Polynomial-Time Optimal Liveness Enforcement for Guidepath-based Transport Systems

Spyros Reveliotis[a,1], Tomáš Masopust[b,2], Michael Ibrahim[c]

[a]*School of Industrial & Systems Engineering, Georgia Institute of Technology*
[b]*Faculty of Science, Palacky University in Olomouc, and Institute of Mathematics of the Czech Academy of Sciences*
[c]*Department of Computer Engineering, Faculty of Engineering, Cairo University*

Abstract

Zone-controlled guidepath-based transport systems is a modeling abstraction representing the traffic dynamics of a set of agents circulating in a constricted medium. An important problem for the traffic coordinator of these systems is to preserve liveness, that is, the ability of each agent to successfully complete its current trip and to be engaged in similar trips in the future. We present a polynomial-time algorithm for enforcing liveness in a class of these systems, in a maximally permissive manner. Our result is surprising and applicable in the traffic control of various unit-load material handling systems and other robotic applications.

Keywords: Guidepath-based transport systems; traffic liveness and its enforcement; deadlock avoidance; discrete event systems

1. Introduction

Zone-controlled guidepath-based transport systems is a model representing the traffic dynamics of a set of agents circulating in a constricted medium. In this model, the traffic-supporting medium is abstracted to a set of interconnected zones that are traversed by the agents as they travel between different locations. The zone connectivity is represented by a multigraph that is known as the guidepath network. Depending on the traffic dynamics and the abstracting semantics, the zones can be represented either by the vertices or by the edges of this multigraph.

In order to ensure a certain level of separation among the traveling agents, and establish various notions of safety for the agents and the transported payloads, it is stipulated that (i) each zone can be occupied by no more than one agent at a time, and (ii) the allocation of the requested zones to the agents observes a zone allocation protocol encoded and enforced by a traffic coordinator. Additional restrictions on the motion of agents and on the ensuing traffic dynamics may be defined by inherent limitations of the agents and by other attributes and concerns depending on the underlying application context.

Daugherty et al. [1] provide an extensive survey of the literature on the real-time traffic management of zone-controlled guidepath-based transport systems. This model is a natural abstraction of unit-load material handling systems (MHSs), like the automated guided vehicles (AGVs) and the overhead monorail systems used in many production and distribution facilities [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. The model has further been applied in many robotic applications [14, 15, 16, 17], in the representation of some board games [18, 19, 20], in the programming of animations developed by the current video game industry [21], and more recently, in the modeling and analysis of the elementary physical operations taking place in quantum computing [1, 22].

Email addresses: spyros@isye.gatech.edu (Spyros Reveliotis), masopust@math.cas.cz (Tomáš Masopust), michael.nawar@eng.cu.edu.eg (Michael Ibrahim)

[1]Corresponding author. Partially supported by NSF grant ECCS-1707695.

[2]Supported by the Ministry of Education, Youth and Sports under the INTER-EXCELLENCE project LTAUSA19098, by the Czech Science Foundation grant GC19-06175J, by the university grant IGA PrF 2020 019, and by RVO 67985840.

For the systematic investigation of the traffic dynamics taking place in the guidepath-based transport systems, and the effective real-time management of the generated traffic, these transport systems are classified by the following attributes [23]:

(i) The accommodation of the idle agents in the guidepath network. Many systems avail of a *home* location where the idling agents retire. In the case of MHS and robotics applications, the home location is an area where the idle vehicles rest, receive maintenance, and recharge batteries. In the case of quantum computing, the home location is the system memory that holds the qubits – the ionized atoms that act as the primary information carriers – that are not involved in the currently executed operation. A guidepath-based transport system that possesses a home location is classified as *open*; the rest as *closed*.

(ii) The agent ability to reverse their direction of motion while traversing any zone of the guidepath network. Systems that provide this ability are classified as *reversible*; the rest as *irreversible*. Irreversibility of agents can result from inherent limitations of the mechanisms that generate the agent motion, or from spatial and other constrictions that are imposed by the guidepath network and render infeasible and/or unsafe the reversal of the agent motion.

(iii) The mechanism generating the routes to be followed by the agents, as they reach their various destinations in the guidepath network. If these routes are predetermined upon the allocation of a certain trip to an agent, then the routing scheme is classified as *static*. If the agent routes are determined more incrementally, while accounting for the prevailing traffic conditions in various parts of the guidepath network, then the routing scheme is classified as *dynamic*.

Combinations of the above attributes define several classes of zone-controlled guidepath-based transport systems. These classes pose different control requirements for the traffic coordinators. Also, the complexity of certain control requirements can be significantly different across different classes.

A primary requirement for the traffic coordinator of any zone-controlled guidepath-based transport system is the establishment of expedient traveling for the agents. This requirement is attained through (a) a pertinent allocation of the arising transport tasks to available agents and (b) the routing and scheduling of the resulting trips through various zones of the underlying guidepath network in a way that controls the congestion and the delays experienced by the agents. The works of [1, 24] have investigated this traffic control problem and the synthesis of a solution for certain classes using a Model Predictive Control (MPC) scheme.

For *irreversible* zone-controlled guidepath-based transport systems, it is necessary to impose an additional type of control for preserving the *liveness* of the traffic, that is, the ability of each agent to successfully complete its current trip and to be engaged in similar trips in the future [23, 24]. The primary reason for a loss of liveness is the formation of deadlocks and livelocks among some traveling agents, which may prevent the agent advancement from their current zones, or restrict their circulation to particular regions of the guidepath network.

A controller preventing the development of deadlocks and livelocks is known as a *liveness-enforcing supervisor* (LES). A desirable property of an LES is *maximal permissiveness*. A maximally permissive LES establishes traffic liveness while imposing the minimal possible restriction on the generated traffic. Therefore, this LES maximizes the performance potential for the transport system.

A maximally permissive LES permits the advancement of a set of agents to some neighboring zones in the guidepath network only if the traffic state resulting from this advancement satisfies certain conditions. These conditions characterize the traffic state as *live*. In statically routed, irreversible, zone-controlled guidepath-based transport systems, deciding the liveness of a traffic state is NP-hard [23, 25]. On the other hand, this decision problem is open for, both, open and closed, irreversible, dynamically routed, zone-controlled guidepath-based transport systems.

Researchers have coped with this situation by developing suboptimal LES that might not be maximally permissive, but are polynomial with respect to the size of the guidepath network. For some examples, we refer the reader to the works of [9, 10, 11, 13, 26, 27, 28, 29, 30].

Recently, we identified certain traffic-state attributes that render easier the assessment of liveness. Namely, if the traffic state of an open, irreversible, dynamically routed, zone-controlled, guidepath-based transport system is *totally congested*, that is, if every zone of the network is occupied by a traveling agent, then the assessment of its liveness is of linear complexity with respect to the size of the guidepath network [25]. A similar result holds if some graph induced by a particular graphical representation of the traffic state possesses a *tree*-structure [31]. We provided representations and algorithms for computing this graph and for resolving liveness with polynomial complexity with respect to the size of the guidepath network. We further employed these tools in a general algorithm that can assess liveness of any

arbitrary traffic-state of an open, irreversible, dynamically routed, zone-controlled, guidepath-based transport system, but without polynomial-time guarantees [32].

In this paper, we answer the open question of the worst-case complexity of deciding traffic-state liveness for *open*, irreversible, dynamically routed, zone-controlled guidepath-based transport systems, by developing a polynomial-time algorithm for this problem. Our algorithm also enables a practical deployment of the maximally permissive LES for this class of guidepath-based transport systems, which has a strong presence in contemporary applications. From a methodological standpoint, we build upon the representational and analytical concepts and tools developed in [13, 31]. However, in order to address the representational and analytical needs that arise in the subsequent developments, we extend and complete these past results in a substantial manner. Furthermore, our theoretical developments are complemented by some numerical experimentation that is presented in the last part of the paper and reveals that the developed algorithm executes extremely fast, even when applied on traffic states coming from some very large instantiations of the considered transport systems. Finally, our developments also provide a methodological base for addressing the companion decision problem of assessing traffic-state liveness in *closed*, irreversible, dynamically routed, zone-controlled, guidepath-based transport systems. For space-related reasons, we do not treat this problem here, but rather establish some connections in the conclusions.

The paper is organized as follows. Section 2 defines the considered class of guidepath-based transport systems and the liveness-assessment problem, and provides background material that is necessary for the main developments. Section 3 presents the new algorithmic developments together with the experimental results. Finally, Section 4 concludes the paper and highlights some future directions. In addition, Appendix A provides a statement of the Master Theorem of complexity theory [33], that is necessary for the complexity analysis of our algorithms, and Appendix B collects, for the reader's convenience, the primary notation, concepts and terminology used in the manuscript.

2. Preliminaries

This section provides background material that is necessary for the development of our results. This material concerns (i) the graph-theoretic concepts and results that are employed in this work, (ii) the complete definition of the considered transport systems and their liveness, (iii) a representation of some qualitative dynamics of the generated traffic that we developed in [31], and (iv) the connection of this representation to the notion of traffic-state liveness.

2.1. Graph-theoretic concepts and results

This section overviews the necessary graph-theoretic concepts and results. For more details, we refer to [34].

A *multigraph* $G = (V, E, \xi)$ consists of a finite set of *vertices* or *nodes*, V, a finite set of *edges*, E, and a function ξ that associates with each edge $e \in E$ two vertices v_1 and v_2 (not necessarily distinct), the *endpoints* of e. In particular, the endpoints v_1, v_2 can be an ordered or an unordered set. In the first case, edge e is *directed*; otherwise, it is *undirected*. Graph G is *undirected* if all its edges are undirected. Similarly, G is *directed* if all its edges are directed. *Partially directed graphs*, or *PDGs*, are graphs with some edges directed and the others undirected.

For a directed edge $e \in E$ with $\xi(e) = (v_1, v_2) \in V \times V$, v_1 is its *tail* and v_2 its *head*, and we say that edge e *emanates* from vertex v_1 and *leads to* vertex v_2. An edge where the tail coincides with the head is a *loop*. Two edges e and e' are *parallel* if $\xi(e) = \xi(e')$. A *graph* is a multigraph without parallel edges. A graph is *simple* if it contains no loops and parallel edges. For simple undirected graphs, we write $e = \{v_1, v_2\}$ for an edge e with $\xi(e) = \{v_1, v_2\}$. Similarly, we write $e = (v_1, v_2)$ for simple directed graphs.

Vertices v_1, v_2 of an undirected graph G are *adjacent* or *neighboring* if they are endpoints of an edge e of G; edge e is *incident to* vertices v_1 and v_2. The number of edges of G incident to a vertex v is the *degree* of v. A vertex of degree zero is *isolated*, and a vertex of degree one is a *leaf* (vertex) of G. The *minimal degree* of G is the minimal degree among its vertices. Analogously we define the above notions for directed graphs. Additionally, we define the *indegree* (resp., *outdegree*) of a vertex v as the number of edges e having v as their head (resp., tail) vertex. A vertex with zero indegree is a *source* vertex, and a vertex with zero outdegree is a *terminal* vertex.

The undirected graph $\tilde{G}$ that *is induced* by a (partially) directed graph G has the same sets of vertices and edges as G, but the edges are undirected in $\tilde{G}$. A subgraph H of a graph $G = (V, E, \xi)$ is a graph $H = (V', E', \xi')$ such that $V' \subseteq V$, $E' \subseteq E$, V' contains all the endpoints of every edge $e \in E'$ according to the definition of these vertex sets by ξ, and ξ' is the restriction of ξ on the edge set E'. The *union* of two or more multigraphs $\mathcal{G}_i$, $i = 1, \ldots, n$, is a

multigraph $\mathcal{G}$ that has as vertex (resp., edge) set the union of the vertex (resp., edge) sets of the multigraphs $\mathcal{G}_i$. An undirected graph G is *bipartite* if its vertex set V is partitioned into two subsets V_1 and V_2, and for every edge $e \in E$, $\xi(e) \cap V_1 \neq \emptyset \wedge \xi(e) \cap V_2 \neq \emptyset$.

A *walk* $\pi = \langle v_0, e_1, v_1, \ldots, e_k, v_k \rangle$, $k \geq 0$, in an undirected graph is a sequence of vertices and edges such that, for $1 \leq i \leq k$, v_{i-1} and v_i are the endpoints of e_i. A *trail* is a walk with no repeated edge. A *path* is a walk with no repeated vertex. These concepts can be extended to directed and partially directed graphs by requesting that each oriented edge e_i, $1 \leq i \leq k$, emanates from vertex v_{i-1} and leads to vertex v_i. The vertices v_0 and v_k on path π are the *terminal* vertices. The nonterminal vertices of π are *internal*. We also say that path π connects its terminal vertices v and v' in G. In the case of directed graphs, we refer to vertex v_0 as the *tail* vertex, and to the destination vertex v' as the *head* vertex. A *cycle* c of an undirected graph G is a subgraph of G such that, after removing a single edge, the subgraph is a path in G. In directed graphs, a *(directed) cycle* is created from a path by adding an edge connecting the head vertex and the tail vertex. A *directed acyclic graph* or a *DAG* is a directed graph without cycles. The *length* of a path, cycle, walk or trail is the number of edges in the corresponding subgraph.

An undirected graph G is *connected* if every two vertices v_1, v_2 are terminal vertices for a path of G. A directed graph G is *strongly connected* if, for every two vertices v_1, v_2, there is a directed path leading from v_1 to v_2. A directed graph G is *weakly connected* if the induced undirected graph $\tilde{G}$ is connected. The *components* of an undirected graph are its maximal connected subgraphs. A component is *trivial* if it has no edges; otherwise, it is *nontrivial*. For an undirected graph G, $G - v$ denotes the graph that contains all vertices and edges of G except for vertex v and the edges incident to v. Similarly, $G - e$ denotes the graph that contains all vertices and edges of G except for edge e. A vertex v of an undirected graph G is a *cut-vertex* (or an *articulation*) if the graph $G - v$ has more components than G. Similarly, an edge e of an undirected graph G is a *cut-edge* (or a *bridge*) if the graph $G - e$ has more components than G. Notice that e is a cut-edge if and only if it does not belong on any cycle of G.

A *block* of an undirected graph G is a maximal connected subgraph of G that has no cut-vertex. An isolated vertex of G is a block. We refer to such blocks as *vertex-blocks*. If G is connected with no cut-vertex, it is a single block. Consequently, an undirected graph consisting of a single edge is a block. A single edge of a larger graph is a block if it is a bridge. We refer to such single-edge blocks as *bridge-blocks*. A block of a connected undirected graph G that is not a single edge, has a minimal vertex degree of 2, and it is a *biconnected* (or *bi*)*-block*.

An undirected graph T is a *tree* if it is connected and acyclic. Hence, every pair of vertices v, v' of T is connected by a unique path. A *rooted tree* is a tree T with one vertex r of T chosen as the *root*. For each vertex $v \neq r$, the node v' that is adjacent to v on the path from v to the root r is the *parent (vertex)* of v. All the remaining neighboring vertices of v in T are the *children* of v. The *ancestors* of v are all the vertices on the path from the parent of v to the root. The *descendants* of v are all the vertices u for which v is an internal vertex on the path from u to the root. In a rooted tree, a *leaf* is a vertex with no children. An *in-tree* is a DAG whose induced undirected graph is a rooted tree, and whose edges point towards the root.

The decomposition of a connected, undirected graph G to its blocks can be effectively represented by the *block-cutpoint tree*. This is a bipartite graph in which one vertex subset, V_1, consists of the cut-vertices of G, and the second vertex subset, V_2, has a vertex b_i for every block B_i of G. There is an edge $\{v, b_i\}$ in the block-cutpoint tree with $v \in V_1$ and $b_i \in V_2$ if and only if v is a vertex of the block B_i. An efficient algorithm for computing the block-cutpoint tree can be found in [34]

2.2. The considered transport systems and their liveness

The structure of open, irreversible, dynamically routed, zone-controlled guidepath-based transport systems is formally represented by a tuple $(\mathcal{A}, G)$, where $\mathcal{A}$ is the set of agents, and G is the guidepath network. Specifically, $G = (V, E \cup \{h\}, \xi)$ is an undirected connected multigraph with vertex set V, edge set $E \cup \{h\}$, and a function ξ assigning to each edge e an unordered pair of vertices, its endpoints. It is stipulated that G has a minimal vertex degree of 2.[3]

The edge subset E is the set of zones of the guidepath network, and edge h is a self-loop representing the home location of the transport system. The zone control imposed on the transport system requires that no edge $e \in E$ can

[3] While G and its derivative graphs are actually multigraphs, to avoid an overloading of the notation, we employ a representation for edges that is more appropriate for simple graphs. We believe that this practice does not create any confusion, and we take special effort to maintain a clear presentation in the few cases where it might result in some ambiguity.

be occupied by more than one agent $a \in \mathcal{A}$ at a time. On the other hand, edge h has infinite capacity; i.e., it can accommodate an arbitrary number of agents at the same time. We use the notation v_h to denote the single vertex of the self-loop h.

An agent $a \in \mathcal{A}$ executes trips in the guidepath network G, with each trip being defined by a sequence $\Sigma_a = \langle e_i \in E \setminus \{h\} \rangle$ of (not necessarily neighboring) edges that must be visited in the specified order. The route of agent a during its travel between edges e_i and e_{i+1} of Σ_a is a walk in the guidepath network G that is determined by the traffic coordinator in real-time. Irreversibility implies that an agent entering an edge $e = \{v, v'\} \in E$ from vertex v can exit the edge only from vertex v'. Furthermore, the agent can move to an edge $e' = \{v', v''\}$, which is neighboring to edge e at vertex v', only if e' is free of any other agents; this restriction also excludes the possibility of edge-swapping between a pair of agents.

The description of the agent motion through the guidepath network can be based on a notion of *state* that captures (i) the distribution of agents on the edges of the guidepath network, (ii) the direction of the agent motion, and (iii) the edge sequences Σ_a characterizing the remaining trip for each agent $a \in \mathcal{A}$. This state evolves upon the occurrence of *events* that correspond to the transition of an agent a from its current edge e to a neighboring edge e', and the update of its edge sequence Σ_a in the case that the executed transition has attained the milestone of agent a with respect to the edge-visitation requirements expressed by Σ_a.

We are interested in the problem of preserving liveness for the traffic taking place in the above transport system. This task can be attained, in a maximally permissive manner, by restricting the system operation to its live subspace. In the case of open, dynamically routed, zone-controlled guidepath-based transport systems, the live state space consists of those traffic states for which there exists an event-sequence collecting all agents in the home location h (without necessarily completing the visitation requirements defined by sequences Σ_a) [23, 25]. Hence, when deciding liveness of a traffic state coming from an open, dynamically routed, zone-controlled guidepath-based transport system, all agents are destined to the home location h, and the algorithms resolving this problem need not maintain an identity for each agent $a \in \mathcal{A}$ and its sequence Σ_a. We thus define the traffic state as follows:

Definition 1. *The* traffic state *s of an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system is defined by (i) the distribution of agents on the edges of the guidepath network G, and (ii) the orientation of the motion of each agent a on its current edge $e \neq h$.*

In the following, we use the notation $\epsilon(a; s)$ to denote the edge occupied by agent a in state s. An encoding of the traffic state s is through a partially directed graph (PDG) $\hat{G}(s)$; this graph is obtained from the guidepath network G by turning an edge $e \in E$ occupied by an agent a into a directed edge indicating the direction of motion of agent a on e. State s then evolves by advancing a single agent a from its edge $\epsilon(a; s) = (v_1, v_2)$ in the PDG $\hat{G}(s)$ to an edge e' of $\hat{G}(s)$ that is (a) incident to vertex v_2 and (b) an undirected edge in $\hat{G}(s)$.

Let S denote the set of traffic states that can occur in an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system. Then S is a finite set, and the set Q of all events evolving states of S is a finite set, as well. Therefore, the traffic dynamics can be represented by a finite-state automaton (FSA) Φ. In the following, we denote the extended (partial) transition function of Φ by $f : S \times Q^* \to S$.[4] Furthermore, we use s_h to denote the home state of the automaton, i.e., the state where all agents of $\mathcal{A}$ are located at the home edge h.

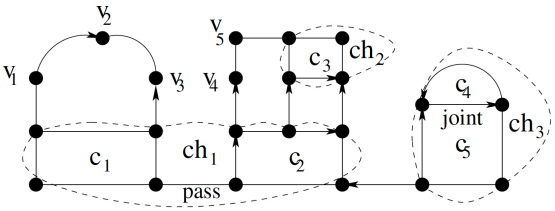
The notion of liveness for open, irreversible, dynamically routed, zone controlled guidepath-based transport systems can now be formally defined as follows [23, 25]:

Definition 2. *In the class of open, irreversible, dynamically routed, zone-controlled guidepath-based transport systems, a traffic state $s \in S$ is* live *if it is* co-reachable *to state s_h in the dynamics of the corresponding FSA Φ; i.e., s is live if there exists $\sigma \in Q^*$ such that $f(s, \sigma) = s_h$.*

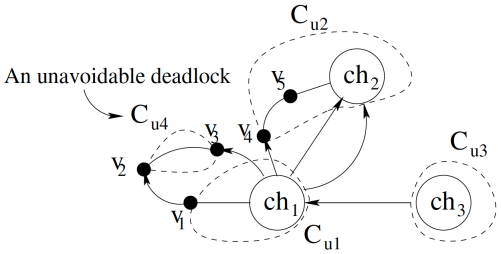
Let S_l denote the set of all live states of Φ. Then, the corresponding (state-)liveness-assessment problem can be defined as follows:

Problem 3 (Traffic-State Liveness)**.** *Determine whether a traffic state s of an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system belongs to S_l.*

[4] Q^* denotes the Kleene closure of the event set Q, i.e., Q^* is the set of all finite sequences of the elements of Q, including the empty sequence ε.



