$ | $[e_5,v_{12}]$ | $[e_{10},v_{11}]$ | $[e_{14},v_{12}]$ | $[e_{17},v_{13}]$ | $[e_7,v_{14}]$ | $[e_{21},v_{15}]$ | $[e_{21},v_{16}]$ | $[e_9,v_{16}]$ |

TABLE 9. The reduced distance matrix (RDM) for the parameter $L=120$ and vacant track layout.

| Start | $[e_1,v_1]$ | $[e_2,v_2]$ | $[e_3,v_3]$ | $[e_4,v_4]$ | $[e_5,v_5]$ | $[e_6,v_6]$ | $[e_7,v_7]$ | $[e_8,v_8]$ | $[e_9,v_9]$ |
|-------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| $[e_1,v_1]$ | - | 398 | 652 | 1112 | 659 | 1126 | 193 | 1537 | 1036 | 1543 | 1039 | 1116 | 1208 | 960 | 1235 |
| $[e_2,v_2]$ | 398 | - | 254 | 714 | 261 | 728 | 205 | 1139 | 638 | 1555 | 641 | 1128 | 810 | 972 | 837 |
| $[e_3,v_3]$ | 652 | 254 | - | 768 | 315 | 1805 | 459 | 1193 | 692 | 1616 | 695 | 1382 | 864 | 1033 | 891 |
| $[e_4,v_4]$ | 1112 | 714 | 768 | - | 1806 | 314 | 919 | 733 | 232 | 1156 | 235 | 990 | 404 | 573 | 431 |
| $[e_5,v_5]$ | 659 | 261 | 315 | 1806 | - | 789 | 466 | 1208 | 707 | 1631 | 710 | 1389 | 879 | 1048 | 906 |
| $[e_6,v_6]$ | 1126 | 728 | 1805 | 314 | 789 | - | 933 | 739 | 238 | 1162 | 241 | 996 | 410 | 579 | 437 |
| $[e_7,v_7]$ | 193 | 205 | 459 | 919 | 466 | 933 | - | 1344 | 843 | 1767 | 846 | 1601 | 1015 | 1184 | 1042 |
| $[e_8,v_8]$ | 1537 | 1139 | 1193 | 733 | 1208 | 739 | 1344 | - | 817 | 743 | 1081 | 316 | 896 | 160 | 1016 |
| $[e_9,v_9]$ | 1036 | 638 | 692 | 232 | 707 | 238 | 843 | 817 | - | 1240 | 319 | 1766 | 1593 | 657 | - |
| $[e_{10},v_{10}]$ | 1543 | 1555 | 1616 | 1156 | 1631 | 1162 | 1767 | 743 | 1240 | - | 2196 | 739 | 153 | - | 180 |
| $[e_{11},v_{11}]$ | 1039 | 641 | 695 | 235 | 710 | 241 | 846 | 1081 | 319 | 2196 | - | 1077 | 491 | 1613 | 518 |
| $[e_{12},v_{12}]$ | 1116 | 1128 | 1382 | 990 | 1389 | 996 | 1601 | 316 | 1766 | 739 | 1077 | - | 892 | 156 | 1965 |
| $[e_{13},v_{13}]$ | 1208 | 810 | 864 | 404 | 879 | 410 | 1015 | 896 | 1393 | 153 | 491 | 892 | - | 829 | 333 |
| $[e_{14},v_{14}]$ | 960 | 972 | 1033 | 573 | 1048 | 579 | 1184 | 160 | 657 | - | 1613 | 156 | 829 | - | 856 |
| $[e_{15},v_{15}]$ | 1235 | 837 | 891 | 431 | 906 | 437 | 1042 | 1016 | - | 180 | 518 | 1965 | 333 | 856 | - |

the set of its adjacent transit vertices. Where the condition: currdist $>$ 706 ($666 + \omega(e_{23})$) was met and hence, the current value (currdist) of the mark ($[\text{currdist},e_{23}] \in \text{m}_{18}$) could be changed, this was performed.

From the above description it will be clear that the minimum (639) from the distance marks of vertex $v_{19}$ mirrors correctly the length of the shortest walk to this vertex via edge $e_{22}$ (starting from $[e_{10},v_{21}]$). Despite that, when seeking for the shortest walk to the adjacent vertex $v_{18}$ (with respect to vertex $v_{19}$), it is more advantageous to use another walk (starting from $[e_{10},v_{21}]$) to $v_{19}$ via edge $e_{11}$. The reason for this is that when vertex $v_{18}$ is ultimately re-marked (as the successor of vertex $v_{19}$ via edge $e_{23}$), the lowest of the values $\text{len}_{22} + L + \omega(e_{23})$ and $\text{len}_{11} + \omega(e_{23})$, where $[\text{len}_{11},e_{11}]$, $[\text{len}_{22},e_{22}] \in \text{m}_{19}$, will be relevant.

Two additional illustrative cases beyond this example are worth presenting:
If the whole track yard including track $e_5$ were unoccupied (except for the start track $e_{10}$) and the remaining parameters remained unchanged, the track route would be as follows:

$$
\begin{align*}
& e_4(724) \rightarrow v_9(674) \rightarrow e_{16} \rightarrow v_8 \rightarrow e_3 \rightarrow v_6 \rightarrow e_{14} \rightarrow v_6 \rightarrow e_3 \rightarrow v_4(590) \rightarrow e_{17} \rightarrow v_{10}(549) \rightarrow e_5 \rightarrow v_{12}(290) \rightarrow e_{19} \rightarrow v_{13}(250) \\
& \rightarrow e_7 \rightarrow v_{14}(210) \rightarrow e_{20} \rightarrow v_{15}(172) \rightarrow e_8 \rightarrow v_{20}(33) \rightarrow e_{24} \rightarrow v_{21}(0) \rightarrow e_{10}
\end{align*}
$$

In this case, it is possible to find a track route that is substantially shorter (724 m) than with the basic parametrization in Example 2 (where the route is 1752 m long).

| $k$ | $d_k$ | $m_k$ |
|-----|-------|-------|
| 1   | 1663  | $[1663,e_1]$ |
| 2   | 1668  | $[1668,e_1]$ |
| 3   | 1506  | $[-e_1]$ |
| 4   | 1518  | $[1518,e_1]$ |
| 5   | 1648  | $[-e_1]$ |
| 6   | 1433  | $[1433,e_1]$ |
| 7   | 1668  | $[1668,e_1]$ |
| 8   | 1702  | $[-e_1]$ |
| 9   | 1709  | $[-e_1]$ |
| 10  | 284   | $[-e_1]$ |
| 11  | 290   | $[-e_1]$ |
| 12  | 250   | $[250,e_1]$ |
| 13  | 210   | $[210,e_1]$ |
| 14  | 172   | $[172,e_1]$ |
| 15  | 301   | $[301,e_1]$ |
| 16  | 603   | $[603,e_1]$ |
| 17  | 706   | $[706,e_1]$ |
| 18  | 639   | $[639,e_1]$ |
| 19  | 33    | $[33,e_1]$ |
| 20  | 0     | $[0,e_1]$ |
| 21  | 143   | $[143,e_1]$ |
| 22  | 112   | $[112,e_1]$ |
| 23  | $[-e_1]$ |

In the parametrization in Example 2, no track route is found if the $L$ value is changed to $L=150$. Track/edge $e_4$ cannot be accessed via its end/vertex $v_9$ because a relocation object 150 m long cannot perform the required reversal on track/edge $e_2$ due to the inadequate vacancy $\omega(e_2)=100$.

### C. Example 3—Locomotive Serving One Train

This example demonstrates a case where a shunting locomotive is relocated from one train end to the other end. A train car set without a locomotive, 200 m total length, stands in the track yard the model of which is shown in Fig. 6. The car set is regarded as object $O_2$ standing on track/edge $e_5$. A shunting locomotive — object $O_1$ - 20 m long has just been uncoupled from the train to be coupled up to it at the opposite end $A$. For this, the locomotive must enter track/edge $e_5$ through its end $v_{10}$.