(a) A PDG $\hat{G}(s)$ and the "chain" structure that is recognized in it.



(b) The condensation $C(\hat{G}(s))$ of the above PDG $\hat{G}(s)$ and its u−connected components

Figure 1: Highlighting the definitions and the technical results of Section 2.3.

As discussed in the introduction, we have developed an algorithm that resolves efficiently this problem for a class of traffic-states characterized by a certain property that is defined in terms of a digraph induced by the state-representing PDG $\hat{G}(s)$ [31]. In the following, we first review some of the corresponding results, and then, in Section 3, we employ and extend these results towards the development of an algorithm that provides efficient liveness assessment for any traffic state $s \in S$.

2.3. Representations

In this subsection, we present a series of more compressed representations of a traffic state s that are derived from the PDG $\hat{G}(s)$, and collectively retain the necessary information for resolving the problem whether s is live. These results first appeared in [31], and we refer the reader to that work for more details. Furthermore, Figure 1, reproduced from [31], will help us highlight the various concepts and structures presented in the following.

Given a PDG $\hat{G}(s)$ for a traffic state s, a *path* π in $\hat{G}(s)$ is a sequence $\pi = \langle v_0, e_1, v_1, e_2, \ldots, e_n, v_n \rangle$, $n \geq 0$, where (i) the subsequence $\langle v_i,\ i = 0, \ldots, n \rangle$ consists of distinct vertices of $\hat{G}(s)$, (ii) for all $i = 1, \ldots, n$, e_i is an edge connecting v_{i-1} and v_i, and (iii) if edge e_i is directed, then its direction is from v_{i-1} to v_i; that is, the sense of direction of edges in π is consistent with the direction of motion implied by the ordering of the path vertices. Also, a *cycle* c of $\hat{G}(s)$ has a structure similar to that of a path, but it contains at least one edge and the starting and the ending vertices, v_0 and v_n, are coinciding.

A *joint* between two cycles is a path that belongs to both cycles. A *pass* between two cycles c and c' is a path π such that its first vertex lies on c, its last vertex lies on c', and all edges of π are undirected and do not belong to c, c', or any other directed cycle of $\hat{G}(s)$.

We highlight the above definitions by means of the top part (part (a)) of Figure 1. This part depicts a state s of a guidepath-based transport system with a guidepath graph $G = (V, E)$ corresponding to the undirected graph that is induced by the depicted PDG $\hat{G}(s)$.[5] The agents $a \in \mathcal{A}$ that are not located on the "home" edge h in the considered

[5]The reader should notice that the "home" edge h has not been depicted in this figure, since its inclusion would complicate the accompanying discussion without adding anything substantial to it.

state s are represented by the directed edges of the PDG $\hat{G}(s)$; this representation defines, both, the particular edge that is occupied by the agent in state s and the direction of its motion in this edge.

The considered PDG has five cycles annotated by $c_1, \ldots, c_5$; the reader can check that these are indeed the only closed paths of the PDG $\hat{G}(s)$ that preserve the sense of direction for their directed edges. The (directed) edge labeled as "joint" in the figure constitutes a path belonging to both cycles c_4 and c_5. The edge marked as "pass" in the figure is a path that consists of undirected edges only, and links cycles c_1 and c_2 while possessing no common edges with any of these two cycles or any other directed cycle of $\hat{G}(s)$. On the other hand, the edge of the PDG $\hat{G}(s)$ linking cycles c_2 and c_5 does not constitute a pass for these two cycles because it is directed (i.e., occupied by an agent).

The concepts introduced in the next definition play a central role in the subsequent developments.

Definition 4. *A* chain *ch of a PDG $\hat{G}(s)$ is the subgraph induced by the sequence $ch \equiv \langle c_1, \pi_2, c_2, \pi_3, \ldots, \pi_n, c_n \rangle$, $n \geq 1$, where, for each $i = 1, \ldots, n$, c_i is a cycle and π_i is a path that is a joint or a pass between cycles c_{i-1} and c_i. Two edges e, e' are* chain-connected *(or simply* chained*) if there exists a chain ch that contains both e and e'. Graph $\hat{G}(s)$ is* chained *if every two edges are chained.*

Each of the cycles $c_1, \ldots, c_5$ of the PDG $\hat{G}(s)$ depicted in part (a) of Figure 1, constitutes also a chain for this PDG. But the PDG depicted in part (a) of Figure 1 also possesses the additional chains annotated by ch_1 and ch_3 in the figure. Chain ch_1 consists of cycles c_1 and c_2 linked by the corresponding pass that was discussed in the previous paragraphs, and chain ch_3 consists of the cycles c_4 and c_5 which are linked by the annotated joint. On the other hand, the depicted chain ch_2 comprises cycle c_3 only, which is the only remaining cycle that is not contained in the other two chains.

More generally, chain connectivity is symmetric and transitive. Therefore, we can consider *maximal* chains. Subgraphs of $\hat{G}(s)$ induced by maximal chains are *chained components* of $\hat{G}(s)$. The next definition introduces a new PDG induced by $\hat{G}(s)$ and the notion of the chained component.

Definition 5. *The PDG $C(\hat{G}(s))$ obtained from $\hat{G}(s)$ by replacing each chained component by a simple vertex is the* condensation *of $\hat{G}(s)$. Vertices of $C(\hat{G}(s))$ that correspond to chained components of $\hat{G}(s)$ are* macro-vertices*, while the remaining vertices are* simple*. The capacity $\zeta(ch)$ of a chained component ch of $\hat{G}(s)$ (or, equivalently, of a macro-vertex of $C(\hat{G}(s))$) is the number of free edges of ch located on the cycles of the corresponding subgraph (or, equivalently, the number of free edges of ch that are not bridges in the undirected graph induced by ch).*

The chained components of the PDG $\hat{G}(s)$ in part (a) of Figure 1 are the previously described chains ch_1, ch_2 and ch_3. It can also be checked that $\zeta(ch_1) = 7$, $\zeta(ch_2) = 3$ and $\zeta(ch_3) = 2$. Finally, the condensation $C(\hat{G}(s))$ that results from the reduction of each of the above three chained components to a single macro-vertex (with the same label), is the PDG that is depicted in part (b) of Figure 1.

By construction, condensation $C(\hat{G}(s))$ is an *acyclic* PDG. Furthermore, every path π of $C(\hat{G}(s))$ connecting two different macro-vertices v_1 and v_2 contains a directed edge; otherwise, the chains corresponding to the macro-vertices v_1 and v_2 would not be maximal.

The next abstraction, defined on the condensation $C(\hat{G}(s))$, distinguishes the connected subgraphs of $C(\hat{G}(s))$ that (i) contain no directed edges, and (ii) are connected to the complement part of $C(\hat{G}(s))$ only by directed edges.

Definition 6. *An* undirected component *(or a* u-component*) of condensation $C(\hat{G}(s))$ is a maximal connected subgraph C_u of $C(\hat{G}(s))$ that contains no directed edges. The edges of $C(\hat{G}(s))$ that point to C_u are the* inputs *of C_u, and the edges that point away from C_u are the* outputs *of C_u. A u-component C_u is a* source *if it has no inputs, and a* sink *if it has no outputs. Finally, C_u is* complex *if it contains a macro-vertex of $C(\hat{G}(s))$, and* simple *otherwise.*

Part (b) of Figure 1 also highlights the u-components of the depicted condensation $C(\hat{G}(s))$; these u-components are labelled $C_{u1}, \ldots, C_{u4}$ in the figure. It is important to notice that every u-component of the condensation $C(\hat{G}(s))$ is an undirected tree and it contains at most one macro-vertex. Furthermore, the set of all u-components of $C(\hat{G}(s))$ is partially ordered by the direction of the edges of this condensation.

In the PDG $C(\hat{G}(s))$ depicted in part (b) of Figure 1, it can also be checked that the further compression of each of the identified u-components into a single node results in a directed multigraph. In particular, the directed edges of this multigraph will be (C_{u1}, C_{u2}) with a multiplicity of 3, (C_{u1}, C_{u4}) with a multiplicity of 2, and (C_{u3}, C_{u1}) with a multiplicity of 1. The next definition introduces more formally the reduction of the condensation $C(\hat{G}(s))$ to a directed acyclic (multi)graph (DAG).

Algorithm 1 Determining the maximal chains ch_i, $i = 1, \ldots, k$, of a PDG $\hat{G}(s)$ and their capacities $\zeta(ch_i)$.

Input: The PDG $\hat{G}(s)$.
Output: PDGs $\hat{\mathcal{D}}_i$, $i = 1, \ldots, k$, corresponding to the maximal chains of $\hat{G}(s)$.

1: Convert $\hat{G}(s)$ to a digraph $\mathcal{D}$ by replacing each undirected edge e with two directed edges e' and e'' of opposite directions;
2: Extract the strongly connected components, $\mathcal{D}_1, \ldots, \mathcal{D}_k$, of $\mathcal{D}$;
3: **for** $i := 1$ to k **do**
4: Convert $\mathcal{D}_i$ to a PDG $\tilde{\mathcal{D}}_i$ by replacing each pair of edges (e', e'') introduced in Step 1 with a single undirected edge e;
5: Remove iteratively all leaves and their incident edges from $\tilde{\mathcal{D}}_i$, obtaining the PDG $\hat{\mathcal{D}}_i$;
6: **end for**
7: **for** $i := 1$ to k **do**
8: Convert $\hat{\mathcal{D}}_i$ to the undirected graph D_i by replacing each directed edge in $\hat{\mathcal{D}}_i$ by an undirected edge;
9: Compute the bridges of D_i according to [35];
10: Compute the capacity $\zeta(\hat{\mathcal{D}}_i)$ as the number of free edges in $\hat{\mathcal{D}}_i$ that are not bridges;
11: **end for**
12: **return** $\hat{\mathcal{D}}_i$ and their capacities $\zeta(\hat{\mathcal{D}}_i)$, $i = 1, \ldots, k$.

Definition 7. *We denote by $\mathcal{U}(\hat{G}(s))$ the DAG obtained from the condensation $C(\hat{G}(s))$ by replacing each u-component of $C(\hat{G}(s))$ by a single vertex.*

In the following, we refer to the vertices of $\mathcal{U}(\hat{G}(s))$ as *nodes* and denote them by n. The node that contains the home edge h of the guidepath network G is denoted by n_h and referred to as the home node. We now associate capacity with each node n of $\mathcal{U}(\hat{G}(s))$.

Definition 8. *The* capacity $\chi(n)$ *of a node n of DAG $\mathcal{U}(\hat{G}(s))$ is defined as follows: If n is a simple vertex of the guidepath network G or a simple u-component of $C(\hat{G}(s))$, the capacity $\chi(n) = 0$. If n is a complex u-component C_u of $C(\hat{G}(s))$, the capacity $\chi(n) = \zeta(ch)$, where ch is the chained component that constitutes the unique macro-vertex of C_u. We set $\chi(n_h) = \infty$.*

Furthermore, unless stated otherwise, the DAG $\mathcal{U}(\hat{G}(s))$ is encoded according to the following representation: (I) We define as the *major nodes* of $\mathcal{U}(\hat{G}(s))$ those nodes that (i) either correspond to a complex u-component, or (ii) have their indegree or outdegree greater than 1 (and, therefore, are branching nodes in $\mathcal{U}(\hat{G}(s))$). (II) We collapse each path π that connects a pair (n_1, n_2) of major nodes and contains only non-major nodes of $\mathcal{U}(\hat{G}(s))$ as interior nodes, into a single directed edge (n_1, n_2) weighted by the number of edges in path π; we denote the weight associated with an edge (n_1, n_2) by $w(n_1, n_2)$.

In our previous work, we provide algorithms for the construction of the weighted DAG $\mathcal{U}(\hat{G}(s))$ from the PDG $\hat{G}(s)$, and for the computation of the capacities $\chi(n)$ [31]. The computation is reproduced in Algorithm 1 that takes as input the PDG $\hat{G}(s)$ of a traffic state s, and returns the maximal chains ch_i, $i = 1, \ldots, k$, of the PDG and their capacities $\zeta(ch_i)$. The DAG $\mathcal{U}(\hat{G}(s))$ is obtained from the PDG $\hat{G}(s)$ and the outcome of Algorithm 1 according to the Definitions 6–8.

The worst-case algorithmic complexity of the entire construction is $O(|V| + |E|)$, where V and E are the vertex set and the zone set of the guidepath network G, respectively. Thus, the construction of the more compact representation of a traffic state s introduced above is efficient.

2.4. Inference mechanisms resolving traffic-state liveness

In this section, we review some further results from [31] that use the above concepts and constructs, and provide motivation and a theoretical justification for the algorithm developed in Section 3. We start with the following two propositions.

Proposition 9. *If the condensation $C(\hat{G}(s))$ of a traffic state $s \in S$ contains a* simple sink *u-component C_u, then the agents on the input edges of C_u are heading to an unavoidable deadlock; therefore, state s is not live.* □

Proposition 10. *Consider a* complex *u-component C_u of the condensation $C(\hat{G}(s))$ with a macro-vertex ch, and an agent a located on an input edge of C_u. Since C_u is a tree, there is a unique path p in $\hat{G}(s)$ consisting of free edges through which agent a can access the macro-vertex ch. Also, let $G(ch)$ denote the subgraph of the guidepath graph G that is induced by chain ch. Then, the following two statements, based on the capacity of chain ch, are true:*

1. *If $\zeta(ch) = 0$ and C_u is a* sink *u-component in $C(\hat{G}(s))$, then, every event sequence $\sigma \in Q^*$ advancing agent a on an edge e of the subgraph $G(ch)$ leads to an unavoidable deadlock.*
2. *If $\zeta(ch) \geq 1$, then there exists an event sequence $\sigma \in Q^*$ advancing agent a on an edge of $G(ch)$ in a way that, at the resulting state s', the part of the PDG $\hat{G}(s')$ corresponding to the subgraph $G(ch)$ is chained.* □

Proposition 9 and the first part of Proposition 10 provide a mechanism for detecting unavoidable deadlocks, inferring, thus, non-liveness for a traffic state. On the other hand, a practical implication of the second part of Proposition 10 is that when the head node n of an edge $e = (n', n)$ in DAG $\mathcal{U}(\hat{G}(\tilde{s}))$ has capacity $\chi(n) \geq w(n', n)$ – i.e., the number of agents located on the path p of the underlying PDG $C(\hat{G}(s))$ represented by edge e in $\mathcal{U}(\hat{G}(\tilde{s}))$ – it is possible to clear path p from its occupying agents by absorbing these agents in the chain ch of node n. Furthermore, the traffic state s' that will result from this operation, has a smaller number of chains than the starting state s, and this fact is reflected by the collapse of nodes n' and n, and of the interconnecting edge e, into a single new node in the new DAG $\mathcal{U}(\hat{G}(\tilde{s}))$. Hence, in the following, we think of the aforementioned operation as a "merging" operation, or, more briefly, as a *"merger"*. The notion of a merger acquires a very central role in the subsequent developments thanks to the following theorem that is also established in [31] and constitutes an alternative characterization of traffic-state liveness.

Theorem 11. *A traffic state $s \in S$ of an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system is live if and only if s is co-reachable to a state $\tilde{s}$, for which the PDG $\hat{G}(\tilde{s})$ is chained (or, equivalently, for which the DAG $\mathcal{U}(\hat{G}(\tilde{s}))$ consists solely of the home node n_h and no edges).* □

Theorem 11, together with Propositions 9 and 10, reduce the problem of the assessing the liveness of any given traffic state $s \in S$ of an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system, to a search for a *merger sequence* that will lead from state s to a state $\tilde{s}$, for which the PDG $\hat{G}(\tilde{s})$ is chained (or, equivalently, for which the DAG $\mathcal{U}(\hat{G}(\tilde{s}))$ consists solely of the home node n_h and no edges). This search can be based on the graphical representations of the traffic state s introduced in Section 2.3, and their nodal capacities. The corresponding search process starts with the DAG $\mathcal{U}(\hat{G}(s))$, and iteratively merges some nodes, in the spirit of the discussion that preceded the statement of Theorem 11. Every time a merging operation is executed, the current DAG $\mathcal{U}(\hat{G}(s))$ and the underlying condensation $C(\hat{G}(s))$ are updated to reflect the new traffic state, $\tilde{s}$, resulting from the agent advancements that correspond to this merger. Also, the new condensation $C(\hat{G}(\tilde{s}))$ and the corresponding DAG $\mathcal{U}(\hat{G}(\tilde{s}))$ are computed, and a new merger is sought in the DAG $\mathcal{U}(\hat{G}(s))$. The entire search process will terminate with the singular DAG $(\{n_h\}, \emptyset)$ if and only if the evaluated state s is live.

In the context of the search process that is described in the previous paragraph, Proposition 9 and the first part of Proposition 10 provide a mechanism for fathoming any particular thread that is pursued by the search process, by resolving the non-liveness of the state $\tilde{s}$ that has been reached by this thread.

Furthermore, the outlined search process can provide *a polynomial-time algorithm* for resolving the liveness of a traffic state s if (i) the mergers performed in each iteration are identified and executed in polynomial time with respect to the size of the guidepath network G, and (ii) there is no need for backtracking upon these mergers.

Recently, we developed a polynomial-time algorithm realizing such an efficient search process for the particular class of traffic states where the undirected graph induced by the DAG $\mathcal{U}(\hat{G}(s))$ is a tree [31].

In the next section, we develop a new algorithm that resolves the liveness of *any* traffic state coming from the class of open, irreversible, dynamically routed, zone-controlled guidepath-based transport systems, and is polynomial with respect to the size of the guidepath network G. Similarly to our recent algorithm in [31], the new algorithm is a greedy search for a merger sequence that certifies (non-)liveness of the assessed traffic state. But its development requires a very significant extension of the concepts and the techniques that provided the corresponding results in [31].

3. Main results

In this section, we present a polynomial-time algorithm assessing traffic-state liveness in an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system, together with some experimental results that

demonstrate the effectiveness and the computational efficiency of this algorithm. We organize this section into four subsections. The first subsection formalizes the concept of a nodal *merger* performed in a DAG $\mathcal{U}(\hat{G}(s))$, and differentiates mergers into several types. It also details the execution of a merger through updating the traffic-state representation introduced in Section 2.3. The second subsection presents a polynomial-time algorithm deciding traffic-state liveness for the special case where the undirected graph induced by the DAG $\mathcal{U}(\hat{G}(s))$ of the evaluated traffic state s is biconnected. The third subsection extends the results of the second subsection to the general algorithm which is the main contribution of this work. Finally, the fourth subsection reports the experimental results.

3.1. Mergers

As explained in Section 2.4, our algorithm is a search process for a sequence of mergers reducing the DAG $\mathcal{U}(\hat{G}(s))$ of the input traffic state s to the singular DAG $(\{n_h\}, \emptyset)$. In this subsection, we formally define the notion of a merger, and elaborate this concept to be used in the developed algorithm. We start with the following definition.

Definition 12. *An edge (n', n) of a DAG $\mathcal{U}(\hat{G}(s))$ defines a* feasible basic merger *if*

$$\chi(n) \geq w(n', n) \tag{1}$$

The execution of this merger replaces the edge (n', n) in $\mathcal{U}(\hat{G}(s))$ by a new node, n'', with capacity

$$\chi(n'') = \chi(n') + \chi(n) - w(n', n) \tag{2}$$

Definition 12 is motivated and justified by the second part of Proposition 10. As pointed out in Section 2.4, Condition (1) is necessary and sufficient to advance the $w(n', n)$ agents, located on the path p of the condensation $C(\hat{G}(s))$ represented by the edge (n', n) in $\mathcal{U}(\hat{G}(s))$, into the chain ch of the u-component represented by node n in $\mathcal{U}(\hat{G}(s))$. Emptying path p from its occupying agents establishes a pass linking chain ch to the chain ch' of the u-component represented by node n' in $\mathcal{U}(\hat{G}(s))$. Hence, in the state $\tilde{s}$ resulting from the agent advancements, chains ch and ch' can be merged into a larger chain ch'' that is the unique chain of the new node n'' in $\mathcal{U}(\hat{G}(\tilde{s}))$.

The estimation of the capacity $\chi(n'')$ of the new node n'' in (2) recognizes the following three facts: (i) The advancement of the $w(n', n)$ agents into chain ch reduces the original capacity $\chi(n)$ of chain ch. (ii) The edges released by this advancement are on a pass of the newly formed chain ch'', and hence these edges do not contribute any additional capacity to chain ch''. (iii) The newly formed chain ch'' possesses the original capacity $\chi(n')$ of chain ch' of node n'.

Conditions (1) and (2) imply that, for a feasible basic merger corresponding to an edge (n', n) of a DAG $\mathcal{U}(\hat{G}(s))$,

$$\chi(n'') \geq \chi(n') \,. \tag{3}$$

In the sequel, we acknowledge Condition 3 by saying that the merger (n', n) is a *producer merger with respect to node n'*. A feasible basic merger (n', n) that is also a producer with respect to node n – that is, $\chi(n'') \geq \chi(n)$ – is a *producer merger*. The significance of these merger characterizations in our algorithmic developments is revealed by the following proposition:

Proposition 13. *Let $(\hat{n}, n')$ and (n', n) be two feasible basic mergers in a DAG $\mathcal{U}(\hat{G}(s))$. Then, $(\hat{n}, n'')$ is a feasible basic merger in the DAG $\mathcal{U}(\hat{G}(\tilde{s}))$ resulting from the execution of merger (n', n). If (n', n) is a producer merger, it also preserves the feasibility of any feasible basic merger $(\hat{n}, n)$ in $\mathcal{U}(\hat{G}(s))$.* □

Proposition 13 follows immediately from Definition 12. It implies that the execution of a merger that is a producer for node n preserves all the merging potential of the remaining nodes of the DAG with respect to node n. This interpretation of the producer merger justifies the greedy execution of certain mergers in the conducted search process, and establishes the polynomial-time nature of our algorithm.

In the algorithms presented in the sequel, we also employ mergers that are defined as sequences of basic feasible mergers, and reduce more complex structures of the processed DAG $\mathcal{U}(\hat{G}(s))$ into a single node. We call them *macro-mergers*.

Definition 14. *Let $\mathcal{M} = \langle e_1, e_2, \ldots, e_n \rangle$ be an edge sequence of a DAG $\mathcal{U}(\hat{G}(s))$ with the edge set $\{e_i,\ i = 1, \ldots, n\}$ inducing a weakly connected sub-DAG in $\mathcal{U}(\hat{G}(s))$.*

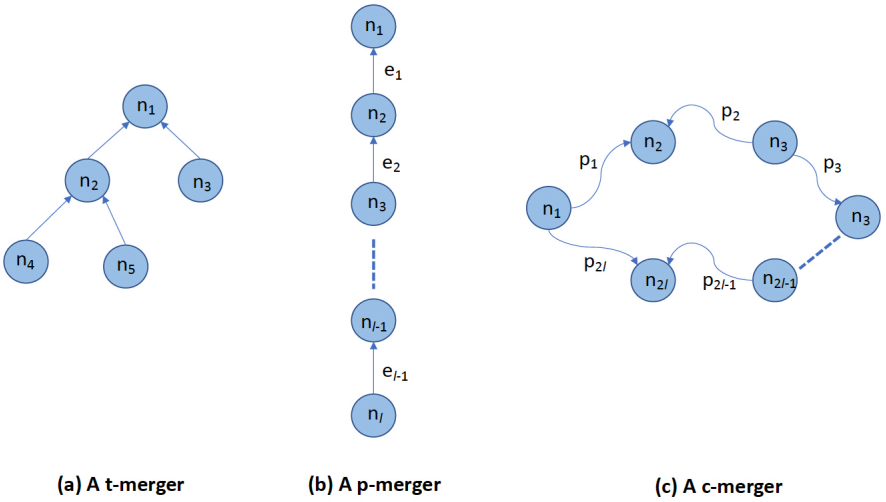


Figure 2: Some macro-merger types.

1. $\mathcal{M}$ *is a* feasible macro-merger *in* $\mathcal{U}(\hat{G}(s))$ *if the basic merger* e_1 *is feasible in* $\mathcal{U}(\hat{G}(s))$ *and every basic merger* e_i, $i = 2, \ldots, n$, *is feasible in the DAG resulting from the execution of the merger sequence* $\langle e_1, e_2, \ldots, e_{i-1} \rangle$ *in* $\mathcal{U}(\hat{G}(s))$.[6]
2. *The node resulting from the execution of* $\mathcal{M}$ *is denoted by* $n(\mathcal{M})$ *and the corresponding nodal capacity is denoted by* $\chi(n(\mathcal{M}))$.
3. $\mathcal{M}$ *is* producer with respect to a merged node n *if* $\chi(n(\mathcal{M})) \geq \chi(n)$; $\mathcal{M}$ *is a* producer *macro-merger if it is producer with respect to every merged node* n.
4. *Let* $\mathcal{M}$ *be a feasible macro-merger in* $\mathcal{U}(\hat{G}(s))$ *and* $\mathcal{M}'$ *be a feasible macro-merger in the DAG that results from the execution of* $\mathcal{M}$. *Then,* $\mathcal{M}'' \equiv \mathcal{M} + \mathcal{M}'$ *is the macro-merger that is defined in* $\mathcal{U}(\hat{G}(s))$ *and is obtained by appending the edge sequence* $\mathcal{M}'$ *to the edge sequence* $\mathcal{M}$. *This definition extends naturally to sums with more than two terms.*

Clearly, Proposition 13 and its accompanying remarks extend to producer macro-mergers. Next, we present three particular types of macro-mergers that have a prominent role in the subsequent developments. The graphical structures that are involved in the definition of these macro-mergers are demonstrated in Figure 2. The first type is exemplified in the left part of Figure 2, and it reduces an in-tree of the DAG $\mathcal{U}(\hat{G}(s))$ to a single node; therefore, we call it a *(feasible) in-tree-based merger*.

Definition 15. *A feasible macro-merger* $\mathcal{M}$ *whose edges induce an in-tree in the underlying DAG* $\mathcal{U}(\hat{G}(s))$ *is a* feasible in-tree-based merger *(or simply a* t-merger*). The corresponding in-tree is denoted by* $\mathcal{T}(\mathcal{M})$, *and the root node of this in-tree is denoted by* $r(\mathcal{M})$.

Each basic merger e in a t-merger $\mathcal{M}$ establishes a pass between two maximal chains of the underlying PDG $C(\hat{G}(s))$ as discussed in the explanation of Definition 12. Furthermore, the in-tree structure of the DAG $\mathcal{T}(\mathcal{M})$ that is merged by $\mathcal{M}$, implies that the development of these passes links the involved maximal chains of $C(\hat{G}(s))$ into even larger chains, but it does not lead to the formation of any new cycles in $C(\hat{G}(s))$. Hence, for a feasible t-merger $\mathcal{M}$,

$$\chi(n(\mathcal{M})) = \sum_{n' \in \mathcal{T}(\mathcal{M})} \chi(n') - \sum_{e \in \mathcal{T}(\mathcal{M})} w(e) \tag{4}$$

[6]The computation of the DAG that results from the execution of a feasible basic merger e or a feasible macro-merger $\mathcal{M}$ in a given DAG $\mathcal{U}(\hat{G}(s))$ is an elaborate process that is described at the end of this subsection.

Eq. 4 further implies that a feasible t-merger $\mathcal{M}$ is a producer macro-merger with respect to $r(\mathcal{M})$ if and only if

$$\sum_{n' \in \mathcal{T}(\mathcal{M})} \chi(n') - \sum_{e \in \mathcal{T}(\mathcal{M})} w(e) \;\geq\; \chi(r(\mathcal{M})) \quad \Longleftrightarrow \sum_{\substack{n' \in \mathcal{T}(\mathcal{M}) \\ n' \neq r(\mathcal{M})}} \chi(n') \;\geq\; \sum_{e \in \mathcal{T}(\mathcal{M})} w(e) \tag{5}$$

The next proposition establishes a necessary condition for the feasibility of a t-merger $\mathcal{M}$.

Proposition 16. *Let $\mathcal{M} = \langle e_1, e_2, \ldots, e_n \rangle$ be a feasible t-merger in a DAG $\mathcal{U}(\hat{G}(s))$. Then, there exists a permutation $\hat{\mathcal{M}} = \langle e_{[1]}, e_{[2]}, \ldots, e_{[n]} \rangle$ of $\mathcal{M}$ that (i) constitutes another t-merger for $\mathcal{T}(\mathcal{M})$, and (ii) is such that $e_{[1]}$ is incident to the root node $r(\mathcal{M})$ and each edge $e_{[i]}$, $i = 2, \ldots, n$, is incident to the root node of the in-tree that results from the execution of the macro-merger $\langle e_{[1]}, e_{[2]}, \ldots, e_{[i-1]} \rangle$ in $\mathcal{T}(\mathcal{M})$.*

Proof. By induction on the number of edges, n, in $\mathcal{M}$. For $n = 1$, the result holds trivially. Next, suppose that the result holds for t-mergers involving up to $n - 1$ edges, and consider a t-merger $\mathcal{M}$ with n edges. Let $e_{[1]} = e_i = (n', r(\mathcal{M}))$ be the first edge in $\mathcal{M}$ that is incident to $r(\mathcal{M})$ in $\mathcal{T}(\mathcal{M})$. Definition 12 and the in-tree structure of $\mathcal{T}(\mathcal{M})$ imply that $e_{[1]}$ is a feasible basic merger in $\mathcal{T}(\mathcal{M})$. Let $\mathcal{T}'$ be the in-tree resulting from the execution of $e_{[1]}$ in $\mathcal{T}(\mathcal{M})$. Also, set $\mathcal{M}' = \langle e_1, e_2, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n \rangle$. We claim that $\mathcal{M}'$ is a feasible t-merger of $\mathcal{T}'$. To see this, first notice that (i) the macro-merger $\mathcal{M}'' = \langle e_1, e_2, \ldots, e_{i-1} \rangle$ does not involve node $r(\mathcal{M})$, and, (ii) according to Eq. 3, the basic merger $e_{[1]}$ is a producer merger with respect to its tail node n'. Hence, the feasibility of the macro-merger $\mathcal{M}''$ in $\mathcal{T}(\mathcal{M})$ implies its feasibility in $\mathcal{T}'$. Furthermore, the execution of the macro-merger $\mathcal{M}''$ in $\mathcal{T}'$ results in the same DAG with the execution of the macro-merger $\langle e_1, e_2, \ldots, e_i \rangle$ in $\mathcal{T}(\mathcal{M})$. Hence, the feasibility of the remaining part of $\mathcal{M}'$ results from the feasibility of $\mathcal{M}$. Since $\mathcal{M}'$ contains $n - 1$ edges, by the induction hypothesis, there exists a permutation $\hat{\mathcal{M}}'$ of $\mathcal{M}'$ that satisfies the requirement of Proposition 16. The macro-merger $\langle e_{[1]} \rangle + \hat{\mathcal{M}}'$ provides the required permutation of $\mathcal{M}$. □

Next we focus on sub-DAGs $\mathcal{T}$ of a DAG $\mathcal{U}(\hat{G}(s))$ with an in-tree structure. It is clear from Proposition 16 that, for any such in-tree $\mathcal{T}$, the capacity $\chi(r(\mathcal{T}))$ of its root node $r(\mathcal{T})$ has a significant role for the existence of a feasible t-merger $\mathcal{M}$ that reduces $\mathcal{T}$ into a single node. On the other hand, Eq. 5 implies that this capacity plays no role in determining the producing nature of any such t-merger with respect to the root node $r(\mathcal{T})$. These remarks motivate the following definition.

Definition 17. *An in-tree $\mathcal{T}$ of a DAG $\mathcal{U}(\hat{G}(s))$ is a* potential producer with respect to the root node $r(\mathcal{T})$ *if it satisfies Eq. 5.*

The next lemma and its corollary establish some properties for (i) the class of in-trees in any given DAG $\mathcal{U}(\hat{G}(s))$ that are potential producers with respect to their root nodes, and (ii) the complement of this class.

Lemma 18. *Consider two potential producer in-trees $\mathcal{T}$ and $\mathcal{T}'$ in a DAG $\mathcal{U}(\hat{G}(s))$ with no common edges, and suppose that $\mathcal{T}'$ is rooted in a node n of $\mathcal{T}$. The in-tree $\mathcal{T}''$ obtained by augmenting in-tree $\mathcal{T}$ with in-tree $\mathcal{T}'$ is a potential producer with respect to $r(\mathcal{T})$.*

Proof. Since $\mathcal{T}$ and $\mathcal{T}'$ are potential producers with respect to their root nodes, we have that $\sum_{\substack{n' \in \mathcal{T} \\ n' \neq r(\mathcal{T})}} \chi(n') \;\geq\; \sum_{e \in \mathcal{T}} w(e)$ and $\sum_{\substack{n' \in \mathcal{T}' \\ n' \neq n}} \chi(n') \;\geq\; \sum_{e \in \mathcal{T}'} w(e)$. Adding these two inequalities, and considering the fact that $\mathcal{T}$ and $\mathcal{T}'$ have no common edges, we get $\sum_{\substack{n' \in \mathcal{T}'' \\ n' \neq r(\mathcal{T})}} \chi(n') \;\geq\; \sum_{e \in \mathcal{T}''} w(e)$, which implies that $\mathcal{T}''$ is a potential producer with respect to $r(\mathcal{T})$. □

In the sequel, the in-tree $\mathcal{T}''$ defined in Lemma 18 is called the *union* of the in-trees $\mathcal{T}$ and $\mathcal{T}'$. Also, a sub-tree $\mathcal{T}'$ of an in-tree $\mathcal{T}$, rooted at some node n of $\mathcal{T}$, is characterized as *maximal*, if for every node $n' \neq n$ in $\mathcal{T}'$, the tree $\mathcal{T}'$ also contains all the children of n' in $\mathcal{T}$. This concept is used in the next corollary of Lemma 18.

Corollary 19. *Consider an in-tree $\mathcal{T}$ of a DAG $\mathcal{U}(\hat{G}(s))$ that is not a potential producer with respect to $r(\mathcal{T})$. Then, there exists a node n' of $\mathcal{T}$ and a maximal sub-tree $\hat{\mathcal{T}}$ of $\mathcal{T}$ that is rooted at n' and contains no potential producers with respect to n'.*

Proof. If every sub-tree $\mathcal{T}'$ of $\mathcal{T}$ that is rooted at $r(\mathcal{T})$ is not a potential producer with respect to $r(\mathcal{T})$, then the condition of Corollary 19 is true with $n' = r(\mathcal{T})$. Otherwise, consider a sub-tree $\mathcal{T}'$ of $\mathcal{T}$ that is rooted at $r(\mathcal{T})$ and is a potential producer with respect to $r(\mathcal{T})$. Since, by the working hypothesis, $\mathcal{T}$ is not a potential producer with respect to node $r(\mathcal{T})$, there is a node n' of $\mathcal{T}'$ and a maximal sub-tree of $\mathcal{T}$ rooted at n' and having no common edges with $\mathcal{T}'$. Let us denote this sub-tree by $\mathcal{T}(n')$. If $\mathcal{T}(n')$ satisfies the condition of Corollary 19, then we are done. Otherwise, let $\mathcal{T}''$ be a sub-tree of $\mathcal{T}(n')$ that is rooted at n' and is a potential producer with respect to n'. According to Lemma 18, the in-tree $\mathcal{T}'''$ that is the union of $\mathcal{T}'$ and $\mathcal{T}''$ is a potential producer with respect to $r(\mathcal{T})$. Iterating the above argument, we can obtain a sequence of in-trees $\mathcal{T}^{(0)} = \mathcal{T}'$, $\mathcal{T}^{(1)} = \mathcal{T}''$, ..., each rooted at $r(\mathcal{T})$, and being a potential producer with respect to $r(\mathcal{T})$. Furthermore, the in-tree $\mathcal{T}^{(i-1)}$ is strictly contained in the in-tree $\mathcal{T}^{(i)}$. But then, the validity of Corollary 19 results from the working hypothesis and the finiteness of $\mathcal{T}$. □

Next, we define the second type of macro-merger considered in this section.

Definition 20. *A* feasible path-based merger *(or simply a* p-merger*) is a feasible t-merger where the in-tree $\mathcal{T}(\mathcal{M})$ is a single path $\pi = \langle n_l, \ldots, n_2, e_1, n_1 \rangle$.*

The structure of a p-merger is depicted in the middle part of Figure 2. Being a specialization of t-mergers, p-mergers inherit all the concepts and the properties that were presented for t-mergers. The next proposition is an immediate implication of Proposition 16 and Definition 12.

Proposition 21. *There exists a feasible p-merger merging path $\pi = \langle n_l, \ldots, n_2, e_1, n_1 \rangle$ into a single node $n(\pi)$ if and only if, for all $i = 1, \ldots, l-1$,*

$$\sum_{j=1}^{i} \chi(n_j) - \sum_{j=1}^{i} w(e_j) \geq 0\,. \tag{6}$$

□

Also, the next proposition is useful in the subsequent developments.

Proposition 22. *A p-merger merging path $\pi = \langle n_l, \ldots, n_2, e_1, n_1 \rangle$ is producer with respect to node n_l.*

Proof. According to (4), $\chi(n(\pi)) = \chi(n_l) + \sum_{j=1}^{l-1} \chi(n_j) - \sum_{j=1}^{l-1} w(e_j)$. By (6) we have that $\sum_{j=1}^{l-1} \chi(n_j) - \sum_{j=1}^{l-1} w(e_j) \geq 0$, and hence $\chi(n(\pi)) \geq \chi(n_l)$. □

The third type of macro-mergers concerns the reduction to a single node of subgraphs of the DAG $\mathcal{U}(\hat{G}(s))$ that possess the cyclical structure exhibited in the right part of Figure 2. We call them *cycle-generating mergers*. Each label p_i, $i = 1, \ldots, 2l$, in Figure 2 denotes a directed path in $\mathcal{U}(\hat{G}(s))$. The paths are mutually disjoint, i.e., they share no interior nodes. Suppose that all the paths p_i with odd index i are feasible p-mergers. Since the paths are mutually disjoint, all the corresponding p-mergers can be executed simultaneously. Let $\tilde{s}$ denote the state after the simultaneous execution of these p-mergers, and let $n_{i,i+1}$ denote the node resulting from the execution of the p-merger defined by path p_i. Then, these nodes together with the paths p_i with even index i define a directed cycle in $\mathcal{U}(\hat{G}(\tilde{s}))$. According to Definition 4, this cycle constitutes a chain in $\mathcal{U}(\hat{G}(\tilde{s}))$. Therefore, the original structure of the right part of Figure 2 reduces into a single node. The next definition formalizes the notion of the cycle-generating merger.

Definition 23. *The sequence $\vartheta = \langle n_1, p_1, n_2, \ldots, n_{2l}, p_{2l} \rangle$ in a DAG $\mathcal{U}(\hat{G}(s))$ defines a* cycle-generating merger *(or simply a* c-merger*) if it satisfies the following conditions:*

1. *For $i = 1, \ldots, 2l$, n_i are distinct nodes of $\mathcal{U}(\hat{G}(s))$.*
2. *For $i = 1, \ldots, 2l$, p_i are mutually disjoint directed paths of $\mathcal{U}(\hat{G}(s))$.*
3. *For i odd, p_i leads from node n_i to node n_{i+1} and defines a feasible p-merger $\mathcal{M}(p_i) = \langle e_{i,1}, \ldots, e_{i,q_i} \rangle$.*
4. *For $i \neq 2l$ even, p_i leads from node n_{i+1} to node n_i; p_{2l} leads from node n_1 to node n_{2l}.*

The c-merger defined by ϑ is obtained by the execution of the macro-merger $\sum_{i\ odd} \mathcal{M}(p_i)$ and the further condensation of the PDG $C(\hat{G}(\tilde{s}))$ of the traffic state $\tilde{s}$ that will result from the execution of this macro-merger. The single node resulting from the execution of this c-merger is denoted by $n(\vartheta)$.

In the following, we use $\mathcal{N}(\vartheta)$ to denote the set of all nodes of the subgraph of $\mathcal{U}(\hat{G}(s))$ corresponding to the label sequence ϑ; i.e., the set $\mathcal{N}(\vartheta)$ contains (i) nodes n_i, $i = 1, \ldots, 2l$, and (ii) all the interior nodes of the paths p_i, $i = 1, \ldots, 2l$. The next proposition establishes an important property of c-mergers.

Proposition 24. *Any c-merger $\vartheta = \langle n_1, p_1, \ldots, n_{2l}, p_{2l}\rangle$ defined in a DAG $\mathcal{U}(\hat{G}(s))$ is a producer merger; that is, for all $n \in \mathcal{N}(\vartheta)$, $\chi(n(\vartheta)) \geq \chi(n)$, where $\chi(n(\vartheta))$ denotes the capacity of the resulting node $n(\vartheta)$.*

Proof. We prove the result by showing that

$$\chi(n(\vartheta)) \geq \sum_{n\in\mathcal{N}(\vartheta)} \chi(n)\,. \tag{7}$$

To see that it is true, first notice that each agent a advanced to clear an odd-indexed path p_i is absorbed in the chain of some node n of p_i, reducing the capacity $\chi(n)$ by one unit. But the freed edge $\epsilon(a; s)$ belongs to the newly formed cycle, and hence it will be part of the capacity $\chi(n(\vartheta))$. □

In fact, inequality (7) can be strict, since the newly formed cycle might contain additional free edges of the condensation $\mathcal{C}(\hat{G}(s))$ contained in some nodes of $\mathcal{N}(\vartheta)$ that are not part of a directed cycle in $\mathcal{C}(\hat{G}(s))$. The additional capacity for the newly formed node $n(\vartheta)$ is detectable in the PDG $\mathcal{C}(\hat{G}(\tilde{s}))$ but it is not traceable in the DAGs that we use in the algorithm for the identification of the sought mergers. Therefore, we work with both of these representations of the traffic states in our algorithm.

More specifically, starting with a traffic state s, our algorithm generates the PDG $\mathcal{C}(\hat{G}(s))$ and the DAG $\mathcal{U}(\hat{G}(s))$ as described in Section 2.3. The DAG $\mathcal{U}(\hat{G}(s))$ provides the representation of the current state at any major iteration of the algorithm, and facilitates all the inference of the algorithm taking place during the iteration. However, any merger determined by this processing is executed in the PDG $\mathcal{C}(\hat{G}(s))$ corresponding to the current traffic state s. The PDG resulting from this execution is further reduced to its maximal chains by Algorithm 1. The returned PDG $\mathcal{C}(\hat{G}(\tilde{s}))$ is the condensation of the new state $\tilde{s}$ resulting from the executed merger. The algorithm also generates the DAG $\mathcal{U}(\hat{G}(\tilde{s}))$ from the PDG $\mathcal{C}(\hat{G}(\tilde{s}))$. The complete specification of these iterations is provided in the following.