The following settings are considered for the track route computation (by the `Shortest_Walk` routine):

$$
S = [e_5, v_{12}], F = [e_5, v_{10}], L = 20, \omega(e_5) = 259, \kappa([e_5, v_{12}]) = 0, \kappa([e_5, v_{10}]) = 39, e_5 \in E_{\text{dest}}(G), v_{10}, v_{12} \in V(G)
$$

The contents of the auxiliary vectors $M$ and $D$ once the `Shortest_Walk` subprogram is over is listed in Table 12 and the resulting walk $Seq$, in Table 13 (this walk includes 2 reversals: $v_{13} \rightarrow e_7 \rightarrow v_{13}$ and $v_8 \rightarrow e_3 \rightarrow v_8$).

The track route topology is as follows:

$$
\begin{align*}
& e_5(508) \rightarrow v_{10}(469) \rightarrow e_{17} \rightarrow v_9 \rightarrow v_{13} \rightarrow v_8 \rightarrow e_3 \rightarrow v_9(408) \rightarrow e_{16} \rightarrow v_{10}(374) \rightarrow e_4 \rightarrow v_{11}(94) \rightarrow e_{18} \rightarrow v_{13} \rightarrow e_7 \rightarrow v_{13}(40) \rightarrow e_{19} \rightarrow v_{12}(0) \rightarrow e_5
\end{align*}
$$

The total length of the trajectory travelled by object $O_1$, including the distance travelled to reach the end $A$ of object $O_2$, is 508 m.

### VI. Extensions of the Primary Algorithm

Algorithm 1 is designed to search for the shortest route in the model of a currently occupied track yard. The route starts from one end of the start track (where object $O$ is present) and leads to one defined end of the finish track (to which object $O$ is to be relocated). This can be referred to as the single-source single-destination shortest walk (that is, $|S|=|F|=1$).

If no requirement exists as to the end through which object $O$ should leave the start track and/or the end through which the object should access the finish track, then Algorithm 1 must be modified: set $S$ and/or set $F$ ($|S|\in\{1,2\}$, $|F|\in\{1,2\}$) must be specified differently. This is so because it can be irrelevant through which end the start track is left and/or through which end the finish track is accessed. The combinations of the differently specified sets $S$ and $F$ that are associated with the different modifications of Algorithm 1 are listed in Table 14.

Algorithms 2-4 differ from the primary Algorithm 1 only by partial changes in the implementation of the two functions `Start_Finish_Test` and `Start_Finish_Init`, the remaining parts of the algorithms are the same as in Algorithm 1. The
TABLE 11. The shortest walk topology (Seq) computed by Shortest_Walk, $S=\{e_{10},v_{21}\}$, $F=\{e_{4},v_{9}\}$, $L=50$.

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| dir | in | out | in | out | in | out | in | out |
| $[e,v]$ | $[e_{10},v_{21}]$ | $[e_{10},v_{13}]$ | $[e_{10},v_{14}]$ | $[v_{9},v_{14}]$ | $[v_{9},v_{13}]$ | $[v_{9},v_{12}]$ | $[v_{9},v_{11}]$ | $[v_{9},v_{10}]$ |

| i | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|
| dir | in | out | in | out | in | out | in | out |
| $[e,v]$ | $[e_{9},v_{4}]$ | $[e_{9},v_{3}]$ | $[e_{9},v_{2}]$ | $[v_{9},v_{11}]$ | $[v_{9},v_{10}]$ | $[v_{9},v_{9}]$ | $[v_{9},v_{8}]$ | $[v_{9},v_{7}]$ |

| i | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|
| dir | in | out | in | out | in | out | in | out |
| $[e,v]$ | $[e_{9},v_{22}]$ | $[e_{9},v_{21}]$ | $[e_{9},v_{20}]$ | $[v_{9},v_{20}]$ | $[v_{9},v_{20}]$ | $[v_{9},v_{20}]$ | $[v_{9},v_{20}]$ | $[v_{9},v_{20}]$ |

routines test by various methods the admissibility of combinations of the input parameters and perform the initialization operations regarding the relocation start and finish elements.

*Algorithm 2* searches for the shortest admissible walk between two two-member sets $S$ and $F$ ($|S|=|F|=2$), that is, a *two-sources two-destinations shortest walk*. From the practical point of view, this means that the algorithm starts its propagation in 2 directions from the side vertices of the same edge (mirroring the opposite ends of that track on which the relocation object $O$ stands). Hence, initialization includes a procedure during which two elements representing the two possible starting positions (two sources) of the route are inserted into set $T_{V}$ (temporarily marked vertices). If the walk was found, the algorithm was terminated under the condition that one of the finish vertices was withdrawn from the set $T_{V}$. The use of *Algorithm 2* is illustrated in *Example 4*. 
TABLE 12. Vectors D, M after computations of Algorithm 1, \( S = [e_5, v_{12}], F = [e_5, v_{16}], L = 20. \)

| \( k \) | \( d_k \) | \( m_k \) |
|---|---|---|
| 1 | \(-e_1\) | |
| 2 | \(-e_6\) | |
| 3 | \(-e_1\) | \(-e_{12}\) |
| 4 | 508 | \(-e_2\) | \(-e_{13}\) | \([508, e_{11}]\) |
| 6 | 428 | \([428, e_3]\) | \(-e_4\) |
| 7 | \(-e_6\) | \(-e_{12}\) | \(-e_{13}\) |
| 8 | 408 | \([408, e_{16}]\) | \(-e_{17}\) |
| 9 | 374 | \([374, e_4]\) | \(-e_{16}\) |
| 10 | 469 | \(-e_6\) | \([469, e_7]\) |
| 11 | 94 | \(-e_4\) | \([94, e_8]\) |
| 12 | 0 | \([0, e_6]\) | \(-e_{18}\) |
| 13 | 40 | \(-e_3\) | \(-e_{19}\) | \([40, e_{19}]\) |
| 14 | 80 | \([80, e_3]\) | \(-e_{20}\) | \(-e_{21}\) |
| 15 | 118 | \(-e_2\) | \([118, e_{20}]\) |
| 16 | 121 | \(-e_3\) | \([121, e_{21}]\) |
| 17 | 423 | \([423, e_4]\) | \(-e_{22}\) |
| 18 | 519 | \(-e_6\) | \([519, e_{23}]\) |
| 19 | 459 | \([459, e_{11}]\) | \([459, e_{22}]\) | \(-e_{23}\) |
| 20 | 257 | \([257, e_6]\) | \(-e_{24}\) | \(-e_{25}\) |
| 21 | 290 | \(-e_3\) | \([290, e_{25}]\) |
| 22 | 317 | \([982, e_{11}]\) | \([317, e_{25}]\) |
| 23 | 526 | \([526, e_{11}]\) | |

A. EXAMPLE 4—TWO-SOURCES TWO-DESTINATIONS SEARCH

A locomotive (object \( O_1 \) whose length is \( L = 20 \) m) stands on track/edge \( e_6 \) of the track yard shown in Fig. 7. The track length and the position of object \( O_1 \) on the track are characterized as follows:

\[
\omega(e_6) = 727, \kappa([e_6, v_{18}]) = 207, \kappa([e_6, v_7]) = 500
\]

The object should be relocated by the shortest route to track/edge \( e_4 \). The object \( (O_1) \) may be stopped at that track end through which the track was accessed.

So, the parametrizations of Algorithm 2 (or the Shortest_Walk function) is as follows:

\[
S = \{ [e_6, v_{18}], [e_6, v_7] \}, F = \{ [e_4, v_9], [e_4, v_{11}] \}, L = 20
\]

The contents of the auxiliary vectors \( M \) and \( D \) once the Shortest_Walk routine is over is listed in Table 15 and the resulting walk Seq is shown in Table 16 (this walk includes one reversal: \( v_{19} \leftarrow e_{11} \leftarrow v_{19} \)).

Algorithm 3 applies the two-sources single-destination strategy. The potential result obtained with this algorithm is the shorter of the two walks starting from the specific start positions (the two side vertices of the start edge) and ending in one specified finish position.

The last modification of the primary algorithm is Algorithm 4 which makes the single-source two-destinations type search, that is, searches for the shorter of two walks starting from one start position and leading to two defined finish positions (the two opposite ends of the finish track).