3.2. Polynomial-time algorithm for a special case

In this subsection, we construct a polynomial-time algorithm deciding traffic-state liveness for the case where the DAG $\mathcal{U}(\hat{G}(s))$ of the evaluated traffic state s satisfies the following condition.

Condition 25. *The undirected graph induced by the DAG $\mathcal{U}(\hat{G}(s))$, denoted by $\widehat{\mathcal{U}(\hat{G}(s))}$, is a single bi-block.*

We start by showing the following result that applies to any DAG $\mathcal{U}(\hat{G}(s))$, irrespective of whether it satisfies Condition 25 or not, and establishes powerful mechanisms for the processing of these DAGs.

Proposition 26. *Consider a node n of the DAG $\mathcal{U}(\hat{G}(s))$ that corresponds to the evaluated traffic state s. Then the following statements hold true:*

1. *If node n is a terminal node of the DAG $\mathcal{U}(\hat{G}(s))$ and it has no incident edges corresponding to a feasible basic merger, then the considered state s is not live.*
2. *If node n is a terminal node of the DAG $\mathcal{U}(\hat{G}(s))$ and it has only one incident edge $e = (n', n)$ corresponding to a feasible basic merger, then the search process outlined in Section 2.4 can execute this merger without any need for backtracking.*

Proof. Part 1) results immediately from Proposition 9, the first part of Proposition 10, and the definition of a feasible basic merger in Definition 12. In order to establish part 2), notice that the basic merger defined by edge e is unavoidable in any merger sequence that satisfies the liveness criterion of Theorem 11. Since, according to (3), this merger is a producer with respect to node n', it can be executed at any point of the search process for the aforementioned merger sequence, with no need for backtracking. □

Clearly, part 1) of Proposition 26 is a powerful mechanism to detect non-liveness of a traffic state s through the examination of the DAG $\mathcal{U}(\hat{G}(s))$, or the DAGs derived from $\mathcal{U}(\hat{G}(s))$ through a sequence of nodal mergers that can be executed greedily. On the other hand, part 2) of this proposition, and Propositions 13 and 24, provide some criteria to identify mergers that are executable in a greedy manner. Next, we strengthen Proposition 26 for the subclass of DAGs that satisfy Condition 25. To this aim, we need to introduce new concepts on the considered DAGs.

Definition 27. *For a terminal node n of a DAG $\mathcal{U}(\hat{G}(s))$ and an incident edge e, we denote by $\mathcal{T}(n,e)$ the subgraph of $\mathcal{U}(\hat{G}(s))$ consisting of all the maximal directed paths π of $\mathcal{U}(\hat{G}(s))$ that satisfy the following two properties:*

1. *Every path π leads to node n via edge e.*
2. *At every interior node n' of path π, the only edge emanating from n' in $\mathcal{U}(\hat{G}(s))$ is the edge that belongs on π.*

We further denote by $\mathcal{T}(n)$ the union of $\mathcal{T}(n,e)$ over all edges e that are incident to node n.

The next result summarizes properties of the graph $\mathcal{T}(n,e)$.

Proposition 28. *Any subgraph $\mathcal{T}(n,e)$ of a DAG $\mathcal{U}(\hat{G}(s))$ has the following properties:*

1. *$\mathcal{T}(n,e)$ is a DAG, and n is its single terminal node.*
2. *Every interior node n' of $\mathcal{T}(n,e)$ – i.e., a node with an indegree and an outdegree greater than zero in $\mathcal{T}(n,e)$ – has an outdegree of one in $\mathcal{U}(\hat{G}(s))$.*
3. *If $\mathcal{U}(\hat{G}(s))$ satisfies Condition 25, then every source node n'' of $\mathcal{T}(n,e)$ – i.e., a node n'' with indegree equal to zero in $\mathcal{T}(n,e)$ – has an outdegree in $\mathcal{U}(\hat{G}(s))$ that is greater than one.* □

The following proposition will be useful.

Proposition 29. *For a terminal node n of a DAG $\mathcal{U}(\hat{G}(s))$, the DAG $\tilde{\mathcal{T}}(n)$ obtained from $\mathcal{T}(n)$ by removing all its source nodes is an in-tree rooted at node n.*

Proof. By Definition 27, there is a directed path leading from every node n' of $\tilde{\mathcal{T}}(n)$ to n. The definition further implies that the only nodes shared by two DAGs $\mathcal{T}(n,e)$ and $\mathcal{T}(n,e')$, for some $e \neq e'$, are source nodes of both of these DAGs. Since $\tilde{\mathcal{T}}(n)$ does not contain source nodes of $\mathcal{T}(n)$, Proposition 28 implies that every node n' of $\tilde{\mathcal{T}}(n)$ is connected to node n by a unique directed path leading from n' to n. Therefore, $\tilde{\mathcal{T}}(n)$ is an in-tree rooted at node n. □

Next, we address the following optimization problem.

Problem 30 (In-tree root capacity maximization)**.** *For the in-tree $\tilde{\mathcal{T}}(n)$ corresponding to a terminal node n of a DAG $\mathcal{U}(\hat{G}(s))$, let $Q(n)$ denote the class of the feasible t-mergers $\mathcal{M}$ in $\tilde{\mathcal{T}}(n)$ with the corresponding in-tree $\mathcal{T}(\mathcal{M})$ rooted at n. We want to compute a t-merger $\mathcal{M}^* \in Q(n)$ such that*

$$\mathcal{M}^* = \arg\max_{\mathcal{M}\in Q(n)} \chi(n(\mathcal{M}))$$

Since the set $Q(n)$ is finite, Problem 30 is well-defined and possesses a finite set of optimal solutions. Every t-merger $\mathcal{M}^*$ that solves Problem 30 is an *optimal t-merger*. The next proposition establishes some important properties for the t-mergers that constitute optimal solutions to Problem 30.

Proposition 31. *Let $\mathcal{M}^*$ denote an optimal t-merger for Problem 30. Also, let n' denote a non-zero indegree node of $\mathcal{T}(\mathcal{M}^*)$, e' denote an edge of $\mathcal{T}(\mathcal{M}^*)$ leading into node n', and $\mathcal{T}(n',e';\mathcal{M}^*)$ denote the maximal sub-tree of $\mathcal{T}(\mathcal{M}^*)$ that is rooted at node n' and has edge e' as the only edge incident to node n'. Then, $\mathcal{T}(n',e';\mathcal{M}^*)$ is a potential producer with respect to its root node n'.*

Furthermore, there is an optimal t-merger $\hat{\mathcal{M}}^*$ with the following additional property: For every edge e'' of $\tilde{\mathcal{T}}(n)$ that is not in $\mathcal{T}(\hat{\mathcal{M}}^*)$ and leads into a node n'' of $\mathcal{T}(\hat{\mathcal{M}}^*)$, the corresponding maximal sub-tree $\tilde{\mathcal{T}}(n'',e'')$ of $\tilde{\mathcal{T}}(n)$ contains no t-merger that is feasible in the in-tree resulting from the execution of $\mathcal{M}^*$ and producer with respect to node $n(\hat{\mathcal{M}}^*)$. An optimal t-merger $\hat{\mathcal{M}}^*$ satisfying this property is structurally maximal.

Proof. We prove the first part of Proposition 31 by contradiction. Hence, let $\mathcal{M}^*$ denote an optimal t-merger with its in-tree $\mathcal{T}(\mathcal{M}^*)$ containing a pair (n',e') such that the corresponding maximal sub-tree $\mathcal{T}(n',e';\mathcal{M}^*)$ of $\mathcal{T}(\mathcal{M}^*)$ is not a potential producer with respect to node n'. By Corollary 19, there is a maximal sub-tree $\mathcal{T}'$ of $\mathcal{T}(n',e';\mathcal{M}^*)$ that contains no potential producers with respect to its root node n''. Then, from Eq. 5, we have

$$\sum_{v\in\mathcal{T}'} \chi(v) - \sum_{e\in\mathcal{T}'} w(e) \; < \chi(n'')\,. \tag{8}$$

Let $\mathcal{M}'$ denote the subsequence of $\mathcal{M}^*$ obtained by removing from $\mathcal{M}^*$ the edges belonging in the sub-tree $\mathcal{T}'$. Since the sub-tree $\mathcal{T}'$ does not contain any potential producers with respect to its root node n'', $\mathcal{M}'$ is feasible. Furthermore, Eqs 4 and 8 imply that $\chi(n(\mathcal{M}')) > \chi(n(\mathcal{M}^*))$, which contradicts the presumed optimality of $\mathcal{M}^*$.

In order to prove the second part of the proposition, let $\mathcal{M}^*$ be an optimal t-merger, and suppose that there exist (i) a node n'' of $\mathcal{T}(\mathcal{M}^*)$ and (ii) an edge e'' of $\tilde{\mathcal{T}}(n)$ not belonging in $\mathcal{T}(\mathcal{M}^*)$ and leading into node n'', such that the maximal sub-tree $\tilde{\mathcal{T}}(n'', e'')$ of $\tilde{\mathcal{T}}(n)$ contains a t-merger m that is feasible in the in-tree resulting from the execution of $\mathcal{M}^*$ and producer with respect to node $n(\mathcal{M}^*)$. Then, the macro-merger $\mathcal{M}' = \mathcal{M}^* + m$ is a feasible t-merger in $\tilde{\mathcal{T}}(n)$ and $\chi(n(\mathcal{M}')) \geq \chi(n(\mathcal{M}^*))$. This inequality cannot be strict since it would contradict the presumed optimality of $\mathcal{M}^*$. But it can hold as equality. Then, $\mathcal{M}'$ is an alternative optimal solution of Problem 30 that subsumes $\mathcal{M}^*$. The optimal t-merger $\hat{\mathcal{M}}^*$ claimed in the second part of the proposition is obtained through the iterative execution of all such possible subsumptions. □

The next proposition has an inverse role to Proposition 31 and enables an algorithmic construction of structurally maximal, optimal t-mergers for Problem 30.

Proposition 32. *Consider the in-tree $\tilde{\mathcal{T}}(n)$ corresponding to a terminal node n of a DAG $\mathcal{U}(\hat{G})$, and a t-merger $\mathcal{M} \in Q(n)$ that satisfies the following two conditions:*

1. *For any non-zero indegree node n' of $\mathcal{T}(\mathcal{M})$ and edge e' of $\mathcal{T}(\mathcal{M})$ leading into node n', the maximal sub-tree of $\mathcal{T}(\mathcal{M})$ that is rooted at node n' and has edge e' as the only edge incident to node n', $\mathcal{T}(n', e'; \mathcal{M})$, is a potential producer with respect to node n'.*
2. *For every edge e'' of $\tilde{\mathcal{T}}(n)$ not belonging in $\mathcal{T}(\mathcal{M})$ and leading into a node n'' of $\mathcal{T}(\mathcal{M})$, the maximal sub-tree of $\tilde{\mathcal{T}}(n)$ rooted at node n'' and has edge e'' as the only edge incident to node n'', $\tilde{\mathcal{T}}(n'', e'')$, contains no t-merger that is feasible in the in-tree resulting from the execution of $\mathcal{M}$ and producer with respect to node $n(\mathcal{M})$.*

The t-merger $\mathcal{M}$ is an optimal and structurally maximal solution for the instance of Problem 30 defined by $\tilde{\mathcal{T}}(n)$.

Proof. It suffices to establish the optimality of $\mathcal{M}$, since, then, its structural maximality results immediately from Condition 2. We prove this result by contradiction. Hence, let m be a t-merger in $Q(n)$ with

$$\chi(n(m)) > \chi(n(\mathcal{M})) \tag{9}$$

From Proposition 16, there is a permutation $\hat{m}$ of m that is a feasible t-merger in $\tilde{\mathcal{T}}(n)$, and since m and $\hat{m}$ merge the same subtree of $\tilde{\mathcal{T}}(n)$, from Eqs 4 and 9 we have that $\chi(n(\hat{m})) = \chi(n(m)) > \chi(n(\mathcal{M}))$. When combined with Condition 1, the inequality of (9) implies that m contains at least one edge not belonging in $\mathcal{M}$. Let $\mathcal{L} = \langle e_1, e_2, \ldots, e_l \rangle$ denote the subsequence of $\hat{m}$ consisting of the edges not belonging in $\mathcal{M}$. Next we show that $\mathcal{M} + \mathcal{L}$ is a feasible t-merger in $\tilde{\mathcal{T}}(n)$.

We establish only the feasibility of the basic merger defined by edge e_1 in the in-tree that results from the execution of $\mathcal{M}$; the feasibility of the basic mergers e_i, $i = 2, \ldots, l$, in the corresponding in-trees, can be argued in a similar manner. So, let $\hat{m}_1$ denote the subsequence of $\hat{m}$ preceding edge e_1 in $\hat{m}$. Since $\hat{m}$ meets the structural condition of Proposition 16, $\hat{m}_1$ is in $Q(n)$, and e_1 is an edge incident to $n(\hat{m}_1)$ and defining a feasible basic merger in the in-tree that results from the execution of $\hat{m}_1$ in $\tilde{\mathcal{T}}(n)$. Since every edge appearing in $\hat{m}_1$ is included in $\mathcal{M}$, from Condition 1 we have that $\chi(n(\hat{m}^1)) \leq \chi(n(\mathcal{M}))$, and since the basic merger of e_1 is feasible in the in-tree induced by $\hat{m}_1$, it is also feasible in the in-tree induced by $\mathcal{M}$.

The t-merger $\mathcal{M} + \mathcal{L}$ contains all the edges contained in both sequences $\mathcal{M}$ and $\hat{m}$. When combined with Eqs 4 and 9, and Condition 1, this fact implies that $\chi(n(\mathcal{M}+\mathcal{L})) > \chi(n(\mathcal{M}))$. But the last inequality contradicts the presumed Condition 2 for $\mathcal{M}$, and establishes the optimality of $\mathcal{M}$. □

Algorithm 2 takes as input the in-tree $\tilde{\mathcal{T}}(n)$ corresponding to a terminal node n of the DAG $\mathcal{U}(\hat{G}(s))$, and computes a structurally maximal, optimal solution for the corresponding instance of Problem 30. The algorithm returns (a) the computed t-merger $\mathcal{M}^*$ and (b) the capacity $\chi(n(\mathcal{M}^*))$ of $n(\mathcal{M}^*)$.

More specifically, if the DAG $\tilde{\mathcal{T}}(n)$ is a single terminal node n, Algorithm 2 returns its capacity $\chi(n)$. Otherwise, Algorithm 2 goes into an iterative mode, and at each iteration, it searches among all edges incident to the terminal node $\tilde{n}$ of the running in-tree $\tilde{\mathcal{T}}$, for an edge e that is either a producer feasible basic merger, or a non-producer feasible

Algorithm 2 Searching the in-tree $\tilde{\mathcal{T}}(n)$ of a terminal node n of $\mathcal{U}(\hat{G}(s))$ for a t-merger $\mathcal{M}$ that is rooted at node n and maximizes $\chi(n(\mathcal{M}))$.

Input: The in-tree $\tilde{\mathcal{T}}(n)$.
Output: An edge-sequence $MSEQ$ defining the sought t-merger, and the capacity CAP of the resulting node.

Procedure: Marco-Merger($\tilde{\mathcal{T}}(n)$)

$MSEQ := \langle\rangle$;
if $\tilde{\mathcal{T}}(n)$ consists of node n only **then**
  $CAP := \chi(n)$;
  return $MSEQ$, CAP
end if
$\tilde{\mathcal{T}} := \tilde{\mathcal{T}}(n)$;
$\mathcal{E}(\tilde{n})$:= the set of edges incident to the terminal node $\tilde{n}$ of $\tilde{\mathcal{T}}$;
Mark all edges in $\mathcal{E}(\tilde{n})$ as `'UNPROCESSED'`;
while $\exists$ unprocessed edge in $\mathcal{E}(\tilde{n})$ **do**
  e := an unprocessed edge $(n', \tilde{n})$ in $\mathcal{E}(\tilde{n})$;
  if e is a producer feasible basic merger **then**
    $MSEQ := MSEQ + \langle e\rangle$;
    Execute in $\tilde{\mathcal{T}}$ the merger specified by e;
    $\mathcal{E}(\tilde{n})$:= the set of edges incident to the terminal node $\tilde{n}$ of $\tilde{\mathcal{T}}$;
    Mark all edges in $\mathcal{E}(\tilde{n})$ as `'UNPROCESSED'`;
  else if e is a non-producer feasible basic merger **then**
    $\mathcal{G}$:= a copy of the sub-tree $\tilde{\mathcal{T}}(n')$ of $\tilde{\mathcal{T}}$ that is rooted in n', the tail node of edge e;
    In $\mathcal{G}$, set $\chi(n') := \chi(n') + \chi(\tilde{n}) - w(e)$;
    $(\mathcal{M}, \chi)$:= Marco-Merger($\mathcal{G}$);
    if $\chi \geq \chi(\tilde{n})$ **then**
      $MSEQ := MSEQ + \langle e\rangle + \mathcal{M}$;
      Execute the merger sequence $\langle e\rangle + \mathcal{M}$ in $\tilde{\mathcal{T}}$;
      $\mathcal{E}(\tilde{n})$:= the set of edges incident to the terminal node $\tilde{n}$ of $\tilde{\mathcal{T}}$;
      Mark all edges in $\mathcal{E}(\tilde{n})$ as `'UNPROCESSED'`;
    else
      Mark edge e as `'PROCESSED'`;
    end if
  else
    Mark edge e as `'PROCESSED'`;
  end if
end while
CAP := the capacity of the terminal node $\tilde{n}$ of $\tilde{\mathcal{T}}$;
return $MSEQ$, CAP

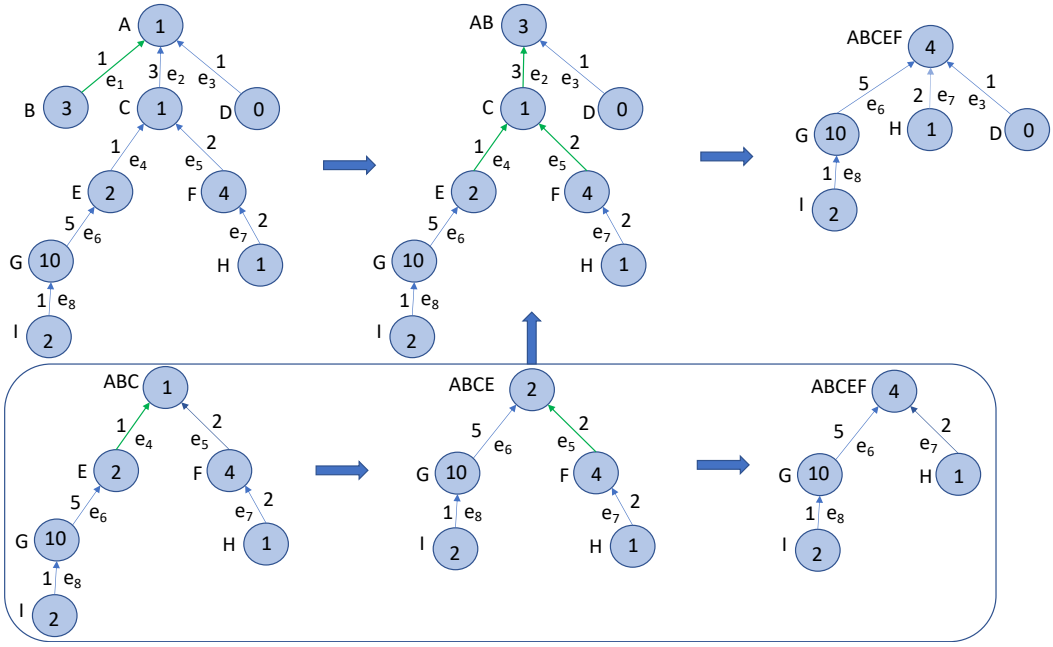


Figure 3: An example demonstrating the execution of Algorithm 2.

basic merger whose execution will enable a t-merger that is producer with respect to the terminal node $\tilde{n}$ of $\tilde{\mathcal{T}}$. Identified producer mergers are executed immediately and the algorithm starts a new iteration. Also, every producer (macro-)merger identified by the algorithm is appended to a merger sequence $MSEQ$ that eventually specifies the sought t-merger $\mathcal{M}^*$.

In order to assess whether the execution of a non-producer feasible basic merger corresponding to edge $e = (n', \tilde{n})$ will enable a producer t-merger with respect to node $\tilde{n}$, Algorithm 2 uses a recursion that executes on a copy of the sub-tree $\tilde{\mathcal{T}}(n')$ of the in-tree $\tilde{\mathcal{T}}$ that is rooted to node n'. Before running Algorithm 2 on the copy of $\tilde{\mathcal{T}}(n')$, the capacity of the root node n' of this sub-tree is updated to the capacity of the node n'' that results from the execution of the basic merger defined by edge e.

Algorithm 2 terminates when it is unable to detect an edge incident to the terminal node $\tilde{n}$ of the current in-tree $\tilde{\mathcal{T}}$ with the properties discussed above. At this point, the algorithm returns the sequence $MSEQ$ and the capacity $\chi(\tilde{n})$.

Example 33. We demonstrate the execution logic of Algorithm 2 by applying it to the in-tree depicted in the top-left part of Figure 3. The numbers in the nodes of this DAG are the corresponding nodal capacities and the numbers next to the edges are the edge weights.

During its first iteration, Algorithm 2 recognizes edge e_1 as a producer feasible basic merger, and the execution of this merger leads to the in-tree that is depicted in the top-middle part of Figure 3. This in-tree is processed during the second iteration of Algorithm 2. Edge e_2 is a feasible basic merger but not a producer. Hence, Algorithm 2 must examine whether the execution of this basic merger can result in a t-merger that is rooted at node AB, it is contained in the subtree that connects to node AB through edge e_2, and it is producer with respect to node AB. The execution of this test is depicted in the boxed part of Figure 3. Algorithm 2 first generates a copy of the sub-tree that is rooted at C, the tail node of edge e_2, and updates the capacity of the root node of this sub-tree to the capacity of the node that results from the execution of the basic merger corresponding to e_2. The resulting in-tree is the left in-tree depicted in the boxed part of Figure 3. Subsequently, Algorithm 2 executes itself recursively on this in-tree, and this execution first recognizes edge e_4 as a producer feasible basic merger among the edges that are incident to the root node ABC. The execution of this basic merger results in the in-tree depicted in the middle of the boxed part of Figure 3, and, in this in-tree, edge e_6 is an infeasible basic merger while edge e_5 is a producer feasible basic merger. The execution of the basic merger e_5 results in the third in-tree depicted in the boxed part of Figure 3. In this in-tree, edge e_6 is an infeasible

basic merger and edge e_7 is a feasible basic merger but not a producer. Hence, the recursion returns at this point, with a nodal capacity of 4 for the terminal node of the generated in-tree. Since 4 is larger than 3 – i.e., the capacity of the terminal node AB in the in-tree that originated this recursion – the edge sequence $\langle e_2, e_4, e_5\rangle$ constitutes a producer t-merger with respect to node AB. The execution of this t-merger results in the in-tree depicted at the top-right part of Figure 3. In this in-tree, edge e_6 is an infeasible basic merger, and edges e_7 and e_3 are feasible basic mergers but they cannot generate any t-merger that is producer with respect to the root node $ABCEF$. Hence, Algorithm 2 completes, returning $MSEQ = \langle e_1, e_2, e_4, e_5\rangle$ as the identified t-merger $\mathcal{M}^*$ and the capacity of the node $ABCEF$ as the value of CAP.

Before concluding this example, it is useful to make the following remarks. First, notice that the t-merger defined by the edge sequence $\langle e_2, e_4\rangle$ is not a producer t-merger with respect to node AB, since the capacity of the terminal node $ABCE$ of the in-tree resulting from this t-merger is $\chi(ABCE) = 2 < 3 = \chi(AB)$. On the other hand, the basic merger e_4 is necessary for enabling the producer basic merger e_5 that establishes eventually the producer t-merger that is generated by the executed recursion.

Also, if the capacity of node $ABCEF$ had ended up being greater than or equal to 5, Algorithm 2 would have continued its execution, entering the basic edges e_6 and e_8 in $MSEQ$ as producer basic mergers. Hence, the various maximal sub-trees of the in-tree $\mathcal{T}(\mathcal{M}^*)$ are built incrementally, as a union of smaller in-trees, during the computation of the algorithm. Furthermore, each such smaller in-tree is a potential producer with respect to its root node.

We now establish some properties of the edge-sequence $MSEQ$ computed by Algorithm 2, that also imply the correctness of Algorithm 2.

Proposition 34. *The edge sequence $MSEQ$ obtained from the execution of Algorithm 2 on an in-tree $\tilde{\mathcal{T}}(n)$, has the following properties.*

1. *It defines a feasible t-merger $\mathcal{M}$ in $\tilde{\mathcal{T}}(n)$ with the corresponding in-tree $\mathcal{T}(\mathcal{M})$ rooted at node n.*
2. *For any non-zero indegree node n' of $\mathcal{T}(\mathcal{M})$ and edge e' of $\mathcal{T}(\mathcal{M})$ leading into node n', the maximal sub-tree of $\mathcal{T}(\mathcal{M})$ that is rooted at node n' and has edge e' as the only edge incident to node n', $\mathcal{T}(n', e'; \mathcal{M})$, is a potential producer with respect to node n'.*
3. *For every edge e'' of $\tilde{\mathcal{T}}(n)$ not belonging in $\mathcal{T}(\mathcal{M})$ and leading into a node n'' of $\mathcal{T}(\mathcal{M})$, the maximal sub-tree of $\tilde{\mathcal{T}}(n)$ that is rooted at node n'' and has edge e'' as the only edge incident to node n'', $\tilde{\mathcal{T}}(n'', e'')$, contains no t-merger that is feasible in the in-tree resulting from the execution of $\mathcal{M}$ and producer with respect to $n(\mathcal{M})$.*
4. *$\mathcal{M}$ is a structurally maximal, optimal solution of the corresponding instance of Problem 30. Also, the returned value of CAP is the capacity $\chi(n(\mathcal{M}))$ of node $n(\mathcal{M})$.*
5. *$\mathcal{M}$ is a producer macro-merger.*

Proof. Property 1 can be established by a simple induction on the depth of the in-tree $\tilde{T}(n)$; we omit the details.

Property 2 results from (i) the tests in lines 10 and 18 of Algorithm 2, which ensure that every basic merger or t-merger $\mathcal{M}$ entering the list $MSEQ$ has an in-tree $\mathcal{T}(\mathcal{M})$ that is potential producer with respect to its root node, and (ii) Lemma 18.

To prove Property 3, first notice that in the in-tree resulting from the execution of $\mathcal{M}$ there is no edge e'' incident to node $n(\mathcal{M})$ and corresponding to a producer feasible basic merger, since this producer merger would have been detected and executed by Algorithm 2 during its last iteration. Next, suppose that there is a sub-tree $\tilde{\mathcal{T}}(n'', e'')$ of $\tilde{\mathcal{T}}(n)$ with (i) node n'' being merged into node $n(\mathcal{M})$, (ii) edge e'' leading into node $n(\mathcal{M})$, and (iii) $\tilde{\mathcal{T}}(n'', e'')$ itself containing a t-merger $\mathcal{M}'$ that is feasible in the in-tree resulting from the execution of $\mathcal{M}$ and producer with respect to node $n(\mathcal{M})$. According to Proposition 16, there is a permutation of $\mathcal{M}'$, $\hat{\mathcal{M}}'$, such that each executed basic merger in $\hat{\mathcal{M}}'$ is incident to the root of the running in-tree. Hence, edge e'' is a feasible basic merger with respect to $\chi(n(\mathcal{M}))$, and Algorithm 2 must have run the part that is defined by Lines 14 to 24 on edge e'' during its last iteration. The recursion executed by this part would have constructed an edge sequence m containing $\hat{\mathcal{M}}'$, possibly interleaved with other t-mergers that are producers with respect to their root nodes. This sequence would have been added to $\mathcal{M}$ in Line 19, resulting in a different outcome by the algorithm.

The first part of Property 4 results from Properties 2 and 3 and Proposition 32. The second part results from the fact that Algorithm 2 executes immediately every detected basic merger or t-merger that is appended to $MSEQ$.

To prove Property 5, we must show that, for every node ν in $\mathcal{T}(\mathcal{M})$, $\chi(n(\mathcal{M})) = \sum_{n' \in \mathcal{T}(\mathcal{M})} \chi(n') - \sum_{e \in \mathcal{T}(\mathcal{M})} w(e) \geq \chi(\nu)$. For any node ν in $\mathcal{T}(\mathcal{M})$, let $\pi(\nu)$ denote the path leading from ν to the root node of $\mathcal{T}(\mathcal{M})$. Then, the sum $\sum_{n' \in \mathcal{T}(\mathcal{M})} \chi(n') - \sum_{e \in \mathcal{T}(\mathcal{M})} w(e)$ can be organized as follows:

$$\sum_{\nu' \in \pi(\nu)} \sum_{\substack{e' \text{ leading into } \nu' \\ e' \notin \pi(\nu)}} \Big(\sum_{\substack{\nu'' \in \mathcal{T}(\nu', e'; \mathcal{M}) \\ \nu'' \neq \nu'}} \chi(\nu'') - \sum_{e \in \mathcal{T}(\nu', e'; \mathcal{M})} w(e) \Big) + \Big(\sum_{\nu' \in \pi(\nu)} \chi(\nu') - \sum_{e \in \pi(\nu)} w(e) \Big)$$

From Proposition 22, $\sum_{\nu' \in \pi(\nu)} \chi(\nu') - \sum_{e \in \pi(\nu)} w(e) \geq \chi(\nu)$. Also, the first term in the above expression is nonnegative due to Property 2 in Proposition 34 that was already established in this proof. Hence, $\mathcal{M}$ is indeed a producer with respect to ν. Since ν was arbitrarily chosen, Property 5 has been established. □

The next result discusses the complexity of Algorithm 2.

Proposition 35. *Algorithm 2 runs in time $O(|\mathcal{E}(\tilde{\mathcal{T}}(n))|^2)$, where $\mathcal{E}(\tilde{\mathcal{T}}(n))$ denotes the set of edges of $\tilde{\mathcal{T}}(n)$.*

Proof. Let $\mathcal{E}(\tilde{\mathcal{T}}(n)) = \nu$, and notice that every local operation of Algorithm 2 is of complexity $O(\nu)$. But to this cost, we need to add the cost of the recursive call on the in-tree $\mathcal{T}(n')$ that corresponds to the tail node n' of edge e.

For $\tilde{\mathcal{T}}(n)$ with a single layer of edges, the complexity of Algorithm 2 is bounded by $\nu + (\nu - 1) + \ldots + 1 = O(\nu^2)$. For $\tilde{\mathcal{T}}(n)$ that is a single path, the complexity of Algorithm 2 is $O(\nu)$, since the producer p-merger that maximizes the capacity of the resulting node, is detected during a single traversal of $\tilde{\mathcal{T}}(n)$.

For the general case, let $\bar{l}$ denote the largest indegree of $\tilde{\mathcal{T}}(n)$. Then, the computational cost of Algorithm 2 is bounded by the solution of the recursion $q(\nu) = \bar{l}^2 \cdot q(\nu/\bar{l}) + f(\nu)$, where the term $f(\nu) = O(\nu)$ and represents the cost pertaining to the local processing of the edges that belong in the top layer of edges of the in-tree $\tilde{\mathcal{T}}(n)$. The term $\bar{l}^2 \cdot q(\nu/\bar{l})$ accounts for the cost of the recursive invocation of Algorithm 2 during the processing of these edges. Since $f(\nu) = O(\nu^{\log_{\bar{l}} \bar{l}^2 - 1})$, from the part 1) of the Master Theorem in Appendix A, we have that $q(\nu)$ is $\Theta(\nu^{\log_{\bar{l}} \bar{l}^2}) = \Theta(\nu^2)$, and therefore, the complexity of Algorithm 2 is $O(\nu^2)$. □

The next definition is useful for the subsequent results.

Definition 36. *A terminal node n of a DAG $\mathcal{U}(\hat{G}(s))$* has maximal capacity *if the sequence $MSEQ$ returned by Algorithm 2 applied on $\tilde{\mathcal{T}}(n)$ is empty.*

For DAGs $\mathcal{U}(\hat{G}(s))$ that satisfy Condition 25, the concepts introduced by Definitions 27 and 36, and Algorithm 2, enable the strengthening of Proposition 26 as follows:

Proposition 37. *Consider a DAG $\mathcal{U}(\hat{G}(s))$ satisfying Condition 25, and a terminal node n of this DAG having maximal capacity. If, for every edge e incident to n in $\mathcal{U}(\hat{G}(s))$, there is no feasible t-merger in $\mathcal{T}(n, e)$ merging node n with a source node n'' of $\mathcal{T}(n, e)$, then the underlying traffic state s is not live.*

Proof. Consider an edge e incident to n in $\mathcal{U}(\hat{G}(s))$. By Proposition 28 and the working assumptions, any maximal t-merger rooted at node n in $\mathcal{T}(n, e)$, leads to a DAG $\mathcal{T}'$ having a unique terminal node n', where every edge e' incident to n' in $\mathcal{T}'$ corresponds to an infeasible basic merger. Furthermore, since the considered terminal node n has maximal capacity in $\mathcal{U}(\hat{G}(s))$, none of these basic mergers can become feasible by injecting further capacity into the aforementioned terminal node n' through the execution of a feasible macro-merger defined in the union of the remaining DAGs $\mathcal{T}(n, e'')$, for $e'' \neq e$; any such macro-merger either (a) would have been identified and executed by Algorithm 2, when applied to the corresponding in-tree $\tilde{\mathcal{T}}(n)$, or (b) would connect node n with a source node of the DAG $\mathcal{T}(n)$, which contradicts the assumptions of Proposition 37. Finally, since edge e was selected arbitrarily, it follows that n cannot be merged with any of the source nodes in $\mathcal{T}(n)$, and therefore, the underlying traffic state s cannot be live. □

Algorithm 3 tests the condition of Proposition 37 on $\mathcal{T}(n, e)$. It first assesses the feasibility of the basic merger defined by edge e. If e is not a feasible basic merger, then the algorithm deduces the lack of merger sequences over the nodes of $\mathcal{T}(n, e)$ that can merge n with a source node of $\mathcal{T}(n, e)$. Otherwise, it executes the basic merger e, resulting in the DAG $\mathcal{T}'$ with the terminal node n', and checks whether e has merged n with a source node of $\mathcal{T}(n, e)$. If so,

Algorithm 3 Determining the existence of a t-merger merging a terminal node n of maximal capacity in $\mathcal{U}(\hat{G}(s))$ with a source node in $\mathcal{T}(n, e)$.

Input: DAG $\mathcal{T}(n, e)$.
Output: A binary variable taking the value of 1 if there is a merger sequence with the desired property, and the value of 0, otherwise.

Procedure: MergingSource($\mathcal{T}(n, e)$)

if e is an infeasible basic merger **then**
  return 0;
end if
$\mathcal{T}'$:= the DAG resulting from the execution of the basic merger defined by e in $\mathcal{T}(n, e)$;
n' := the terminal node of $\mathcal{T}'$;
if the nodes merged in n' include a source node of $\mathcal{T}(n, e)$ **then**
  return 1;
end if
$\tilde{\mathcal{T}}'$:= the in-tree corresponding to the terminal node of DAG $\mathcal{T}'$;
$(\mathcal{M}, \chi)$:= result of running Algorithm 2 on $\tilde{\mathcal{T}}'$;
$\mathcal{T}''$:= the DAG resulting from the execution of the t-merger $\mathcal{M}$ in $\mathcal{T}'$;
for every edge e'' incident to the terminal node n'' of $\mathcal{T}''$ **do**
  if MergingSource($\mathcal{T}''(n'', e'')$) = 1 **then**
    return 1;
  end if
end for
return 0;

the algorithm exits with a positive outcome. Otherwise, it maximizes the capacity of the terminal node n' in $\mathcal{T}'$ by running Algorithm 2 on the corresponding in-tree $\tilde{\mathcal{T}}'$. The execution of the computed t-merger results in the DAG $\mathcal{T}''$. Subsequently, it recursively searches for a t-merger with the desired property in each of the sub-DAGs $\mathcal{T}''(n'', e'')$ defined by the DAG $\mathcal{T}''$ and the edges e'' incident to node n'' in $\mathcal{T}''$.

Example 38. We demonstrate Algorithm 3 on the DAG $\mathcal{T}(n, e)$ depicted by solid lines in Figure 4; dashed lines indicate the connections of this DAG to the rest of the underlying DAG $\mathcal{U}(\hat{G}(s))$. The numbers associated with nodes indicate the capacity of the nodes, and the numbers associated with edges e_i, $i = 1, \ldots, 4$, are the corresponding weights $w(e_i)$.

In the DAG $\mathcal{T}(n, e_1)$, edge e_1 defines a feasible basic merger. The execution of e_1 results in the DAG $\mathcal{T}'$ where nodes n and n_1 are merged into a new terminal node n' with capacity $\chi(n') = 0 + 2 - 1 = 1$. Algorithm 2 applied to $\mathcal{T}'$ returns the t-merger defined by edge e_2. The execution of this t-merger in $\mathcal{T}'$ results in the DAG $\mathcal{T}''$ with terminal node n'' of capacity $\chi(n'') = 2$ and incident edges e_3 and e_4. At this point, Algorithm 3 proceeds to the execution of the FOR-loop. The recursive call of Algorithm 3 on the DAG $\mathcal{T}''(n'', e_3)$ results in a positive outcome, which is propagated to the original thread of the algorithm. Thus, the algorithm returns a positive outcome for the considered DAG $\mathcal{T}(n, e)$.

Concluding the example, we point out that the t-merger constructed by Algorithm 3 for the DAG $\mathcal{T}(n, e)$ of Figure 4 is the edge-sequence $\langle e_1, e_2, e_3 \rangle$. The execution of this t-merger merges node n with the source node n_3 of $\mathcal{T}(n, e)$. Indeed, it can be seen in Figure 4 that edges e_1 and e_3 constitute a path leading from node n_3 to node n in $\mathcal{T}(n, e)$. But it is also important to notice that the insertion of the producer merger that corresponds to edge e_2 was essential for the enablement of the merger defined by edge e_3.

The correctness of Algorithm 3 follows from the above discussion and Proposition 16. The next result establishes the polynomial-time complexity of the algorithm with respect to the size of the input DAG $\mathcal{T}(n, e)$.

Proposition 39. *Algorithm 3 runs in time $O(|\mathcal{E}(\mathcal{T}(n, e))|^2)$, where $\mathcal{E}(\mathcal{T}(n, e))$ is the set of edges of the input DAG $\mathcal{T}(n, e)$.*

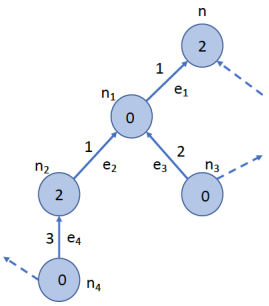


Figure 4: An example DAG $\mathcal{T}(n, e)$ for the demonstration of Algorithm 3.

Proof. Let $O(|\mathcal{E}(\mathcal{T}(n, e))|^2) = \nu$, and denote by $\bar{l}$ the maximum indegree of the input DAG $\mathcal{T}(n, e)$. We can bound the computational cost of Algorithm 3 by the solution of the recursion $q(\nu) = 1 + c\nu^2 + \bar{l}q(\nu/\bar{l})$, where the first two terms correspond to the execution of the basic merger e and the invocation of Algorithm 2 on the resulting DAG, respectively; in particular, the constant c that appears in the second term expresses the fact that the complexity of Algorithm 2 is $O(\nu^2)$. The third term represents the cost that results from the recursive invocation of Algorithm 3. Part 3) of the Master Theorem in Appendix A implies that $q(\nu) = \Theta(\nu^2)$, provided that we have a constant $d < 1$ such that $\bar{l} + c\nu^2/\bar{l} \leq d(1 + c\nu^2)$, for a sufficiently large ν. The existence of such a constant d can be shown as follows. First, notice that smaller values of $\bar{l}$ render easier the decision problem addressed by Algorithm 3. Hence, we consider ν larger than some value $\bar{\nu}_1$ so that we have $\bar{l} > c$. Then, we pick $d \in (c/\bar{l}, 1)$. The requirement $dc\nu^2 + d \geq \bar{l} + c\nu^2/\bar{l}$ can be reformulated as $c\nu^2(d - 1/\bar{l}) \geq \bar{l} - d$. By definition, $\bar{l} \geq 1$. Also, from the definition of c, we can assume that $c \geq 1.0$, which implies that $d > c/\bar{l} \geq 1/\bar{l}$. But then we can satisfy the inequality $dc\nu^2 + d \geq \bar{l} + c\nu^2/\bar{l}$ by picking $\nu \geq \sqrt{(\bar{l}^2 - d\bar{l})/(cd\bar{l} - c)} \equiv \bar{\nu}_2$. From the above, it follows that we can have the aforementioned constant d for $\nu \geq \max\{\bar{\nu}_1, \bar{\nu}_2\}$, and the proof is completed. □

Algorithm 3 can be easily modified to return, in the case of a positive outcome, the identified t-merger that led to this outcome, together with its binary outcome. The details of this modification are similar to the computation of $MS\,EQ$ by Algorithm 2, and they are left to the reader.

For the DAGs $\mathcal{U}(\hat{G}(s))$ satisfying Condition 25, Proposition 37 and Algorithm 3 enable the strengthening of the second part of Proposition 26 according to the following proposition.

Proposition 40. *Consider a DAG $\mathcal{U}(\hat{G}(s))$ satisfying Condition 25, and a terminal node n of $\mathcal{U}(\hat{G}(s))$ having maximal capacity. Suppose that there exists only one edge e incident to n in $\mathcal{U}(\hat{G}(s))$ with the corresponding DAG $\mathcal{T}(n, e)$ possessing a feasible t-merger merging node n with a source node n'' of $\mathcal{T}(n, e)$. Then, the search process outlined in Section 2.4 can execute greedily the basic merger defined by edge e.*

Proof. The arguments in the proof of Proposition 37 imply that any merger sequence satisfying Theorem 11 must contain a t-merger that is defined on the DAG $\mathcal{T}(n, e)$ and merges node n with a source node of $\mathcal{T}(n, e)$. From Proposition 16, such a t-merger can start with the basic merger defined by edge e. Finally, the maximal capacity of node n in $\mathcal{U}(\hat{G}(s))$ implies that the basic merger corresponding to e can be executed immediately by the underlying search process and with no need for backtracking. □

As in the case of Proposition 26, for DAGs $\mathcal{U}(\hat{G}(s))$ satisfying Condition 25, Propositions 37 and 40 enable a greedy advancement of the search for a merger sequence outlined in Section 2.4 by focusing on the terminal nodes of these DAGs. Next we establish that, for a DAG $\mathcal{U}(\hat{G}(s))$ that satisfies Condition 25 and has maximal capacities at its terminal nodes according to the criterion of Definition 36, the inability of the results of Propositions 37 and 40 to provide a greedy advancement of the underlying search process implies the existence of a producer macro-merger in the DAG $\mathcal{U}(\hat{G}(\tilde{s}))$. The proof of this result also provides an efficient algorithm for the identification of such a macro-merger. The producer property of this macro-merger implies that it can be executed greedily by the underlying search process. Hence, we have a complete efficient algorithm for conducting the search process for the merger sequences stipulated by Theorem 11 for any traffic state s with the DAG $\mathcal{U}(\hat{G}(s))$ that satisfies Condition 25.