VII. VERIFICATION AND VALIDATION

The track infrastructure models and shortest route searching algorithms described in the previous sections were implemented within the MesoRail simulator [4]. While initially designed for mesoscopic rail traffic simulations, this simulator was extended recently to be applicable to microscopic simulations as well. Several case studies mirroring different parts of the Czech railway network infrastructure were used for testing. Each case study included the track infrastructure of both selected railway lines (or their parts) and a selected relevant railway station. To illustrate the time demands of the track route searching algorithms, the next parts of this paper use the results of a case study dealing with the application of the algorithms to a model of the infrastructure of the Pardubice Main Train Station (Pardubice hl.n.) and the adjacent railway network (the total length of this track infrastructure was: 71,514 m).

Verification and validation (of the infrastructure models and track route searching algorithms) followed the same procedure as described in detail in ref. [1]. The procedure is based on the approaches published in refs. [38], [39].

First, attention was paid to the conceptual model, containing 2 components:

(i) An undirected edge-weighted graph (mathematical model reflecting the topological situation and associated lengths of the track infrastructure elements).
TABLE 13. The shortest walk topology (Seq) computed by Algorithm 1, S=[e₅,v₁₂], F=[e₅,v₁₅], L=20.

| i   | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8   |
|-----|----|----|----|----|----|----|----|-----|
| dir |    | in | out|    | in | out|    |     |
| [e,v]|   | [e₅,v₂] | [e₅,v₁₂] | [e₁₇,v₈] | [e₇,v₆] | [e₁₇,v₆] | [e₇,v₆] | [e₇,v₆] | [e₅,v₂] |

TABLE 14. Overview of algorithm variants calculating the shortest track routes.

| Algorithm | Start vertices, S⊆I(G) | Finish vertices, F⊆I(G) | Search type |
|-----------|-------------------------|-------------------------|-------------|
| Algorithm 1 | S=[{e₅,v₅}] | F=[{e₅,v₅}] | single-source single-destination |
| Algorithm 2 | S=[{e₅,v₅},{e₅,v₈}] | F=[{e₅,v₅},{e₅,v₈}] | two-sources two-destinations |
| Algorithm 3 | S=[{e₅,v₅},{e₅,v₈}] | F=[{e₅,v₅}] | two-sources single-destination |
| Algorithm 4 | S=[{e₅,v₅}] | F=[{e₅,v₅},{e₅,v₈}] | single-source two-destinations |

(ii) *Graph algorithms* computing the shortest admissible walks reflecting the routes along which the relocation objects can be moved in the track yard. Such algorithms build on the basic concept of Dijkstra’s algorithm, which had to be substantially modified and completed.

*Conceptual model validation* focused on the appropriateness of both the mathematical model describing the track infrastructure and selected algorithms working over the mathematical model. The approach referred to as IV&V (Independent Verification & Validation) [39] was used: this is based on an expert assessment by a professional in the respective application domain. Specifically, a renowned independent railway traffic expert – employee of the railway transport company ČD – Informační systémy a.s. (Czech Railways – Information Systems) was hired for this assignment. This expert used the *Face Validation (Expert Validation)* method [39], requiring in-depth knowledge/expertise in the rail traffic system domain. The following conclusions arose from the conceptual model validation: (i) the mathematical model of the track infrastructure reflects the topological properties and lengths of the actual track elements adequately faithfully; and (ii) the algorithms for the computation of the rail vehicle relocation routes (and working over the given mathematical model) are conceptually well-designed to provide results matching the relevant solutions in practice (i.e., the realistic topologies of the routes in the specifically occupied track yard).
A computerized model was subsequently set up. Model implementation included an appropriate memory representation of the track yard model, briefly described at the Section IV.B. This memory representation (data structure) was designed so that the shortest walk-searching algorithms should make the computations efficiently. Verification of this

Algorithm 2 Computation of the shortest walk, $S=[e_x,v_a],[e_x,v_b], F=[e_y,v_g],[e_y,v_h]$

```
01  function Start_Finish_Test($\downarrow S, \downarrow F, \downarrow L, \uparrow \text{okay})$
02    if $S=\emptyset$ or $F=\emptyset$ or $e_x=e_y$ then // $[e_x, v_a], [e_x, v_b] \in S$, $[e_y, v_g], [e_y, v_h] \in F$
03      okay ← false
04      exit
05   end
06   for each $[e_y, v_j] \in F$ do
07     if $1019  \downarrow S, \downarrow L$ then
08       $F ← F - \{[e_y, v_j]\}$
09     end
10   end
11   if $F=\emptyset$ then
12      okay ← false
13      exit
14   end
15 end
16  function Start_Finish_Init($\downarrow S$, $\downarrow F$)
17    for each $[e_x, v_i] \in S$ do
18      Get_Index($\downarrow v_i, \uparrow i$)
19      $d_i ← \kappa([e_x, v_i])$ // initialisation of selected distance marks
20      Set_Mark($\downarrow m_i, \downarrow e_x, \downarrow d_i$)
21      $V_T ← V_T \cup \{v_i, e_x, d_i\}$ // initial insertion of vertices into the set $V_T$
22    end
23 end
```

Algorithm 3 Computation of the shortest walk, $S=[e_x,v_a],[e_x,v_b], F=[e_y,v_g]$

```
01  function Start_Finish_Test($\downarrow S, \downarrow F, \downarrow L, \uparrow \text{okay})$
02    if $S=\emptyset$ or $F=\emptyset$ then
03      okay ← false
04      exit
05   end
06   if $e_x = e_y$ or $\kappa([e_y, v_g]) < L$ then // $[e_x, v_a], [e_x, v_b] \in S$, $[e_y, v_g] \in F$
07      okay ← false // inadmissible combination of the start and finish edges-vertices
08      exit
09   end
10 end
11  function Start_Finish_Init($\downarrow S$, $\downarrow F$)
12    for each $[e_x, v_i] \in S$ do
13      Get_Index($\downarrow v_i, \uparrow i$)
14      $d_i ← \kappa([e_x, v_i])$ // initialisation of selected distance marks
15      Set_Mark($\downarrow m_i, \downarrow e_x, \downarrow d_i$)
16      $V_T ← V_T \cup \{v_i, e_x, d_i\}$ // initial insertion of vertices into the set $V_T$
17    end
18   if $\kappa([e_y, v_g]) = \omega(e_y)$ then // $[e_y, v_g] \in F$, $\varphi(e_y) = (v_g, v_h)$
19     $\kappa([e_y, v_h]) ← \omega(e_y) - \varepsilon$ // $\varepsilon \in R_0^+$, negligible small value compared to the weights of edges
20   end
21 end
```
Algorithm 4 Computation of the shortest walk, $S=\{e_x,v_a\}$, $F=\{e_y,v_g\},[e_y,v_h]$:

```plaintext
function Start_Finish_Test($\downarrow S, \downarrow F, \downarrow L, \uparrow okay$)
  if $S=\emptyset$ or $F=\emptyset$ or $e_x=e_y$ then // $[e_x,v_a]\in S, [e_y,v_g], [e_y,v_h]\in F$
    okay ← false
  exit
end
for each $[e_y,v_j]\in F$
  if $\kappa([e_y,v_j])<L$
    $F ← F−[e_y,v_j]$
  end
end
if $F=\emptyset$ then
  okay ← false
  exit
end
function Start_Finish_Init($\downarrow S, \downarrow F$)
  Get_Index($\downarrow v_a, \uparrow a$) // $[e_x,v_a]\in S$
  $d_a ← \kappa([e_x,v_a])$ // initialisation of selected distance marks
  Set_Mark($\downarrow m_a, \downarrow e_x, \downarrow d_a$)
  $T_V ← T_V∪[v_a,e_x,d_a]$
end
```

model included checks of the logical correctness of the results provided by the algorithms, which should correspond with the shortest track routes for the relocation of objects having defined lengths. The focus was specifically on:

- **Topological correctness** of the routes found, especially with respect to the reversals of the specifically long relocation objects.
- **Accounting for the track occupancies** (by rail vehicles) or their blocking by the interlocking system.