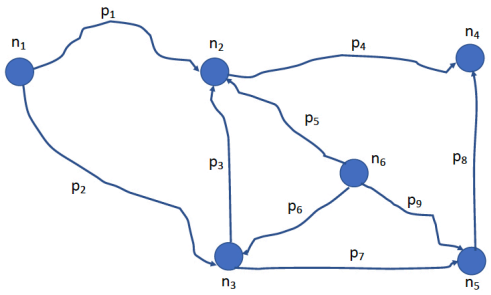


Figure 5: The path-based decomposition of the DAG $\mathcal{U}(\hat{G}(s))$ in the proof of Proposition 41.

Proposition 41. *Consider a DAG $\mathcal{U}(\hat{G}(s))$ satisfying Condition 25, where all of its terminal nodes have maximal capacities. Also, assume that every terminal node n of $\mathcal{U}(\hat{G}(s))$ has at least two incident edges e and e' with the corresponding sub-DAGs $\mathcal{T}(n,e)$ and $\mathcal{T}(n,e')$ possessing t-mergers that merge node n with a source node of these sub-DAGs. Then, the DAG $\mathcal{U}(\hat{G}(s))$ possesses a producer macro-merger.*

Proof. We prove the result by providing an algorithm for identifying a producer macro-merger in $\mathcal{U}(\hat{G}(s))$. The structure of the identified macro-merger is similar to that of the c-merger depicted in Figure 2. However, the simultaneously executed mergers providing this producer merger are not p-mergers as in the case of c-mergers, but t-mergers constructed by Algorithm 3 for some appropriately selected sub-DAGs $\mathcal{T}(n,e)$.

For the specification of the sought producer merger, we employ a path-based decomposition of the DAG $\mathcal{U}(\hat{G}(s))$ as indicated in Figure 5. More specifically, each directed path p recognized by this decomposition starts from some node n of $\mathcal{U}(\hat{G}(s))$ with an indegree of zero or a total degree greater than two, and extends up to a terminal node n' of $\mathcal{U}(\hat{G}(s))$ or a nonterminal node n'' with a total degree greater than two. The construction of the sought producer merger starts with any arbitrary path p from the aforementioned decomposition, considered as the path π_0 in this construction, and proceeds by distinguishing the following two cases.

Case 1: The head node v_1 of path π_0 is a nonterminal node of $\mathcal{U}(\hat{G}(s))$ with total degree greater than two. Then, there exists an edge e_1 emanating from node v_1, which defines a new path π_1 in the decomposition of $\mathcal{U}(\hat{G}(s))$ that can extend path π_0 while preserving the direction of this path. This case is exemplified by the triplet $\langle p_1, n_2, p_4\rangle$ in the DAG depicted in Figure 5.

Case 2: The head node v_1 of path π_0 is a terminal node of $\mathcal{U}(\hat{G}(s))$. Then, since every terminal node n of $\mathcal{U}(\hat{G}(s))$ has at least two incident edges e and e' with the corresponding sub-DAGs $\mathcal{T}(n,e)$ and $\mathcal{T}(n,e')$ possessing a t-merger that merges node n with a source node of these sub-DAGs, it follows that (a) node v_1 has an incident edge e_1 that is not part of path π_0, (b) the corresponding sub-DAG $\mathcal{T}(v_1,e_1)$ possesses a feasible t-merger that merges node v_1 with a source node v_2 of $\mathcal{T}(v_1,e_1)$, and (c) node v_2 has an edge e_2 emanating from it that is not part of the aforementioned merger sequence of $\mathcal{T}(v_1,e_1)$. Edge e_2 is the first edge of a directed path π_1 in the path-based decomposition of the DAG $\mathcal{U}(\hat{G}(s))$, and this path has a consistent sense of direction with the original path π_0. Hence, path π_0 can be extended through path π_1 in a way that preserves its direction, after executing the aforementioned merger sequence merging nodes v_1 and v_2.

The second case can be exemplified in Figure 5 by selecting path p_4 as the path π_0. Then, the role of node v_1 is played by node n_4, and the corresponding sub-DAG $\mathcal{T}(v_1,e_1)$ is the sub-DAG that consists of the paths p_8, p_7, and p_9. Hence, in this example, the source nodes of $\mathcal{T}(v_1,e_1)$ are nodes n_3 and n_6. Furthermore, assuming that node n_4 is connected to source node n_3 by the merger sequence computed by applying Algorithm 3 on $\mathcal{T}(v_1,e_1)$, the path π_1 that extends path p_4 is path p_3. Finally, we notice that the t-merger constructed by Algorithm 3 clears the path $\langle n_3, p_7, n_5, p_8, n_4\rangle$, but it might also contain a sub-path of path p_9 which is a producer feasible p-merger with respect to node n_5 and it is necessary for the enablement of the p-merger corresponding to path p_7.

The path-construction defined above can be repeated for the head node of path π_1 identified in each of the two cases. This fact enables the further extension of the constructed path. Since the DAG $\mathcal{U}(\hat{G}(s))$ has a finite number of nodes, this construction continues until we eventually reach a node that is on the already constructed path. At this point, the constructed path contains a cycle c.

In order to demonstrate the formation of this cycle c in the DAG of Figure 5, suppose that we started the construction with path p_1 as the initial path π_0, and advanced through node n_2 to path p_4. This path leads to node n_4, which is a terminal node, and hence execution of Algorithm 3 on the DAG $\mathcal{T}(n, e)$ consistsing of the paths p_8, p_7, and p_9, provides a t-merger for merging node n_4 with node n_3. From node n_3, the construction advances through path p_3. But path p_3 leads to node n_2, which was already visited, and therefore, the identified cycle c is defined by the paths p_4 and p_3, and the t-merger that was executed when visiting node n_4.

Next, we focus on the set of the macro-mergers that are necessary for the construction of cycle c. As demonstrated in the above example, these macro-mergers are the t-mergers identified by Algorithm 3 at the visited nodes ν_i that satisfy the conditions of Case 2, during the overall construction, and belong on cycle c. From the specification of all these t-mergers, it is clear that they involve the nodal capacities of non-overlapping subsets of nodes of the underlying DAG $\mathcal{U}(\hat{G}(s))$; therefore, they can be executed simultaneously. We claim that the simultaneous execution of all these t-mergers defines a producer macro-merger $\mathcal{M}$.

The validity of this claim can be established by showing that the basic mergers in $\mathcal{M}$ can be partitioned so that each subset in this partition defines a producer merger. From the above definition of $\mathcal{M}$, each basic merger that is in $\mathcal{M}$ belongs to a t-merger $\mathcal{M}(\nu_i)$ merging some terminal node ν_i belonging to the constructed cycle c to a source node n'_i of one of the sub-DAGs $\mathcal{T}(\nu_i, e)$ of ν_i. According to the discussion of Algorithm 3 that generates the merger sub-sequences $\mathcal{M}(\nu_i)$, each edge e' belonging in $\mathcal{M}(\nu_i)$ is either (i) in the directed path $p(\nu_i)$ leading from node n'_i to node ν_i in $\mathcal{T}(\nu_i, e)$, or (ii) in another producer merger of $\mathcal{T}(\nu_i, e)$ that is necessary for rendering path $p(\nu_i)$ a feasible p-merger. Clearly, any edge e' belonging in the second category satisfies the aforestated condition for $\mathcal{M}$. Furthermore, the clearance of the paths $p(\nu_i)$, for all the terminal nodes ν_i belonging in c, together with the paths π_i of the above construction that belong in c, define a c-merger similar to that of Figure 2. Hence, the edges e' belonging in the paths $p(\nu_i)$ are also part of a producer macro-merger, and Proposition 41 has been established. □

Algorithm 4 is a complete formal statement of the construction procedure that was used in the proof of Proposition 41. The next proposition characterizes the complexity of this algorithm.

Proposition 42. *Algorithm 4 runs in time $O(|\mathcal{E}|^2)$, where $\mathcal{E}$ is the set of edges of the input DAG $\mathcal{U}(\hat{G}(s))$.*

Proof. Each path π is constructed by Algorithm 4 through a simple forward-reaching scheme that starts from the initial edge e and advances towards its head node. Hence, this construction is performed in time $O(|\mathcal{E}(\pi)|)$, where $\mathcal{E}(\pi)$ denotes the number of edges in path π. Since the constructed paths π have no common edges, the entire set of these paths developed by Algorithm 4, is constructed in time $O(|\mathcal{E}|)$. From Proposition 39, the construction of the t-merger m at each visited terminal node ν is performed in time $O(|\mathcal{E}(\mathcal{T}(\nu, e'))|^2)$, where $\mathcal{E}(\mathcal{T}(\nu, e'))$ is the set of edges of the corresponding DAG $\mathcal{T}(\nu, e')$. But since the sub-DAGs $\mathcal{T}(\nu, e')$ have no overlapping edges, the entire set of t-mergers developed by Algorithm 4, is constructed in time $O(|\mathcal{E}|^2)$. The thinning of the lists L and $\mathcal{M}$ taking place in lines 23 – 30 can be performed through a synchronized forward scanning of these two lists, and it is performed in time $O(|\mathcal{N}|)$, where $\mathcal{N}$ is the set of nodes of the DAG $\mathcal{U}(\hat{G}(s))$. But the presumed connectivity of DAG $\mathcal{U}(\hat{G}(s))$ implies that $|\mathcal{N}|$ is $O(|\mathcal{E}|)$. Hence, the time complexity of the entire algorithm is $O(|\mathcal{E}|^2)$. □

As already noticed, when combined with Propositions 37 and 40, Proposition 41 enables an efficient algorithm for assessing the liveness of any traffic state s that satisfies Condition 25; the pseudocode of this algorithm is provided in Algorithm 5. Algorithm 5 takes as input the PDG $\hat{G}(s)$ that represents the evaluated traffic state s. It first computes the condensation $\mathcal{C}(\hat{G}(s))$. If the PDG $\mathcal{C}(\hat{G}(s))$ constitutes a single chain, Algorithm 5 deduces the liveness of the considered state s; cf. Theorem 11. Otherwise, Algorithm 5 iteratively computes the DAG $\mathcal{G}$ that corresponds to the current condensation C, and uses this DAG in order to either infer the non-liveness of state s, on the criteria established in Propositions 26 and 37, or identifies a merger in $\mathcal{G}$ that is executed in the PDG C maintained by the algorithm. At each iteration, the selected mergers are such that there is no need for backtracking on them. Finally, the algorithm exits the WHILE-loop with a positive outcome, if the aforementioned mergers have led to a PDG C that consists only of a single chain.

The next theorem establishes the correctness of Algorithm 5.

Theorem 43. *When executed on a traffic state s coming from an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system, and having a DAG $\mathcal{U}(\hat{G}(s))$ that satisfies Condition 25, Algorithm 5 terminates in finite time, and returns a correct assessment of the liveness of s.*

Algorithm 4 The macro-merger construction described in the proof of Proposition 41.

Input: A DAG $\mathcal{U}(\hat{G}(s))$ with every terminal node n having at least two incident edges e and e' with the corresponding sub-DAGs $\mathcal{T}(n, e)$ and $\mathcal{T}(n, e)$ possessing t-mergers merging node n to a source node of these sub-DAGs.
Output: The constructed macro-merger $\mathcal{M}$.

$L := \langle\rangle$; $\mathcal{M} := \langle\rangle$;
$\nu :=$ A node n of $\mathcal{U}(\hat{G}(s))$ with indegree of 0 or a total degree greater than 2;
Append ν to the end of L;
$e :=$ an edge emanating from ν;
$\pi :=$ the directed path of $\mathcal{U}(\hat{G}(s))$ starting with e and leading up to the first encountered node that is a terminal node of $\mathcal{U}(\hat{G}(s))$ or a nonterminal node with total degree greater than 2;
$\nu :=$ the head node of the path $\pi(e)$;
while $\nu \notin L$ **do**
    Append ν to the end of L;
    if ν is a nonterminal node of $\mathcal{U}(\hat{G}(s))$ **then**
        $e :=$ an edge emanating from ν;
        $\pi :=$ the directed path of $\mathcal{U}(\hat{G}(s))$ starting with e and leading up to the first encountered node that is a terminal node of $\mathcal{U}(\hat{G}(s))$ or a nonterminal node with total degree greater than 2;
        $\nu :=$ the head node of the path $\pi(e)$;
    else
        $e' :=$ an edge incident to node ν, different from e, and with the corresponding subgraph $\mathcal{T}(\nu, e')$ possessing a t-merger merging node ν with a source node of this subgraph;
        $m :=$ the t-merger returned by Algorithm 3 when applied on $\mathcal{T}(\nu, e')$;
        Add m in $\mathcal{M}$ labeled by ν;
        $\nu' :=$ the source node of $\mathcal{T}(\nu, e')$ merged with node ν by m;
        $e :=$ an edge emanating from ν' and not belonging into $\mathcal{T}(\nu, e')$;
        $\pi :=$ the directed path of $\mathcal{U}(\hat{G}(s))$ starting with e and leading up to the first encountered node that is a terminal node of $\mathcal{U}(\hat{G}(s))$ or a nonterminal node with total degree greater than 2;
        $\nu :=$ the head node of the path $\pi(e)$;
    end if
end while
$\nu' :=$ first element of L;
while $\nu' \neq \nu$ **do**
    Remove ν' from L;
    if ν' is a terminal node of $\mathcal{U}(\hat{G}(s))$ **then**
        Remove from $\mathcal{M}$ the t-merger m labeled by ν';
    end if
    $\nu' :=$ first element of L;
end while
return $\mathcal{M}$;

Algorithm 5 Assessing the liveness of a traffic state s coming from an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system, and having a DAG $\mathcal{U}(\hat{G}(s))$ that satisfies Condition 25.

Input: The PDG $\hat{G}(s)$ of a traffic state s.
Output: `TRUE` if and only if the traffic state s is live.

$C := C(\hat{G}(s))$;
while `TRUE` **do**
  if C is a single-chained PDG **then**
    return `TRUE`;
  end if
  $\mathcal{G} := \mathcal{U}(C)$;
  if $\exists$ a terminal node n of $\mathcal{G}$ with $\chi(n) < w(n', n)$ for every (n', n) in $\mathcal{G}$ **then**
    return `FALSE`;
  else if $\exists$ a path π in DAG $\mathcal{G}$ leading into node n_h **then**
    Update C by executing the p-merger corresponding to path π and condensing further the resulting PDG;
  else if $\exists$ a terminal node n in $\mathcal{G}$ that does not have maximal capacity **then**
    Update C by executing the t-merger returned by Algorithm 2 on the in-tree $\tilde{\mathcal{T}}(n)$, and condensing further the resulting PDG;
  else if $\exists$ a terminal node n of $\mathcal{G}$ s.t. $\forall$ sub-DAG $\mathcal{T}(n, e)$ defined by an edge e that is incident upon n, $\nexists$ a merger sequence merging node n with a source node of $\mathcal{T}(n, e)$ **then**
    return `FALSE`;
  else if $\exists$ a terminal node n of $\mathcal{G}$ with only a single incident edge e such that the DAG $\mathcal{T}(n, e)$ possesses a merger sequence merging node n with a source node of $\mathcal{T}(n, e)$ **then**
    Update C by executing the basic merger corresponding to edge e and condensing further the resulting PDG;
  else
    Use Algorithm 4 in order to identify a producer merger in $\mathcal{G}$, execute this merger in C, and condense further the resulting PDG;
  end if
end while

Proof. At each iteration, Algorithm 5 either exits inferring the liveness or the non-liveness of the considered traffic state s, or it executes a nodal merger on the current DAG $\mathcal{G}$. Since the number of nodes of $\mathcal{U}(\hat{G}(s))$ is finite, and every executed merger reduces at least by one the number of nodes in the DAG $\mathcal{G}$, it follows that the number of iterations of Algorithm 5 is finite. Also, each operation executed by the algorithm takes place in finite time. Hence, the entire computation of Algorithm 5 takes place in finite time.

In order to argue the correctness of the outcome of Algorithm 5, first we notice that every merger executed by Algorithm 5 results in an updated DAG $\mathcal{G}$ that satisfies Condition 25; this is true since the executed mergers constitute connected sub-DAGs of the current DAG $\mathcal{G}$. But then, the correctness of Algorithm 5 is a straightforward implication of the propositions established in the previous parts. More specifically, the correctness of the inference that takes place in the first IF statement of the algorithm is the result of Theorem 11. The correctness of the first part of the second IF statement is implied by part 1) of Proposition 26. The correctness of the second part in the second IF statement is implied by the realization that the considered mergers are producer mergers since the capacity of the resulting node is ∞, and by Proposition 13. The correctness of the third part in the second IF statement is implied by the producer nature of the executed t-merger, established by Proposition 34. The correctness of the fourth part in the second IF statement is implied by Proposition 37 and the prior execution of the third part of this IF statement with a negative result (since we have advanced to its fourth part). The correctness of the fifth part in the second IF statement of Algorithm 5 is implied by Proposition 40 and the prior execution of the third part of this IF statement with a negative result. Finally, the feasibility of the last part of the second IF statement is implied by Proposition 41 and the prior execution of all the other parts of this IF statement with negative results. □

We now discuss the complexity of Algorithm 5.

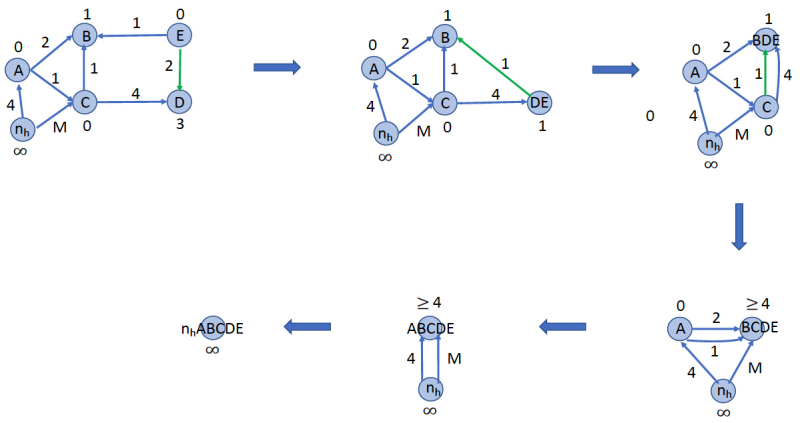


Figure 6: The DAGs $\mathcal{G}$ generated and processed by Algorithm 5 in Example 45.

Theorem 44. *For a guidepath network $G = (V, E)$, the time complexity of Algorithm 5 is $O(|E|^2 \cdot |V|)$.*

Proof. The construction and maintenance of the graphs C and $\mathcal{G}$ by Algorithm 5 has time complexity $O(|V| + |E|)$ per update. Furthermore, since the considered graphs C and $\mathcal{G}$ have no isolated vertices, we can restate this expression as $O(|E|)$. In each iteration, Algorithm 5 tests the feasibility of a basic merger in time $O(1)$, and hence the test in line 7 has complexity of $O(|E|)$. The search for feasible p-mergers performed in line 9 is of complexity $O(|E|)$, as well. Propositions 35 and 39 imply that lines 11, 13, and 15 can be executed in time $O(|E|^2)$. Finally, from Proposition 42, the complexity of line 18 is $O(|E|^2)$. Altogether, the time complexity of each iteration of Algorithm 5 is $O(|E|^2)$. Since every iteration reduces the number of nodes of the DAG $\mathcal{G}$ by at least one node, the time complexity of Algorithm 5 is $O(|E|^2 \cdot |V|)$. □

We conclude this subsection with an illustration of Algorithm 5.

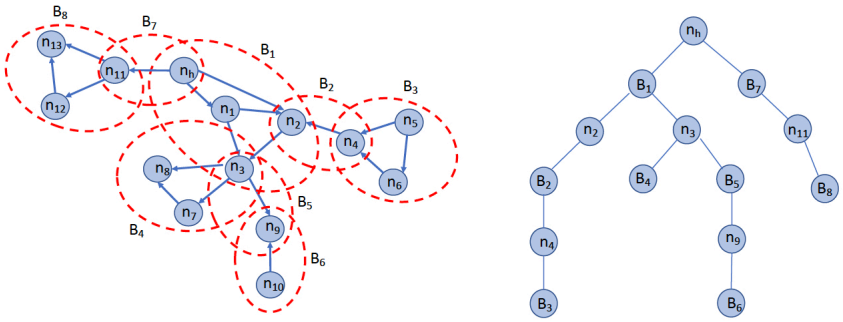
Example 45. We start with the DAG $\mathcal{U}(\hat{G}(s))$ depicted in the top-left part of Figure 6, and proceed in the clockwise mode indicated by the arrows. In each of the depicted DAGs, the number next to each node is the capacity of that node, while the number next to each edge is the weight of that edge. Node n_h indicates the home node with infinite capacity. The weight M of edge (n_h, C) is a sufficiently large value that we use to demonstrate certain effects in the subsequent discussion.

The initial DAG $\mathcal{G}$ has two terminal nodes, namely B and D, with maximal capacity; this is because each of the corresponding in-trees $\tilde{\mathcal{T}}(B)$ and $\tilde{\mathcal{T}}(D)$ is a single-node graph. Furthermore, node D has only one incident edge $e = (E, D)$ for which the corresponding sub-DAG $\mathcal{T}(D, e)$ has a feasible t-merger merging node D with a source node of the DAG $\mathcal{T}(D, e)$. Hence, Algorithm 5 will execute the basic merger (E, D), and this execution results in the second DAG $\mathcal{G}$ depicted in Figure 6.

This new DAG has only one terminal node, B. However, the capacity of B is not maximal. The in-tree $\tilde{\mathcal{T}}(B)$ consists of the single edge (DE, B), and the application of Algorithm 2 on $\tilde{\mathcal{T}}(B)$ returns the merger defined by edge (DE, B) as the macro-merger that maximizes the capacity of node B. The execution of this merger results in the third DAG $\mathcal{G}$ depicted in Figure 6.

The third DAG of Figure 6 has node BDE as its single terminal node. Node BDE has maximal capacity, since the corresponding in-tree $\tilde{\mathcal{T}}(BDE)$ has no edges. The edge $e = (C, BDE)$ of weight 1 is the only edge incident to node BDE that constitutes a feasible merger, and the sub-DAG $\mathcal{T}(BDE, e)$ consists only of the edge e since node C has a parallel edge $f = (C, BDE)$ of weight 4. Hence, Algorithm 5 proceeds to execute the merger defined by edge e.

The execution of this basic merger results in a cycle in the underlying PDG C, that is defined by (i) the undirected path connecting the chains of C that are contained in nodes C and BDE, and corresponding to the cleared edge e, and (ii) a second directed path that connects the same chains and corresponds to edge f. The newly formed cycle possesses, as part of its capacity, all the nodal capacity consumed by the mergers corresponding to the edges (E, D), (DE, B) and $e = (C, BDE)$, since these previously released edges are part of the newly formed cycle. This is recognized by the



a) The considered DAG and the decomposition of the underlying undirected graph to its blocks.

b) The block-cutpoint tree representing the block-based decomposition that is depicted in part (a)

Figure 7: The block-based decomposition of the DAG $\mathcal{U}(\hat{G}(s))$ that is used by Algorithm 6.

further condensation of the maintained PDG C, after the execution of the basic merger $e = (C, BDE)$ in C. Therefore, the capacity of node $BCDE$ resulting from the last merger is at least 4 (it might be even higher if there are additional free edges in the newly formed cycle that constitute part of the trees that define the u-components corresponding to the nodes B, C, D, and E of the initial DAG $\mathcal{G}$ depicted in top-left part of Figure 6).

In the next iteration, the algorithm processes the fourth DAG $\mathcal{G}$ of Figure 6. This DAG has node $BCDE$ as its single terminal node. The corresponding sub-DAG $\mathcal{T}(BCDE)$ consists of (i) the two edges leading from node A to node $BCDE$, and (ii) the edge leading from node n_h to node $BCDE$. Suppose that $M > \chi(BCDE)$. Then, node $BCDE$ has maximal capacity, but it fails the tests of Propositions 37 and 40, since both of the edges leading from node A to node $BCDE$ constitute the corresponding sub-DAGs $\mathcal{T}(BCDE, \cdot)$ and define feasible mergers. Hence, Algorithm 5 executes line 18. One possible execution of this part is to start with the path that is defined by edge (n_h, A), extend this path at node A by one of the two edges that emanate from it, and use the second of these edges at node $BCDE$ for determining the macro-merger that advances the pursued construction. But this last step leads back to node A, and to the detection of the c-merger that is defined by the two edges connecting nodes A and $BCDE$.

The execution of this c-merger leads to the fifth DAG $\mathcal{G}$ of Figure 6. The unique terminal node is $ABCDE$, and the corresponding in-tree $\tilde{\mathcal{T}}(ABCDE)$ consists of this node only; hence, node $ABCDE$ has maximal capacity. If $M > \chi(ABCDE)$, then the edge e with weight 4 is recognized as the only edge incident to the terminal node $ABCDE$ and its corresponding DAG $\mathcal{T}(ABCDE, e)$ possesses a feasible t-merger merging node $ABCDE$ with a source node of this DAG. The execution of the corresponding basic merger turns the second edge of the DAG into a self-loop in the node that results from this merger, and the condensation of the corrresponding PDG results in the single-node DAG depicted in the last part of Figure 6. If, on the other hand, $M \leq \chi(ABCDE)$, the terminal node $ABCDE$ has, for both of its incident edges, the corresponding DAGs $\mathcal{T}(ABCDE, e)$ possessing a feasible t-merger merging node $ABCDE$ with one of their source nodes. Hence, in this case, Algorithm 5 advances according to Proposition 41, and the detected macro-merger is the c-merger defined by the two DAG edges. The execution of this c-merger again leads to the single-node DAG depicted in the last part of the figure.

Concluding this example, we also notice that if the weight of the edge (C, B) in the top-left DAG $\mathcal{G}$ was equal to 2, instead of 1, then we would end up with the third DAG $\mathcal{G}$ depicted in Figure 6 having a single terminal node (node BDE) with all its incident edges corresponding to infeasible basic mergers. Hence, in this case, Algorithm 5 would terminate at this point with a negative outcome for the evaluated state s.

3.3. The general algorithm

In this section, we present a polynomial-time algorithm assessing the liveness of any traffic state s coming from an open, irreversible, dynamically routed, zone controlled guidepath-based transport system. This algorithm is enabled by (i) the developments in the previous parts, and (ii) the tree decomposition of any connected undirected graph to its blocks. Recall that the blocks of a connected undirected graph are its bridges and its maximal biconnected subgraphs.

Algorithm 6 Assessing the liveness of an arbitrary traffic state s coming from an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system.

Input: The PDG $\hat{G}(s)$ of traffic state s.
Output: Boolean variable $LIVE$ indicating whether the evaluated traffic state s is live.

$C := C(\hat{G}(s))$;
while $TRUE$ **do**
  if C is a single-chained PDG **then**
    $LIVE := TRUE$;
    return $LIVE$.
  end if
  $\mathcal{G} := \mathcal{U}(C)$;
  if $\exists$ a terminal node n of $\mathcal{G}$ with $\chi(n) < w(n', n)$ for every edge (n', n) in DAG $\mathcal{G}$ **then**
    $LIVE := FALSE$;
    return $LIVE$.
  else if $\exists$ an edge e in DAG $\mathcal{G}$ that defines a producer feasible basic merger **then**
    Update C by executing this basic merger and condensing further the resulting PDG;
  else if $\exists$ a path π in DAG $\mathcal{G}$ leading into node n_h **then**
    Update C by executing the p-merger corresponding to path π and condensing further the resulting PDG;
  else
    Compute the block-based decomposition of the undirected graph $\widehat{G}$ induced by the DAG $\mathcal{G}$ (e.g., using the algorithm of [34]), and the corresponding block-cutpoint tree Δ;
    Mark every block node in Δ as "UNPROCESSED";
    while $TRUE$ **do**
      B := a leaf node of Δ, or a block non-leaf node with all its children blocks marked as "PROCESSED";
      Process block B based on its classification according to Cases I-V;
      /* Every time the processing of such a block identifies and executes a (macro-)merger, a new iteration starts for the entire algorithm. Also, detection of non-liveness, and, therefore, a negative terminating decision by Algorithm 5 becomes a negative terminating decision for Algorithm 6.*/
    end while
  end if
end while

These blocks are connected through shared cut vertices. Furthermore, the block connectivity through the cut vertices can be represented by a tree known as the *block-cutpoint tree*.

Figure 7 demonstrates the concepts introduced in the previous paragraph, and furthermore, it highlights the way these concepts are used in the subsequent developments. In particular, part (a) depicts a DAG $\mathcal{U}(\hat{G}(s))$ and the block-based decomposition of the induced undirected graph. Part (b) presents the block-cutpoint tree that represents the block-based decomposition depicted in part (a). As discussed in section 2.1, the node set of this tree consists of two distinct subsets: the first subset corresponds to the articulation nodes of DAG $\mathcal{U}(\hat{G}(s))$, and the second subset corresponds to its blocks. The edges of this tree connect a node n_i of the first subset with a node B_j of the second subset; in particular, the edge corresponding to a pair $\{n_i, B_j\}$ is in the block-cutpoint tree if and only if the articulation node n_i belongs in block B_j. Furthermore, we treat the block-cutpoint tree induced by a DAG $\mathcal{U}(\hat{G}(s))$ as a *rooted tree*. The root is the tree node that corresponds to the home node n_h of the DAG $\mathcal{U}(\hat{G}(s))$, if node n_h is an articulation node; otherwise, the root is the tree node corresponding to the unique block that contains node n_h.

As already noticed, the lack of isolated vertices in the considered $\mathcal{U}(\hat{G}(s))$ DAGs implies that their blocks are either bridge-blocks or bi-blocks. The following definition establishes another classification of these blocks.

Definition 46. *A block B of a DAG $\mathcal{U}(\hat{G}(s))$ that is not the root node of the block-cutpoint tree, is* type-I *if its parent articulation node is a terminal node of the corresponding sub-DAG $\mathcal{G}_B$; otherwise, block B is characterized as* type-II.

With all the above concepts in place, we now proceed to the presentation of the algorithm. A pseudo-code of this algorithm is provided in Algorithm 6. The basic logic of the algorithm can be outlined as follows:

Similar to Algorithm 5, Algorithm 6 conducts a greedy search for a feasible merger sequence that can lead the underlying traffic system from the evaluated traffic state s to another traffic state s' for which the corresponding PDG $\hat{G}(s')$ is chained. Hence, in more operational terms, the algorithm takes as input the PDG $\hat{G}(s)$ of the evaluated traffic state s, computes the condensation $C(\hat{G}(s))$ of this PDG, and enters an iterative mode where the algorithm (i) either determines the liveness of the assessed state s, if the maintained PDG C has been reduced to a single chain; (ii) or determines the non-liveness of the assessed state s, if it can infer an inability to further merge certain parts of the currently processed PDG C and the corresponding DAG $\mathcal{G}$; (iii) or identifies a feasible (macro-)merger in the current DAG $\mathcal{G}$ that can be executed in the maintained PDG C.

A particular test employed by Algorithm 6 early in each of its iterations for the support of a type-(ii) decision, is the presence of a terminal node n in the evaluated DAG $\mathcal{G}$ with all of its incident edges corresponding to infeasible basic mergers. Also, some mergers that are easily identifiable and can be executed greedily by the algorithm in any of its iterations, are: (a) basic mergers that are producers, and (b) feasible p-mergers that involve the home node n_h, since any such merger results in a node with infinite capacity, and therefore it is a producer merger.

On the other hand, if Algorithm 6 goes through all the aforementioned easy checks during a given iteration without any success, then, it must employ the block-based decomposition of the current DAG $\mathcal{G}$ described above. During this stage of the algorithm execution, blocks are processed from the leaf nodes of the block-cutpoint tree towards its root, and the processing of each block B is differentiated on the basis of the following three attributes:

1. Whether it is (i) the root node or a child of the root node of the block-cutpoint tree, or (ii) any other node of this tree; i.e., this attribute considers the position of block B in the block-cutpoint tree.
2. Whether it is a bridge-block or a bi-block; i.e., this attribute considers the topological structure of the corresponding DAG $\mathcal{G}_B$.
3. Whether it is a type-I or type-II block, according to Definition 46; this attribute essentially defines a sense of orientation for block B, in the context of the block-cutpoint tree.

In agreement with the overall logic that drives the computation of Algorithm 6, the processing of a block B may (i) determine the non-liveness of the assessed state s, if it can infer an inability to further merge certain parts of the considered block, or (ii) identify a feasible (macro-)merger in the corresponding sub-DAG $\mathcal{G}_B$ that can be executed greedily. Both of these two outcomes advance the underlying search process; the first one by terminating the search process while inferring the non-liveness of the evaluated state s, and the second one by bringing the search process to a new traffic state s' and initiating a new iteration of Algorithm 6 that is performed on the DAG $\mathcal{U}(\hat{G}(\tilde{s}))$. But there is a third possible outcome for a block processing: in this case, no merger is currently possible in the considered block B, but further merging in this block can take place if additional capacity is conveyed to the block via its parent articulation node in the block-cutpoint tree. The mechanism for this capacity conveyance are the mergers that might take place during the processing of the blocks that are located on the path that links block B to the root of the block-cutpoint tree. This realization further implies that the processing of each block B must occur in a way that will not compromise the successful processing of the blocks that are located in that upper part of the block-cutpoint tree (with respect to the considered block B).

The insights and the remarks that are provided in the previous paragraph, are further operationalized through the integration of the following four rules in the logic of Algorithm 6:

Rule 47. *The current DAG $\mathcal{G}$ is processed on a block-by-block basis, and this processing advances from the leaf blocks of the block-cutpoint tree towards its root. In particular, a block that constitutes an internal node of the block-cutpoint tree, is picked for processing only after all the descendant blocks of this node have been processed.*

Rule 48. *A block B with its descendant blocks already processed, is processed in a way that either (i) detects a permanent inability to merge certain parts of the considered block, which further implies the non-liveness of the evaluated state s; or (ii) executes some basic merger that is unavoidable for the merging of the considered DAG $\mathcal{G}$ into a single node and can be executed at the current stage of the computation without any negative repercussions for the merging potential into a single node of the remaining nodes of block B; or (iii) tries to inject the maximum possible capacity to the articulation node n that is the parent node of block B in the block-cutpoint tree, in a way that does not compromise the potential of the sub-DAG corresponding to block B to be merged into a single node.*

Rule 49. *Every time that the capacity of an articulation node n is increased due to an executed merger, the entire sub-tree of the block-cutpoint tree that emanates from this node must be reprocessed, since this capacity increase might enable further developments in this sub-tree that might result in the injection of further capacity into node n.*

Rule 50. *The execution of a merger triggers the initiation of a new (major) iteration by Algorithm 6.*

As already noticed, the detailed implementation of the processing logic that is stipulated by Rules 47–50, during the processing of any given block B, is dependent upon the classification of block B in terms of the three attributes that are determined by the topological structure of the corresponding DAG $\mathcal{G}_B$, and the position and the orientation of block B in the block-cutpoint tree. The processing of bridge-blocks is strongly defined by the role of the corresponding edges as cut-edges of the underlying guidepath network G. On the other hand, the processing of the bi-blocks leverages the results and the insights that led to the development of Algorithm 5 in Section 3.2. Finally, for blocks B that are either the root node of the block-cutpoint tree or children of the root of this tree, the third part of Rule 48 can be relaxed, and this fact is reflected in the processing logic for this block class.[7]

Next, we detail the processing of the block classes in the aforementioned classification scheme that have a distinct role in the computation of Algorithm 6; these classes are discussed in increasing conceptual complexity of the involved processing.

I) Processing a type-II bridge-block B: The processing of a type-II bridge-block B that has all its descendant blocks already processed, evolves according to the following logic:

If the single edge that constitutes this block is a feasible basic merger, then this merger can be executed immediately, since (i) this is the only way to merge the sub-DAG corresponding to the sub-tree of the block-cutpoint tree rooted to block B, to the rest of the DAG $\mathcal{G}$, and (ii) a basic merger is always a producer merger with respect to the tail node of the corresponding edge (hence, this merger is consistent with the requirements of Rule 48).

On the other hand, if the single edge that constitutes the considered block B is an infeasible basic merger, then the merging of the aforementioned two parts of the DAG $\mathcal{G}$ cannot take place, and the algorithm terminates indicating the non-liveness of the assessed traffic state s.

II) Processing a type-I bridge-block B: If the basic merger that corresponds to this block is infeasible, then, the block is characterized as "PROCESSED" and the algorithm proceeds to the processing of another block, in expectation that the considered merger eventually becomes feasible if more capacity is injected to the parent articulation node of block B.

On the other hand, Rule 48 stipulates that the execution of a feasible basic merger corresponding to a type-I bridge-block B with all its descendant blocks already processed, must take place only if this merger does not decrease the capacity of the articulation node n that is the parent of block B in the block-cutpoint tree.

Indeed, if node n is mergeable with the home node n_h without the execution of the basic merger represented by the considered block B, then, the merger represented by block B is still feasible after nodes n and n_h have been merged. On the other hand, if the basic merger corresponding to block B reduces the capacity of the articulation node n, this reduction can have an adversarial effect on the merging potential of the blocks of the DAG $\mathcal{G}$ that do not belong in the sub-tree of the block-cutpoint tree that is rooted at block B. Hence, it is advantageous for the overall objective of the algorithm to postpone the execution of this merger.

However, since a basic merger is always a producer merger with respect to its tail node, Rule 49 implies that a proper assessment of the basic merger of block B as a producer or a consumer with respect to its parent node n, must take into consideration the repercussions of the execution of this merger for the sub-DAG of the DAG $\mathcal{G}$ that corresponds to the sub-tree rooted to the unique child node n' of block B in the block-cutpoint tree. More specifically, if $\chi(n) > w(n', n)$, then, the merger of the nodes n and n' may enable further mergers in this sub-DAG, that eventually lead to a capacity increase of the articulation node n'' that results from the merger of n and n'.

Practically, this possibility can be assessed through a simulation performed on a copy of the sub-tree of the block-cutpoint tree rooted at node n', where the original capacity $\chi(n')$ of node n' has been increased by $\chi(n) - w(n', n)$. The effect of this increase is assessed first on those blocks that are children of the articulation node n' in the block-cutpoint

[7] A DAG $\mathcal{U}(\hat{G}(\bar{s}))$ requiring a block-based processing by Algorithm 6, does not have any edges leading into the home node n_h, since the presence of these edges would have led to the identification of a producer merger before the algorithm reached this stage of its computation. The realization of this fact further implies that the differentiation of the block processing on the basis of their position in the block-cutpoint tree concerns primarily blocks of the bi-block type.

tree. But the processing of these blocks might result in a capacity increase of other articulation nodes that belong to these blocks, and this development triggers the processing of all the additional blocks that share these articulation nodes, according to the requirement that is posed by Rule 49. Repeating the previous remark to this new set of processed blocks, we can see that the effects of the original capacity increase at node n' propagates through the entire DAG that corresponds to the sub-tree of the block-cutpoint tree that is rooted at node n'.

Eventually, the contemplated merger defined by the edge (n', n) is executed on the original DAG $\mathcal{G}$ only if the re-evaluation of the aforementioned sub-tree, according to the logic described above, results in a new capacity value for node n' that exceeds the current capacity $\chi(n)$ of the articulation node n.[8]

III) Processing a type-II bi-block B that is (i) neither the root of the block-cutpoint tree, (ii) nor a child of the root of this tree, if this tree is rooted at node n_h: According to Definition 46, the parent articulation node n of block B is a non-terminal node of the sub-DAG $\mathcal{G}_B$ corresponding to block B. Block B is processed by running on it an adaptation of Algorithm 5, that is defined by the following modifications of the original algorithm:

A) According to Rule 50, each iteration of Algorithm 5 resulting in a merger in block B triggers a new iteration of Algorithm 6.

B) Node n (i.e., the parent articulation node of B) is treated by Algorithm 5 as possessing at least two emanating edges. In particular, if n has an outdegree of one in the DAG $\mathcal{G}_B$, we assume the presence of a fictitious edge $\tilde{e}$ that emanates from node n and leads to some dummy node $\tilde{n}$. Node $\tilde{n}$ does not appear in the performed computation. But the assumption of the edge $\tilde{e}$ ensures that node n is a source node for the sub-DAG $\mathcal{T}(n')$ of any terminal node n' that includes node n.

Treating node n as a source node for the sub-DAGs $\mathcal{T}(n')$, through the above modification, ensures that the identification of a lack of mergeability (and therefore, a non-liveness outcome) during the execution of Algorithm 5 on block B, is not rectifiable through the injection of further capacity in the articulation node n.

More specifically, if Algorithm 5 comes up with a non-liveness outcome during the execution of its current iteration on block B, the problematic terminal node n' that has incurred the non-liveness decision of Algorithm 5, must be different from the parent articulation node n of block B since node n is a non-terminal node of block B. Furthermore, the above modification of the algorithm execution implies that node n is not in the in-tree $\tilde{\mathcal{T}}(n')$ of the problematic node n'. Therefore, the injection of any additional capacity in node n is not able to rectify the experienced mergeability problems of node n', and Algorithm 6 exits correctly with a negative outcome for the evaluated state s.

Furthermore, we notice that the above modification does not compromise the actual merging potential, in the context of the DAG $\mathcal{G}_B$, of any node n' that has its corresponding DAG $\mathcal{T}(n')$ affected by this modification. This is true because the introduced modification preserves the merging potential of any such node n' with node n, and any connectivity of the node n' in the context of the DAG $\mathcal{G}_B$ that seems to be ignored by the effected pruning of the paths of the original DAG $\mathcal{T}(n')$ at node n, will be recognized in the subsequent iterations of the algorithm.

On the other hand, we must ensure that the presence of the fictitious edge $\tilde{e}$ does not confound the construction process defined in the proof of Proposition 41. This can be easily attained by stipulating that any invocation of the aforementioned construction always starts with an edge, and the corresponding path π_0, that emanates from node n. Since node n is a nonterminal node of DAG $\mathcal{G}_B$, such an edge always exists. Furthermore, any return of the construction to node n will result in a cycle detection, and therefore, there is no need for using the fictitious edge.