The verification procedure was performed by the authors of this paper, who found the computation results obtained with the relevant algorithms in the case studies to be logically correct. Some wrong results also occurred, of course, and were corrected by modifying the algorithm implementations accordingly.

The process of checking the suitability of the computerized model was concluded by its operational validation, made by the above Czech Railways company expert using the IV&V approach and the Face Validation method. In the various case studies, the traffic and technical admissibility of selected representative routes (delivered by the algorithms being tested) was examined for the specifically occupied track infrastructure and the specific relocations of the differently long rail vehicles. As a result, the solutions produced by the algorithms were classed as correct, both from the traffic aspect and from the technical aspect. Taking the validation results into account, the algorithms were integrated into the MesoRail simulation tool and are now routinely used.

![Demonstration model of track infrastructure where the degree of each vertex is 3.](image)

**FIGURE 8.** Demonstration model of track infrastructure where the degree of each vertex is 3.

**VIII. COMPUTATIONAL COMPLEXITY AND PRACTICAL USE OF THE ALGORITHMS**

This section analyses (formalizes) the computational complexity of the algorithms introduced above (regarding the implementations) and includes comments on the practical use of the algorithms for rail traffic simulations.

**A. COMPUTATIONAL COMPLEXITY OF THE ALGORITHMS**

The term computational complexity refers herein to the upper bound for the asymptotic computational complexity of an algorithm, which can be formalized by means of the big-$O$ notation [14]. To express the total computational complexity of the algorithms described in the preceding sections (Algorithms 1-4), each algorithm must be first decomposed...
into parts that will be independently analyzed (from the computational complexity aspect).

As mentioned in the previous sections, Algorithms 1-4 use (i) a modified concept of Dijkstra’s algorithm (applied to the calculation of the shortest path between two graph vertices) and (ii) the concept of the depth-first search (DFS) algorithm. The two concepts are used in independent parts of the programs and hence, can be analysed separately. The total computational complexity of the algorithms is then obtained by subsequent aggregation of the results of that analysis.

Each computation of the DFS-algorithm is used to determine the topology of the first free path found (in the undirected graph G specified in Tables 1-2), which serves to locate an object (with a length L) on the modelled rail yard infrastructure “behind” the track switch before reversing the direction of its movement. The complexity of one computation run of this algorithm can be expressed as $O(m + n)$, $n=|V(G)|$, $m=|E(G)|$ [14]. The maximum possible number of reversals computed on graph G (during the computation of each of the Algorithms 1-4) is $|I| \leq \lceil m / 2 \rceil$. This means that the DFS-algorithm can be computed no more than $|m / 2|$ times on a maximum of $m$ edges of graph G (this is the upper bound of the number of edges passed through during a computation). Hence, the total computational complexity associated with the repeated use of the DFS-algorithm can be expressed as $O((m / 2) \cdot (m + n)) = O((1/2) m^2 + (1/2) m n)$. If the upper bound of the number of edges in graph G is considered to be $m=\lceil 3 / 2 \rceil n$, then the complexity can be written: $O((15 / 8) n^2)$. A simple illustrative example of a graph G for which $m=\lceil 3 / 2 \rceil n$ is shown in Fig. 8.

Computations of the modified Dijkstra’s algorithm are also performed over the undirected graph G (Table 1). The standard expression of the computational complexity of the original Dijkstra’s algorithm, however, is associated with its application over an edge-weighted directed graph $G$, for which we have: $G=(V, E, \varphi, o, \omega; \varphi: E(G) \rightarrow [V, V] \times [V, V] \times [V, V] \times [V, V] \times [V, V] \times [V, V])$, $\omega: E(G) \rightarrow \mathbb{R}^+$. If this algorithm uses a Fibonacci heap for the implementation of the set of temporarily marked vertices, then it exhibits the complexity: $O(\log(n + m))$ [14]. Each vertex $V_k \in V(G)$ has available just one mark $q_k$ which, during the computation, reflects the current length of the current shortest path from the starting vertex $V_s \in V(G)$ to vertex $V_k$. During the computation of the original Dijkstra’s algorithm, each oriented edge is passed through/processed once at the most.

When the modified Dijkstra’s algorithm concept is used within Algorithms 1-4, the changes against the original Dijkstra’s algorithm must be analysed when examining the computational complexity. This practically means that the computations performed on the basic undirected graph G can be illustrated as computations performed over a virtual “transformed” directed graph $G$, for which the parameters $m=|V(G)|$, $m=|E(G)|$ must be primarily determined and compared to the parameters $n=|V(G)|$, $m=|E(G)|$ of the initial graph G.

- Each vertex $V_k \in V(G)$ has available a maximum of three marks: $\{len, e_j, e_3\} \in m_k$ ($len \in \mathbb{R}^+$, $e_j \in Y(V_k)$, $j=1, \ldots, \deg(V_k)$, $\deg(V_k) \in 1,2,3$); the mark values mirror the current lengths of the shortest walks leading to vertex $V_k$ through its different incident edges during the computation. In the context of the storage and potential removal (through the Extract_Min function [14]) of each vertex from set $T_V$ (implemented by using the Fibonacci heap), the upper bound of the number of vertices in the virtual graph G can be determined as $m=3n$.

- Each edge $e_3 \in E(G)$ can be passed through/processed four times as a maximum (this case is illustrated in Fig. 9 on edge $e_3$ as part of the infrastructure model fragment shown. So, with respect to the re-marking (by means of the

### Table 15. Vectors $D, M$ after computations of Algorithm 2, $S=[e_6,v_{18}], [e_7,v_{17}], [e_8,v_{13}], [e_4,v_{11}]$. $L=20$

| $k$ | $dk$ | $mk$ |
|-----|------|------|
| 1   | 730  | [730, $e_1$] |
| 2   | 735  | [735, $e_1$] |
| 3   | 573  | [573, $e_1$, $e_2$] |
| 4   | 585  | [585, $e_3$, $e_4$] |
| 6   | 685  | [685, $e_1$, $e_3$] |
| 7   | 500  | [500, $e_4$, $e_9$] |
| 8   | 705  | [705, $e_6$, $e_7$] |
| 9   | 739  | [739, $e_4$, $e_6$] |
| 10  | 746  | [746, $e_1$, $e_7$] |
| 11  | 720  | [720, $e_3$, $e_8$] |
| 12  | 726  | [726, $e_1$, $e_9$] |
| 13  | 686  | [686, $e_3$, $e_4$] |
| 14  | 646  | [646, $e_1$, $e_3$] |
| 15  | 704  | [704, $e_8$, $e_9$] |
| 16  | 605  | [605, $e_3$, $e_9$] |
| 17  | 303  | [303, $e_7$, $e_8$] |
| 18  | 207  | [207, $e_4$, $e_8$] |
| 19  | 247  | [247, $e_{11}$] |
| 20  | 843  | [843, $e_4$, $e_3$] |
| 21  | 770  | [770, $e_1$, $e_3$, $e_5$] |
| 22  | -    | [770, $e_1$, $e_3$, $e_5$] |
| 23  | -    | [770, $e_1$, $e_3$, $e_5$] |
TABLE 16. The shortest walk topology (Seq) from Algorithm 2, \( S=[e_6, v_{18}], F=[e_4, v_9], L=20 \).

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| dir | in | out | in | out | in | out | in | out |
| \([e_6, v_{18}]\) | \([e_6, v_1]\) | \([e_{18}, v_{13}]\) | \([e_7, v_{13}]\) | \([e_{13}, v_{14}]\) | \([e_{23}, v_{14}]\) | \([e_{23}, v_{18}]\) | \([e_6, v_{18}]\) |

| i | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|
| dir | in | out | in | out | in | out | in | out |
| \([e_6, v_{18}]\) | \([e_6, v_{13}]\) | \([e_{13}, v_{13}]\) | \([e_{15}, v_{13}]\) | \([e_{15}, v_{13}]\) | \([e_{23}, v_{18}]\) | \([e_6, v_{18}]\) |

**FIGURE 9.** Illustration of a quadruple processing of the edge \( e_3 \).

*Decrease Key* function \([14]\)) of each vertex in set \( T_V \) (implemented by using the Fibonacci heap), the upper bound of the number of edges in the virtual graph \( G \) is \( m=4 \) m. If the computation were to be performed in the virtual graph \( G \),
TABLE 17. Mean calculation times of the Algorithm 1 related to searching a shortest walk.

| ALGORITHM 1 | | | | | |
|-------------|-------------|-----------|-------------|-------------|
|             | | [V(G)] | [E(G)] | L | Calculated walks | max | max |
|             | | [-] | [-] | [-] | | [ms] | [ms] |
| Demonstration railway yard | | 22 | 24 | | | | |
| total length of tracks | | 3 418 m | | | | | |
| Pardubice main station incl. adjacent railway network | | 677 | 710 | | | | |
| total length of tracks | | 71 514 m | | | | | |

measures would have to be made to ensure meeting of the condition that an oriented edge can be passed through no more than once (as in the original Dijkstra’s algorithm). So, the computational complexity of the modified Dijkstra’s algorithm within the computations of Algorithms 1-4 can be expressed as $O(\log n + m n) = O(3 n \log 3 n + m n)$. If the upper bound of the number of edges is $m = (3/2) n$, then the computational complexity can be alternatively expressed as $O(3 n \log 3 n + 6 n)$.

Fig. 9 shows the sample computation of Algorithm 1 seeking for the shortest route for the relocation of a train which is 200 m long. Among the 4 different treatments of edge $e_3$ (during the different computation phases), one of the incident vertices of $e_3$ is re-marked in 3 cases (Figs. 9(c), 9(d) and 9(e)). In one case the condition for vertex ($v_5$) re-marking is not satisfied and so that vertex is only examined (Fig. 9(f)).

Two of the 4 cases – Figs. 9(c) and 9(e) – mirror the situation where the potential passage through the relevant edge ($e_3$) is not immediately preceded by train movement reversal, whereas the other 2 cases – Figs. 9(d) and 9(f) – describe the potential start of the train movement through $e_3$ immediately following a reversal.

The total computational complexity of each of the Algorithms 1-4 can be expressed by combining the partial computational complexities of the modified Dijkstra’s algorithm and the DFS-algorithm:

$$O(\frac{15}{8} n^2 + 3 n \log 3 n + 6 n).$$  \hspace{1em} \text{(1)}$$

**B. COMPUTATION TIME DEMANDS**

For using the algorithms within simulation experiments, it is useful to know how long each program will actually run in real time. This is important for practical reasons: simulations should not call too many time-consuming routines or else they are unusable. Examples of the mean times (and maximum times) taken by Algorithms 1-4 for two demonstration track infrastructures are included in Tables 17-20. The first infrastructure describes the demonstration track yard shown schematically in Fig. 2 (a), the second infrastructure refers to the track yard of Pardubice hl.n. and some parts of the adjacent rail network (a part of this infrastructure is schematically shown in Fig. 3). Pardubice hl.n. represents a medium-sized passenger railway station. There are 2 double-track lines and 1 single-track line adjacent to the station. These lines lead to neighboring railway stations, whose models are also part of the investigated simulation system. The infrastructure model also includes 97 single switches, 3 track crossings and 15 transit station tracks.

The second infrastructure model (including Pardubice hl.n.) – an edge-weighted undirected graph – encompasses
TABLE 18. Mean calculation times of the Algorithm 2 related to searching a shortest walk.

| ALGORITHM 2               | |   | |   | |   | |   | |   | |   |
|---------------------------|---|---|---|---|---|---|---|---|---|---|---|---|
| [V(G)]                    | |   | |   | |   | |   | |   | |   | |   |
| [E(G)]                    | |   | |   | |   | |   | |   | |   | |   |
| L                         | |   | |   | |   | |   | |   | |   | |   |
| Calculated walks          | |   | |   | |   | |   | |   | |   | |   |
| [ms]                      | |   | |   | |   | |   | |   | |   | |   |
| [ms]                      | |   | |   | |   | |   | |   | |   | |   |
| 677 vertices and 710 edges and its total track length is 71 514 m. Different relocation object lengths were used in the computation experiments: L=0, 20, 50, 100, 200, 300, 400 and 500.

The shortest walk was computed for all the admissible start and finish track pairs (the latter represented by destination edges in the model) regarding the track yard as empty. The condition was imposed that the start and finish track/edge weights are no lower than the L value. From the results it can be concluded that the mean computation time ($\text{mean}_t$) of a track route (represented by the shortest admissible walk in the infrastructure model) did not exceed 2 milliseconds in a rather extensive demonstration case study. Computations through which no shortest walk was found because none exists are also included. From the point of view of rail traffic simulations, the above mean computation time can be considered very favourable because it does not stress the system significantly more than other computation routines involved in simulations. The computations were made on a PC equipped with an Intel i7-8550U CPU@1.80 GHz processor, 16.0 GB RAM.

C. COMBINATION OF STATIC AND DYNAMIC COMPUTATIONS

Static and dynamic computations can be combined during the rail traffic simulations aimed at finding the optimum track routes for the train riding and shunting. Static computations are made by the algorithms before starting the simulation experiments and they may result in a quite extensive base of frequently used alternative routes for train relocations. The routes in the base are computed for the conditions of a free track infrastructure and preference can be given to no-reversal train relocation routes. Dynamic computations are made during the simulation experiment and are largely used to identify routes for rolling stock shunting and some train running routes. Such combinations can bring about reduction in the computation demands of the simulation program runs because the number of dynamic computations of the (frequently repeating) track routes is then lower.

IX. ADDITIONAL POTENTIAL ALGORITHM MODIFICATIONS

Additional modifications and/or extensions of the algorithms presented are feasible. Some are introduced below.

A. ROUTE LENGTH LIMITATION

Very long track routes between the given start position and the desired finish position are seldom set in real rail traffic. If a rail vehicle needs to be relocated at a long distance, the approach typically consists in consecutive setting of several shorter routes. This strategy can be inspirational...
TABLE 19. Mean calculation times of the Algorithm 3 related to searching a shortest walk.

| ALGORITHM 3 | | L | Calculated | max† | max‡ |
|-------------|-------------|-----------|---------|------|------|
|             | [V(G)]     | [E(G)]    |        |       |      |
|             | [\text{-}] | [\text{-}] | [\text{-}] |       |      |
| Demonstration railway yard | 22 | 24 | 0 | 190 | 0.012 | 0.087 |
| total length of tracks | | | 20 | 190 | 0.012 | 0.035 |
| 3 418 m | | | 50 | 120 | 0.012 | 0.039 |
| Pardubice main station incl. adjacent railway network | 677 | 710 | 100 | 120 | 0.012 | 0.040 |
| total length of tracks: | | | 200 | 50 | 0.013 | 0.031 |
| 71 514 m | | | 300 | 10 | 0.011 | 0.031 |
| | | | 400 | 3 | 0.015 | 0.020 |
| | | | 500 | 3 | 0.016 | 0.021 |

for the specification of the requirements for rail vehicle relocation in infrastructure models that are parts of railway traffic simulators. This means in practice that the potential requirements for finding a long route in a simulation trial can be divided into multiple requirements for searching several shorter routes. The partial routes lead consecutively to partial finishes, the last being identical with the desired finish of the complete (long) relocation. For this, a next parameter might be introduced in the route searching algorithms to specify the maximum admissible route length ($\text{dist}^{\text{max}}$) that is still acceptable as the algorithm computation result: no longer routes would be sought and the computation procedure would be terminated. So, if no route is found, the relocation object must temporarily stay in its current position. The requirement for finding a route is repeated after any traffic change (e.g., after another object leaves a track).

For the computations, the condition under which the distance mark of a vertex $v \in \text{V}(G)$ (i.e., currdist) can be changed to a new value ($\text{newdist} \in \mathbb{R}_{\geq 0}^\ell$) would have to be extended. This extension of the condition (which is included in rows 83 and 98 of Algorithms 1-4) is as follows: $\text{newdist} < \text{currdist} \wedge \text{newdist} \leq \text{dist}^{\text{max}}$. If this condition is not met, vertex $v \in \text{V}(G)$ is not re-marked or inserted into set $T_v$.

B. RESTRICTIONS ON REVERSALS

If the route is required to contain no reversals, the algorithms can be simply modified as follows. The code in rows 71-75 is removed (with no substitution) from the $\text{Adj Mark}$ function. This means that if the neighbours of the current vertex $v_c$ are examined, vertices from the set of its reverse adjacent vertices are omitted (and are not re-marked). So, the algorithms will identify only such routes (e.g., for the relocation of trains) as do not require the relocation object’s direction of motion to be reversed during the relocation operation.

In another approach, routes involving reversal are not admissible, but priority is given to reversal-free routes. This can be achieved, for instance, by imposing an expert penalty on each reversal included in the route: for instance, the length parameter of each reversal can be increased by a penalization constant (penconst). As a result, routes with reversals will be selected only if reversal-free routes either do not exist or are too long. The way to do this consists in a minor modification in row 97 of the $\text{Try Change Reverse}$ function, where the new distance-mark value ($\text{newdist}$) is calculated as $\text{newdist} = \text{newdist} - (q + \omega(e_s) + L + \text{penconst})$. It must be borne in mind, however, that in this approach, the vertex distance marks will not contain the true relocation distances anymore: instead, the distance will potentially be increased by the penalty for
reversals. This fact must be accounted for when imposing any route length limitations.

C. SIMPLIFIED PROCEDURE TO GET THE ROUTE TOPOLOGY

The computation aimed at gaining the shortest route topology (by the Get_Walk function) can be simplified by expanding the state space by adding another dynamic component (edge $P_{ej}$) from vector $M$. Edge $P_{ej}$ is the edge-predecessor of edge $ej$ on the walk being sought. Edges $ej$ and $P_{ej}$ are incidental with the same vertex $v_i$. So, the elements $m_k$ are defined as follows:

$$m_k = \{[\text{len}_j, ej, P_{ej}], \text{len}_j \in \mathbb{R}_+, ej, P_{ej} \in Y(v_i), v_k, v_l \in V(G) : \deg(v_k) = 1, \ldots, |V(G)|\}$$

Hence, the implementation of the Get_Walk function can be modified (simplified) for reverse browsing of the walk found. In this modification, it is possible to consecutively reversely traverse directly over the predecessors of each vertex and edge visited while the Insert_Reversal function is used in the unchanged form.

For this extension, the transfer and setting of the $P_{ej}$ components during the computation must be treated. Another parameter must be added to the Set_Mark function for the input of edge $P_{ej}$. The function calls must be modified in rows: 51 (input value: none)

D. HEAD-OF-TRAIN MONITORING

If the train in the finish relocation position is required to be oriented specifically in one of the two possible directions, steps must be made to follow either the head-of-train ($heot$) or the end-of-train ($eot$) during the computation. For example, Algorithm 3 can be modified into Algorithm 3M.

The Algorithm 3 modifications to give Algorithm 3M are as follows:

- The set of the start edge-vertex elements $S$ and the set of the finish edge-vertex elements $F$ obey the relation: $S,F \subseteq I(G) \times \{\text{head, end}\}$, $|S|=2$, $|F|=1$ (a two-sources single destination search algorithm).

The elements $[[e_x,v_a], \text{head}],[[e_y,v_g], \text{end}] \in S$ specify that the relocation object – train stands on edge/track $e_x$ so that the head-of-train is oriented towards vertex $v_a$ and the end-of-train is oriented towards vertex $v_g$. The element $[[e_x,v_g], \text{ind}] \in F$ specifies that the relocation finish track/edge is $e_y$ and this must be entered through the end $v_g$. In addition, it is determined that the train must enter the finish track by its precisely defined part (front part or back part, ind\in\{\text{head, end}\}).

### TABLE 20. Mean calculation times of the Algorithm 4 related to searching a shortest walk.

| ALGORITHM 4 | $|V(G)|$ | $|E(G)|$ | $L$ | Calculated walks | $\text{mean}^q$ | $\text{max}^q$ |
|-------------|--------|--------|-----|-----------------|---------------|-------------|
| Demonstration railway yard |       |        | 22  | 24              |               |             |
| total length of tracks | 3 418 m |        |     |                 |               |             |
| Pardubice main station incl. adjacent railway network | 677   | 710    |     |                 |               |             |
| total length of tracks: | 71 514 m |        |     |                 |               |             |
|                          | 0      | 190    | 0.010 | 0.087 |
|                          | 20     | 190    | 0.009 | 0.025 |
|                          | 50     | 120    | 0.010 | 0.031 |
|                          | 100    | 119    | 0.010 | 0.028 |
|                          | 200    | 50     | 0.010 | 0.018 |
|                          | 300    | 9      | 0.008 | 0.019 |
|                          | 400    | 2      | 0.007 | 0.018 |
|                          | 500    | 2      | 0.007 | 0.018 |
|                          | 0      | 224112 | 1.229 | 4.124 |
|                          | 20     | 142889 | 1.291 | 3.883 |
|                          | 50     | 108781 | 1.321 | 4.139 |
|                          | 100    | 78340  | 1.327 | 3.52 |
|                          | 200    | 23619  | 1.206 | 3.697 |
|                          | 300    | 2242   | 1.135 | 3.158 |
|                          | 400    | 828    | 1.169 | 3.26 |
|                          | 500    | 198    | 1.174 | 2.168 |
TABLE 21. State space of Algorithm 3M, \( S = \{ [e_5, v_{12}], [e_5, v_{10}], F = \{ [e_4, v_{11}], \text{head} \} \}, L = 120. \)

| \( k \) | head dumped \( d_k \) | end dumped \( d_k \) | \( m_k \) |
|-----|----------------|----------------|------|
| 1   | -              | -              | \([e_5, \text{head}]\) |
|     |                |                | \([e_5, \text{end}]\) |
| 2   | -              | -              | \([e_5, \text{head}]\) |
|     |                |                | \([e_5, \text{end}]\) |
| 3   | -              | -              | \([e_5, \text{head}]\) \([e_3, \text{head}]\) |
|     |                |                | \([e_3, \text{end}]\) \([e_3, \text{end}]\) |
| 4   | -              | -              | \([e_5, \text{head}]\) \([e_3, \text{head}]\) \([e_5, \text{head}]\) |
|     |                |                | \([e_5, \text{end}]\) \([e_3, \text{end}]\) \([e_3, \text{end}]\) |
| 6   | -              | -              | \([e_4, \text{head}]\) \([e_4, \text{head}]\) |
|     |                |                | \([e_4, \text{end}]\) \([e_4, \text{end}]\) |
| 7   | 1607           | 1346           | \([1607, e_5, \text{head}]\) \([e_3, \text{head}]\) \([e_3, \text{head}]\) |
|     |                |                | \([1346, e_4, \text{end}]\) \([e_3, \text{end}]\) \([e_3, \text{end}]\) |
| 8   | -              | 180            | \([e_3, \text{head}]\) \([e_3, \text{head}]\) \([e_3, \text{head}]\) |
|     |                |                | \([e_3, \text{end}]\) \([e_3, \text{end}]\) \([180, e_5, \text{end}]\) |
| 9   | -              | -              | \([e_4, \text{head}]\) \([e_4, \text{head}]\) |
|     |                |                | \([e_4, \text{end}]\) \([e_4, \text{end}]\) |
| 10  | -              | 139            | \([e_5, \text{head}]\) \([e_5, \text{head}]\) |
|     |                |                | \([139, e_5, \text{end}]\) \([e_5, \text{end}]\) |
| 11  | 1293           | 194            | \([e_5, \text{head}]\) \([1293, e_4, \text{head}]\) |
|     |                |                | \([e_5, \text{end}]\) \([194, e_4, \text{end}]\) |
| 12  | 0              | -              | \([0, e_5, \text{head}]\) \([1299, e_5, \text{head}]\) |
|     |                |                | \([e_5, \text{end}]\) \([1299, e_5, \text{end}]\) |
| 13  | 40             | -              | \([1259, e_5, \text{head}]\) \([e_5, \text{head}]\) \([40, e_5, \text{head}]\) |
|     |                |                | \([e_5, \text{end}]\) \([e_5, \text{end}]\) \([e_5, \text{end}]\) |
| 14  | 80             | -              | \([80, e_5, \text{head}]\) \([1219, e_5, \text{head}]\) \([1219, e_5, \text{head}]\) |
|     |                |                | \([e_5, \text{end}]\) \([e_5, \text{end}]\) \([e_5, \text{end}]\) |
| 15  | 118            | 1377           | \([1181, e_5, \text{head}]\) \([1181, e_5, \text{head}]\) |
|     |                |                | \([118, e_5, \text{end}]\) \([118, e_5, \text{end}]\) |
| 16  | 121            | 1380           | \([1178, e_5, \text{head}]\) \([121, e_5, \text{head}]\) |
|     |                |                | \([e_5, \text{end}]\) \([1380, e_5, \text{end}]\) |
If **Example 1A** (partly modifying **Example 1**) is defined as follows:

\[
S = \{[e_5, v_{12}, \text{head}], [e_5, v_{10}, \text{end}], [e_4, v_{11}, \text{head}]\}, F = \{[e_4, v_{11}, \text{head}]\}, L = 120.
\]

then the route obtained is different from that obtained in the initial **Example 1**. The topology of this route (Fig. 10) is as follows:

\[
\begin{align*}
&\text{e}_4(1413, \text{head}) \rightarrow v_{11}(1293, \text{head}) \rightarrow e_{18} \rightarrow v_{13}(1259, \text{head}) \rightarrow e_7 \rightarrow v_{14}(1219, \text{head}) \rightarrow e_{20} \rightarrow v_{15}(1181, \text{head}) \rightarrow e_8 \rightarrow v_{20}(1042, \text{head}) \rightarrow e_{25} \rightarrow v_{22}(982, \text{head}) \rightarrow e_{14} \rightarrow v_{19}(459, \text{head}) \rightarrow e_{22} \rightarrow v_{17}(423, \text{head}) \rightarrow e_9 \rightarrow v_{16}(121, \text{head}) \rightarrow e_{21} \rightarrow v_{14}(80, \text{head}) \rightarrow e_7 \rightarrow v_{13}(40, \text{head}) \rightarrow e_{19} \rightarrow v_{12}(0, \text{head}) \rightarrow e_5
\end{align*}
\]

- A case of the state space of **Algorithm 3M** when the route search computation according to **Example 1A** is over is included in Table 21. The mean and max. times taken by computations of **Algorithm 3M** are presented in Table 22. The following holds for the set \(m_k\) of the vector \(M\):

\[
m_k = \{[\text{len}_j, e_j, \text{ind}] | \text{len}_j \in R^+_Y, e_j \in Y(v_k), \text{ind} \in \{\text{head}, \text{end}\}, v_k \in V(G)\}, |m_k| = 2 \deg(v_k), k = 1, \ldots, |V(G)|.
\]
TABLE 22. Mean calculation times of the Algorithm 3M related to searching a shortest walk.

| ALGORITHM 3M                  | |          |          | Calculated | |          |          |
|-------------------------------|-------------------------------|----------|-------------------------------|----------|----------|----------|
|                               | [V(G)]                         | [E(G)]   | [L]                          | Calculated walks | min_{t}  | max_{t}  |
| Demonstration railway yard    | [ ]                            | [ ]      | 0                            | 670       | 0.151    | 8.247    |
| total length of tracks        | 22                            | 24       | 20                           | 670       | 0.093    | 0.424    |
| 3 418 m                       |                               |          | 50                           | 416       | 0.081    | 0.460    |
|                               |                               |          | 100                          | 414       | 0.078    | 0.515    |
|                               |                               |          | 200                          | 156       | 0.066    | 0.258    |
|                               |                               |          | 300                          | 24        | 0.068    | 0.249    |
|                               |                               |          | 400                          | 8         | 0.060    | 0.276    |
|                               |                               |          | 500                          | 8         | 0.033    | 0.733    |
| Pardubice main station incl. adjacent railway network | 677                           | 710      | 0                            | 448224    | 1.524    | 9.642    |
| total length of tracks        |                               |          | 20                           | 285322    | 1.580    | 4.609    |
| 71 514 m                      |                               |          | 50                           | 217710    | 1.620    | 5.103    |
|                               |                               |          | 100                          | 155832    | 1.641    | 4.793    |
|                               |                               |          | 200                          | 54288     | 1.562    | 4.450    |
|                               |                               |          | 300                          | 4484      | 1.480    | 4.217    |
|                               |                               |          | 400                          | 1656      | 1.495    | 3.782    |
|                               |                               |          | 500                          | 396       | 1.503    | 3.839    |

For Example 1A, we can illustrate the element \([118,e_{20}\text{head}]_{m_{15}}\), which stores the current value (118) of the distance mark of vertex \(v_{15}\) with respect to its attainment by the train head through edge \(e_{20}\). As a practical consequence of the structure modification of the elements from the sets \(m_{k}\), we can follow not only the potential attainment of any vertex through all incident edges but also the attainment by either of the train (relocation object) ends. The algorithm is successfully terminated if the specified finish position has been attained by the required train end. If the finish position is approached by the other train end, the train must not proceed further by transiting the finish edge. This must be taken into account in the algorithm.

In relation to the structure modification of the elements of the sets \(m_{k}\), the implementation of Algorithm 3M should modify (against that of Algorithm 3) those parts that are associated with the initializations of the above elements, with their reading and re-marking (by the \(\text{Get Mark}\) and \(\text{Set Mark}\) functions). Furthermore, provisions can be made for the \(\text{Try Change Reverse}\) routine to be able to change the train ride orientation – once the reversal is over, the other adjacent vertices will be attained by the train end opposite to that before the reversal. As regards the attainment of the vertices potentially by either train end, the use of the two-row vectors \(\text{head D a end D}\) is convenient. They are used in a way analogous to that for vector D in Algorithms 1-4.

For the finish position of the relocation object O (whose length is \(L\)) it can be potentially required that the \(\text{heot}\) is located in a given point \(R\) at a distance of \(1\) meters from the approach end/vertex \(v_{g}\) of the finish track/edge \((e_{y},v_{g}\text{head})_{F}, \Delta \geq L \wedge \Delta \leq k((e_{y},v_{g}))\) – point \(R\) can mirror, e.g., the railway signal. For the above example, the total object O relocation length is \(\text{head}_{d_{g}} + \Delta\) where \(\text{head}_{d_{g}}\) is the value of the relevant distance mark of vertex \(v_{g}\) stored in vector \(\text{head D}\).

The procedures to analyse Algorithm 3M may be similar to those used in the analysis of Algorithms 1-4 (and described in Section VIII.A). Since the train orientation is relevant, the number of variants by which each edge of graph \(G\) is passed is twice as high as in the orientation-free approach. In addition, each vertex \(v_{k} \in V(G)\) has available a maximum of six marks \(|m_{k}|=2 \deg(v_{k}),\deg(v_{k}) \in \{1,2,3\}\). In this context, the upper bounds of the number of edges and vertices in the virtual graph \(G\) are \(n=6\ n\), \(m=8\ m\). If the upper bound of the number of edges is \(m=(3/2)\ n\), then the complexity of the modified Dijkstra’s algorithm can be alternatively expressed as \(O(6\ n \log 6\ n + 12\ n)\). After adding the complexity of the \(\text{DFS-algorithm}\), the total computational complexity of
TABLE 23. Comparison of algorithms searching for the shortest train routes.

| Comparison category     | DynSTR-algorithms                                                                 | SCR-algorithm |
|-------------------------|-----------------------------------------------------------------------------------|----------------|
| Algorithm mission       | Searching for a single shortest route within the model of the currently occupied track infrastructure (frequently called during the simulation run) | Searching for a single shortest route within the model of the track infrastructure (however, preprocessing phase makes preparation for searching for all-shortest routes) |
| Phases of algorithms    | Single-phase algorithms combining the principles of Dijkstra’s algorithm and Depth-first search algorithm | Two-phase algorithm composed of: Preprocessing phase (applying linear programming) Routing phase (utilizing Dijkstra’s algorithm) |
| Track infrastructure model | Edge-weighted undirected graph                                                      | Edge-weighted directed (final) graph as a transformation from the initial edge-weighted undirected graph |
| Type of task solved     | Searching for the shortest admissible walk on the graph                            | Searching for the shortest path on the final graph |
| Algorithm parameters    | The length of the train (train orientation within algorithm modification)           | The length of the train, train orientation |
| Train self-collision problem | Not considered                                                                    | Only track layouts where self-collisions are not possible are bound. |
| Computation time demands | Demonstrated in case studies (one algorithm computation takes on average units of milliseconds) | Not sufficiently documented |
| Overall computational complexity | Train orientation-free algorithms: $O((15/8) n^2 + 3 n \log 3 + 6 n)$ | Routing phase: $O(4 m + 2 n + (4 n + 2) \log (4 n + 2))$ |
|                         | Accounting for a train orientation: $O((15/8) n^2 + 6 n \log 6 n + 12 n)$        | Preprocess phase (linear programming approach): not specified at all |
|                         | $n$ – number of vertices within the primary graph-model                            | $n$ – number of vertices within the primary graph-model |
|                         | $m$ – number of edges within the primary graph-model                                | $m$ – number of edges within the primary graph-model |
| Deployment in practice  | MesoRail simulation tool                                                            | Not known |

**Algorithm 3M** is:

$$O((15/8) n^2 + 6 n \log 6 n + 12 n).$$  \hfill (2)

Since only the basic concept of Algorithm 3M is described here, the changes with respect to this algorithm will not be described in more detail.

**E. ALTERNATIVE APPROACHES TO THE TRACK ROUTE EVALUATION**

Evaluation of the identified track routes (admissible walks in the graph) by means of the total distances travelled by the relocation objects may not always be the most suitable approach. Alternative measures may include, e.g., inclusion of the object relocation times. This means that the graph edge “length” weights will be completed with their “time” weights. Such time measures ($\tau$: $E(G) \rightarrow \mathbb{R}^+$) mirror the estimated times required for the particular relocation object to pass through the relevant track or track segment. The times taken by train reversals would also have to be estimated.

**X. COMPARISON WITH OTHER ALGORITHMS**

Comparison of the presented new algorithms with other algorithms that are focused on searching for the shortest routes within the track infrastructure is quite difficult, due to the fact that for other algorithms:

- their detailed formal description is often not available – this typically applies to algorithms used within commercial software tools focused on rail traffic simulations,
- their total asymptotic computational complexity is not quantified,
- their focus is different to a greater or lesser extent (e.g., some algorithms do not take into account the length of the relocation object),
- it is not possible to practically verify the time complexity of their computations (due to the unavailability of their implementation).

The algorithms described in Sections IV and VI and their potential further extensions in Section IX, are referred to as DynSTR-algorithms (Dynamic Search of Train Routes).
From the available literature, these algorithms can only be roughly compared with the SCRP-algorithm(Single-Cut Routing Problem) described in [11] and [12]. The aforementioned comparisons can be made in the categories listed in Table 23.

The SCRP-algorithm is a two-phase algorithm, in which the first phase (preprocessing phase) is used to calculate for all switches on the primary track model (edge-weighted undirected graph) whether it is possible to reverse the specified train (defined by its length L) “behind” them. Based on the result of this phase, the transformation of the primary graph into the final graph (edge-weighted directed graph) is performed. On the final graph, the shortest relocation route is computed (routing phase) as a classical shortest path using Dijkstra’s algorithm.

In the preprocessing phase, a linear programming method (computing the reduced longest path problem) is applied, which is able to achieve the result in polynomial time for certain track infrastructure topologies. However, the specific computational complexity for the relevant class of problems is not given by the authors.

On the other hand, DynSTR-algorithms are single-phase algorithms that combine the principles of two computational methods (a modified Dijkstra’s shortest path search algorithm) and a Depth-first search algorithm that computes path topologies for train reversals only “behind” those switches that were reached during the computation.

A classical comparative analysis of these algorithms cannot be performed, since the authors of this paper do not have enough information on both the implementation of the SCRP-algorithm (in particular its preprocessing phase) and the expression of its respective total asymptotic computational complexity.

However, for the needs of massively called shortest route computations during traffic simulations, the deployment of DynSTR-algorithms seems to be more appropriate, as these purposely focus only on the computation of one specific route (on the currently occupied track infrastructure) and the corresponding computations do not focus on any parts of the infrastructure model, that were not reached during the computation (unlike the SCRP-algorithm, which calculates the feasibility of reversals for a given train with respect to all switches within the track layout during the preprocessing phase – regardless of whether or not these are used in the next routing phase).

XI. CONCLUSION

In conclusion, the above algorithms for automated dynamic search (made during the simulation computation) of the shortest track routes can be included in microscopic railway traffic simulators. The routes found mirror the optimum ways to relocate an object within a specifically occupied rail infrastructure represented by a model-graph. The input parameters for the simulation include the relocation object’s start and finish positions and its length.

A. SUMMARY OF KEY CHARACTERISTICS OF ALGORITHMS

The novelties of the presented algorithms can be summarized as follows:

- The shortest track route within rail infrastructure is searched within the relevant infrastructure model (represented by the edge-weighted undirected graph) as the shortest admissible walk.
- The algorithms are based on the combination of the principles of the Dijkstra’s algorithm and the Depth-first search algorithm.
- The lengths of modelled relocations objects (the trains or train sets) are considered during computations of shortest admissible walks.

The use of the algorithms described in this paper extends the modelling possibilities when searching for realistic track routes (especially for complicated shunting operations), which contributes to better modelling of complex railway traffic (than in the relevant existing rail traffic simulators) and thus to better application of the results of traffic simulations in practice. In addition, the application of the algorithms supports significant simplification of the simulation experiment scenario parametrization compared to approaches that do not include such dynamic computations. The user is freed from the task to tediously pre-define many track routes and hence, the time needed to set up the computer simulation models of the rail systems is appreciably shorter.

The track route optimization method consisting in a minimization of the total route length responds to the practical requirement to cut the railway traffic cost as much as reasonably achievable. In the current environment where electromobility (also in the railway domain) is massively supported, the optimization criterion can be applied with advantage to the traffic patterns of the electric/battery shunting locomotives operating within non-electrified infrastructure zones. For this case, the minimization of the shunting route lengths is interrelated with a maximization of the times between the locomotive recharging periods.

REFERENCES

[1] A. Kavička and P. Krýže, “Dynamic automated search of shunting routes within mesoscopic rail-traffic simulators,” J. Adv. Transp., vol. 2021, pp. 1–22, Apr. 2021, doi: 10.1155/2021/8840516.
[2] A. Gosavi, Simulation-Based Optimization (Operations Research/Computer Science Interfaces Series). Boston, MA, USA: Springer, 2015, doi: 10.1007/978-1-4899-7491-4.
[3] G. Medeossi and S. de Fabris, “Simulation of rail operations,” Int. Ser. Oper. Res. Manage. Sci., vol. 268, pp. 1–24, Mar. 2018, doi: 10.1007/978-3-319-72153-8_1.
[4] R. Diviš and A. Kavička, “Reflective nested simulations supporting optimizations within sequential railway traffic simulators,” ACM Trans. Model. Comput. Simul., vol. 32, no. 1, pp. 1–34, Jan. 2022, doi: 10.1145/3467965.
[5] J.-F. Cordeau, P. Toth, and D. Vigo, “A survey of optimization models for train routing and scheduling,” Transp. Sci., vol. 32, pp. 580–404, Nov. 1998, doi: 10.1287/trsc.32.4.380.
[6] L. G. Kroon, H. E. Romeijn, and P. J. Zwaneveld, “Routing trains through railway stations: Complexity Issues,” Eur. J. Oper. Res., vol. 98, no. 3, pp. 485–498, 1997, doi: 10.1016/S0377-2217(95)00342-8.