IV) Processing a type-I bi-block B that is (i) neither the root node of the block-cutpoint tree, (ii) nor a child of the root node of this tree, if this tree is rooted at node n_h: According to Definition 46, the parent articulation node n of this block is a terminal node of the corresponding sub-DAG $\mathcal{G}_B$. Block B is processed by a call of another adaptation of Algorithm 5, that is defined as follows:

A) Similar to the case of type-II bi-blocks, the execution of an iteration of Algorithm 5 resulting in a merger in block B triggers a new iteration of Algorithm 6.

B) For this block type, a case that needs special attention is that where Algorithm 5 must execute the next-to-last case in the second 'IF'-statement of its pseudo-code, and the parent articulation node n of block B is among the candidate nodes. Let e denote the unique incident edge upon node n, in block B, with the corresponding DAG $\mathcal{T}(n, e)$ possessing a merger sequence that merges node n to a source node of this DAG.

[8] The outlined simulation resembles the logic that is effected in Algorithm 2 for the identification of producer macro-mergers enabled by the initial execution of a feasible basic merger.

The aforementioned complication arises from the fact that edge e is not producer with respect to node n, since otherwise this merger would have been executed at an earlier stage of the current iteration of Algorithm 6. Hence, if Algorithm 5 proceeds to the execution of this basic merger but eventually fails to merge the entire block B into a single node, it is possible that the incurred reduction to the capacity of the parent articulation node of block B has an adversarial effect to the processing of the subsequent blocks that must be processed by Algorithm 6. In order to avoid such an adversarial effect, we stipulate the following:

B1) Algorithm 5 considers the basic merger defined by edge e as the merger to be executed, only if it has no other similar choice defined by another terminal node n' of block B.

B2) Furthermore, if edge e must be picked for processing, then the corresponding basic merger must be tested first in order to prevent a reduction in the capacity of the parent articulation node of block B, due to the reason discussed in the previous paragraphs. This test can be carried out by running a simulation of the further execution of Algorithm 6 on the sub-tree of the block-cutpoint tree rooted at block B, after the execution of the considered merger, as in the case of the processing of type-I bridge-blocks.

B3) The considered basic merger is executed by the algorithm, only if the aforementioned simulation results in no decrease of the capacity of the parent articulation node of block B. Otherwise, this merger is not executed, block B is characterized as "PROCESSED", and Algorithm 6 proceeds to the processing of another block of the underlying block-cutpoint tree.

C) If Algorithm 5 comes up with a non-liveness outcome during the execution of its current iteration on block B, then we must discern two cases:

C1) If the set of the problematic terminal nodes that has incurred the non-liveness decision of Algorithm 5, includes a node n' that is different from the parent articulation node n of block B, then, the situation is similar to the non-liveness outcomes of Case III above, and, therefore, the entire execution of Algorithm 6 will terminate with a negative outcome.

C2) On the other hand, if the problematic terminal node is the parent articulation node n of block B, then it is possible that the current mergeability problems of this node will be resolved through the injection of further capacity into node n, at a later stage of the overall execution of Algorithm 6. Hence, block B is characterized as "PROCESSED", and Algorithm 6 proceeds with the processing of another block.

Furthermore, we notice that, upon the termination of the execution of Algorithm 5 on block B along the lines of case C2), the capacity of the parent node n of this block is maximal according to Definition 36, and it is not possible to inject further capacity to this node through any additional mergers executed in block B. Hence, the specified processing of the considered block B is completely in agreement with the stipulations of Rules 47–50.

V) Processing a bi-block B that is (i) the root node of the block-cutpoint tree, or (ii) a child of the root node of this tree, if this tree is rooted at node n_h: In this case, block B is processed through a straightforward application of the original logic of Algorithm 5. More specifically, the requirement that the execution of a merger by Algorithm 5 in block B triggers a new iteration of Algorithm 6, still applies in order to abide by Rules 49 and 50. But there is no need for any further modifications of Algorithm 5 in this case. Furthermore, a non-liveness inference by Algorithm 5 is accepted unconditionally by Algorithm 6, since the above specifications for the processing of the remaining nodes of the block-cutpoint tree ensure that this conclusion was reached under the maximal possible injection of additional capacity in the children articulation nodes of block B.

Before we proceed with the analysis of Algorithm 6, we provide an example that demonstrates its logic.

Example 51. Consider a traffic state s with its DAG $\mathcal{U}(\hat{G}(s))$ depicted in the top left part of Figure 8. The numbers within the nodes report the nodal capacities, and the numbers next to each edge are the corresponding edge weights. The DAG $\mathcal{U}(\hat{G}(s))$ has node F as its only terminal node, and edge (D, F) is a feasible basic merger. Hence, Algorithm 6 cannot infer non-liveness for state s during its first iteration. Proceeding further in this iteration, the algorithm identifies edge (C, A) as a producer basic merger; in fact, this is the only such merger in this first DAG.

Executing the merger (C, A), we get the second DAG of Figure 8. On this new DAG, Algorithm 6 executes the merger defined by edge (B, n_h) as a maximal p-merger that leads into the home node n_h.

During its iteration on the third DAG of Figure 8, that results from the execution of the basic merger (B, n_h) on the second DAG, Algorithm 6 has to resort to the block-based decomposition of this DAG discussed above. The resulting blocks are highlighted in red in Figure 8. Also, the figure provides the corresponding block-cutpoint tree Δ.

The leaf block B_3 of this block-cutpoint tree consists of the single edge (E, D) and it is a type-I bridge-block. Since the basic merger defined by this edge is not a producer with respect to node D, this merger is not executed,

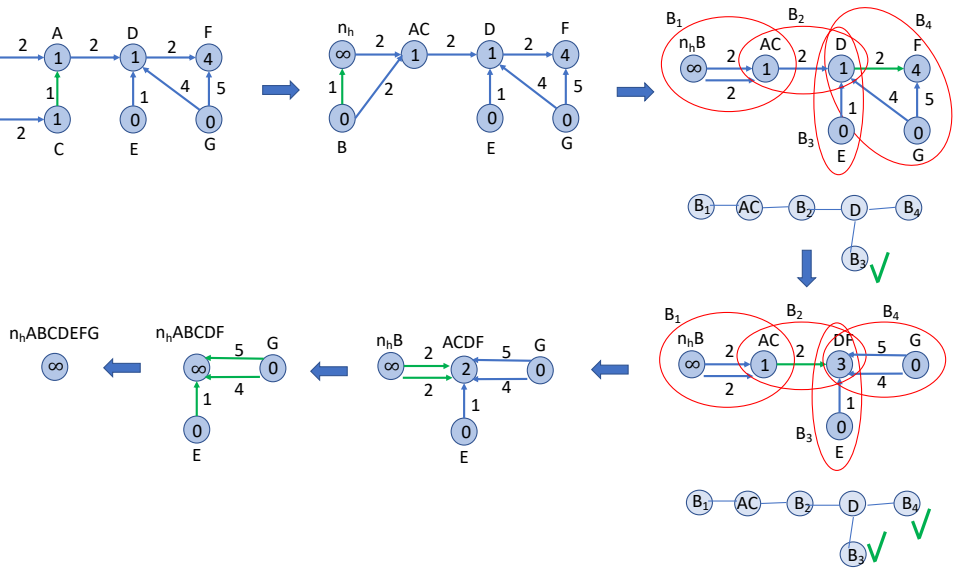


Figure 8: The DAGs generated and processed by Algorithm 6 in Example 51.

block B_3 is marked as "PROCESSED", and Algorithm 6 proceeds to the processing of block B_4, which is the second leaf block of Δ.

Block B_4 is a type-II bi-block. Executing the variation of Algorithm 5 defined in Case III-B, the edge (D, F) of this block defines the unique sub-DAG $\mathcal{T}(n, e)$, for the single terminal node F of block B_4, that possesses a merger sequence merging node F with a source node of this sub-DAG (which is the articulation node D, under the algorithm modification introduced in Case III-B).

The execution of the basic merger (D, F) leads to the fourth DAG $\mathcal{G}$ of Figure 8. Again, the processing of this DAG requires its block-based decomposition provided in Figure 8. The processing of the leaf block B_3 proceeds as in the processing of this block during the previous iteration. On the other hand, the new leaf block B_4 is a type-I bi-block, and its processing by Algorithm 5 reveals that the unique terminal node DF of this block has no incident edge defining a feasible basic merger. But since this terminal node is the parent articulation node of block B_4, the block is marked as "PROCESSED", and Algorithm 6 proceeds to the processing of the block B_2, which has both of its children blocks marked as "PROCESSED" by now.

Block B_2 is a type-II bridge-block, and the edge (AC, DF) that constitutes this block, is a feasible basic merger. Hence, this merger is executed immediately by Algorithm 6, providing the fifth DAG $\mathcal{G}$ in the depicted figure. In this new DAG, both edges leading from node n_hB to node $ACDF$ define producer feasible basic mergers. The execution of any of these two and the further condensation of the resulting PDG leads to the sixth DAG $\mathcal{G}$ depicted in Figure 8. All edges of this new DAG constitute paths leading into the home node n_h, and the execution of the corresponding mergers leads to the single-node DAG depicted in the last part of the figure. At this point, Algorithm 6 terminates with a positive outcome for the evaluated state s.

The next theorem establishes the correctness of Algorithm 6.

Theorem 52. *When applied on any traffic state s coming from an open, irreversible, dynamically routed, zone-controlled guidepath-based transport system, Algorithm 6 terminates in finite time, and returns a correct assessment of the liveness of s.*

Proof. Each iteration of Algorithm 6 either exits inferring liveness or non-liveness of traffic state s, or it executes a nodal merger on the current DAG $\mathcal{G}$. Also, the scanning of the block-cutpoint tree performed by the second WHILE-loop of Algorithm 6 is a finite operation. Hence, the finiteness of the computation of Algorithm 6 results from the finiteness of the number of nodes of the DAG $\mathcal{U}(\hat{G}(s))$, and the fact that every executed merger reduces at least by one the number of nodes in the resulting DAG $\mathcal{G}$.

The correctness of Algorithm 6 results from (i) the correctness of the various operations invoked by it, established in the previous parts of this work, and (ii) the fact that a negative outcome regarding the liveness of the evaluated state s based on the processing of some block node B of the underlying block-cutpoint tree Δ occurs only when block B has the capacity of all its critical articulation nodes maximized by a previous processing of the sub-trees of the tree Δ rooted at these articulation nodes. The last fact should be clear from the specification of Algorithm 6 through Rules 47–50 and the further specialization of these rules to the processing of the various block types discussed in Cases I-V. □

Finally, the next theorem establishes that the complexity of Algorithm 6 is polynomial with respect to the size of the elements that define the underlying guidepath network G.

Theorem 53. *For a guidepath network $G = (V, E)$, the time complexity of Algorithm 6 is $O(|E|^5 \cdot |V|^2)$.*

Proof. The construction and the maintenance of the graphs C and $\mathcal{G}$ employed by Algorithm 6 requires time $O(|V| + |E|)$. Also, the complexity of the algorithm in [34] that computes the block-based decomposition is $O(|E|^2)$. The processing of the conditions that appear in the first three parts in the second IF-statement in the external WHILE-loop of Algorithm 6 needs time $O(|E|)$. Also, during a single iteration of Algorithm 6, the processing of a bridge-block needs time $O(1)$, while the processing of a bi-block corresponds to a single iteration of Algorithm 5, which, according to the proof of Theorem 44, needs time $O(|E|^2)$. Hence, the local processing within a block that takes place during a single iteration of Algorithm 6 needs time $O(|E|^2)$.

But we must also account for the cost of the simulation that is executed during the processing of a type-I block. Under the top-down processing of the block-cutpoint tree suggested for these simulations, the number of blocks to be processed at the first pass is $O(|E|)$, since, in the worst case, every edge of the DAG $\mathcal{G}$ might define a separate block. Furthermore, the identification of an executable merger during such a simulation, might trigger the re-processing of the remaining blocks, and therefore, the total number of block processings that can take place during a simulation is $O(|E|^2)$. From Theorem 44, we can characterize the complexity of the processing of a single block during these simulations as $O(|E|^2 \cdot |V|)$. Hence, the complexity of running an entire simulation is $O(|E|^4 \cdot |V|)$, and this cost dominates the above cost of $O(|E|^2)$ that pertains to the local processing taking place within any given block.

Furthermore, during a single iteration, Algorithm 6 might have to process a number of blocks of the current block-cutpoint-tree until it either identifies an executable merger or determines the non-liveness of the considered state s, and, as discussed above, the number of blocks in the block-cutpoint tree is $O(|E|)$.

Finally, considering that every non-terminating iteration reduces the running DAG $\mathcal{G}$ by at least one node, the above remarks suggest the time complexity for Algorithm 6 of $O(|E|^5 \cdot |V|^2)$. □

3.4. Some experimental results

In this subsection, we report results from two numerical experiments that (i) assess the empirical performance of Algorithm 6 with respect to the required computational times, and (ii) highlight the mechanisms that provide the algorithm efficiency.

In the first experiment, we ran Algorithm 6 on traffic states s having a PDG $\hat{G}(s)$ that was constructed as follows: For each state s, we started with an $n \times n$ grid and removed a certain percentage p of its edges while ensuring that the resulting graph was connected and of minimal degree at least 2. We also chose another percentage q of the remaining edges in this graph, and turned them into directed edges, choosing each of the possible two directions for each edge with probability 0.5. Finally, we placed the home zone h either in the middle of the grid or at one of the four corners. The resulting graph is a planar graph, and its overall topological structure is consistent with the zoning schemes that define the edges and their connectivity in the guidepath networks G that are encountered in the targeted applications.

During the overall experiment, we let n range from 5 to 100 with a step of 5, $p \in \{0.0, 0.1, 0.2, 0.3, 0.4\}$,[9] $q \in \{0.25, 0.50, 0.75\}$, and for each possible parameterization of the tuple (n, p, q, home zone location), we ran 10 replications. Table 1 reports some indicative results regarding the performance of Algorithm 6 on the generated cases. More

[9] An $n \times n$ grid has $2n(n-1)$ edges. Also, among the connected graphs that are defined on the n^2 grid nodes and have a minimal degree of 2, the ones possessing the minimal number of edges are the cycles going through all these nodes. Such a cycle has n^2 edges. These facts imply an upper bound for p of $\bar{p}(n) = n^2/(2n^2 - 2n)$, with $\lim_{n\to\infty} p(n) = 0.50$.

Table 1: Results from the execution of Algorithm 6 on some of the traffic states generated in the first reported experiment.

n	p	q	Home	# chains	# iterations	LIVE?	Comp. Time (secs)
5	0.0	0.50	Corner	3	3	YES	0.000342
5	0.3	0.50	Corner	8	1	NO	0.000273
5	0.3	0.75	Middle	2	2	YES	0.000415
25	0.0	0.50	Corner	3	3	YES	0.031004
25	0.3	0.25	Middle	9	8	YES	0.05449
25	0.4	0.50	Middle	158	1	NO	0.007561
50	0.0	0.50	Corner	24	21	YES	1.96478
50	0.2	0.25	Corner	19	18	YES	1.68667
50	0.4	0.25	Middle	221	1	NO	0.034963
75	0.0	0.25	Corner	5	5	YES	2.86839
75	0.1	0.25	Middle	19	18	YES	8.73625
75	0.4	0.50	Middle	1161	1	NO	0.088312
100	0.0	0.25	Middle	5	4	YES	7.62654
100	0.1	0.25	Corner	24	24	YES	37.7072
100	0.3	0.50	Middle	1041	1	NO	4.02736

specifically, for each reported case, Table 1 first reports, in its first four columns, the tuple ($n, p, q,$ home zone location) that characterizes the corresponding configuration of the underlying transport system. The remaining four columns of Table 1 respectively report: (i) the number of chains in the corresponding condensation $C(\hat{G}(s))$; (ii) the number of the primary iterations executed by Algorithm 6; (iii) the outcome of Algorithm 6 regarding the liveness of the corresponding traffic state; and (iv) the computational time of the algorithm.

Some interesting findings in our experiments, that are also reflected in the data reported in Table 1, are as follows: First, it is clear from the provided data that Algorithm 6 runs very fast even on transport system configurations involving many thousands of zones and traveling agents. Characteristically, the largest execution time reported in Table 1 is 37.7 secs, and the corresponding configuration involves a guidepath network of $2 \times 100 \times 99 \times 0.9 = 17,820$ zones and $17,820 \times 0.25 = 4,455$ agents circulating in this network.

Also, the column "# chains" of Table 1 reveals the drastic compression of the input data that is effected through the computation of the condensed PDG $C(\hat{G}(s))$ at the very first iteration of the algorithm.

In addition, this condensed representation of the traffic state allows the algorithm to detect very easily agent formations that imply unavoidable deadlocks, and therefore, the non-liveness of the evaluated state s.

Finally, an interesting finding in this experiment was the difficulty of generating any live states as the values of p and q were progressively increased beyond the first few possible values for them. Furthermore, this difficulty was even more prominent as n itself was increasing to its higher values. A natural explanation of this effect is as follows: PDGs $\hat{G}(s)$ generated by the aforementioned parameterizations will contain a considerable number of paths p connecting to the rest of the PDG $\hat{G}(s)$ through their endpoints only, and possessing a considerable number of directed edges. Unless all these directed edges have the same orientation, rendering p a directed path, a deadlock is unavoidable and the corresponding traffic state s is not live.

Our second experiment tested the empirical performance of Algorithm 6 on guidepath networks that contain many bridge edges, and therefore, Algorithm 6 must resort to the block-based decomposition of the processed DAGs $\mathcal{U}(\hat{G}(s))$ and the hierarchical processing of the identified blocks according to the logic that was defined in Section 3.3. More specifically, each state s considered in this experiment had a DAG $\mathcal{U}(\hat{G}(s))$ that possessed the structure depicted in Figure 9, for some $k \in \{10, 20, 30, \ldots, 90, 100\}$.

In the DAG depicted in Figure 9, each node is labeled by n_x or $n_{x,y}$ while the number annotated within the node is the nodal capacity. Also, the numbers annotated next to each edge e are the corresponding weights $w(e)$. In addition, it is easy to see that edge (n_h, n_c) and all edges (n_i, n_c), $i = 1, \ldots, k-1$, are bridge blocks of the depicted graph, while the $k-1$ subgraphs induced by the node tuples $(n_{i,1}, n_{i_2}, n_{i,3}, n_{i,4})$, for $i = 1, \ldots, k-1$, are bi-blocks. Furthermore, each of these bi-blocks can be merged into a single node by executing the feasible basic mergers $(n_{i,2}, n_{i.4})$, and $(n_{i,3}, n_{i,1})$. Let the single node that will result from each of these mergers be denoted by n_i. Then, assuming that the corresponding

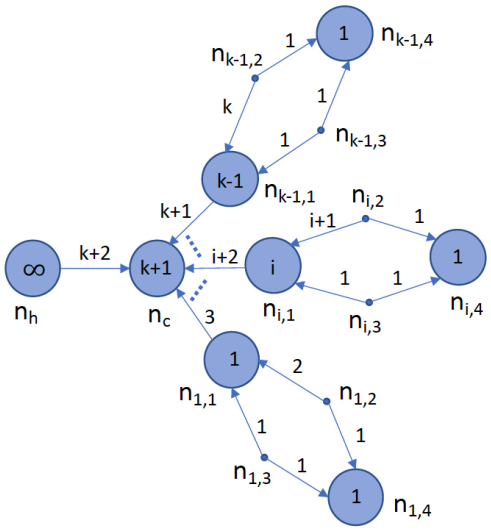


Figure 9: The structure of the DAGs $\mathcal{U}(\hat{G}(s))$ used in the second experiment.

Table 2: The obtained results from the second experiment

k	# nodes	Comp. Time (secs)
10	38	0.047083
20	78	0.274091
30	118	0.888688
40	158	1.97876
50	198	3.58787
60	238	6.12361
70	278	9.9691
80	318	14.4825
90	358	20.4735
100	398	28.4893

bi-block does not contain any hidden capacity in the nodes $n_{i,1}$ and $n_{i,4}$, we have $\chi(n_i) = \chi(n_{i,1}) + \chi(n_{i,4}) = i + 1$. But the bridge edge (n_i, n_c) has a weight of $i + 2$, and therefore, the corresponding basic merger is feasible but not a producer merger. Hence, none of these mergers can increase the capacity of node n_c, and the basic merger (n_h, n_c) remains infeasible, even after the aforementioned mergings of the DAG bi-blocks into single nodes. Therefore, all traffic states s having a DAG $\mathcal{U}(\hat{G}(s))$ belonging in the set of DAGs that are represented by the DAG of Figure 9, are not live.

In order to recognize the nonliveness of these states, Algorithm 6 must process thoroughly and systematically all the blocks of the DAG $\mathcal{U}(\hat{G}(s))$, according to the order that is defined by the corresponding block-cutpoint tree and the algorithm logic. Table 2 reports the computational time of the execution of Algoprithm 6 on ten states s where the corresponding DAGs $\mathcal{U}(\hat{G}(s))$ are defined by the DAG of Figure 9 with the parameter k set to the corresponding values reported in the first column of the table. Also, the column "# nodes" of Table 2 reports the total number of nodes in the corresponding DAG $\mathcal{U}(\hat{G}(s))$. We can see that the required computational times of Algorithm 6 remain very small even for some very large instantiations of the considered DAG.

Finally, the reader should also notice that the DAG used in this experiment represents the considered state s after the compression of the original PDG $\hat{G}(s)$ to the condensed PDG $\mathcal{C}(\hat{G}(s))$ and the DAG $\mathcal{U}(\hat{G}(s))$. Each of the macro-nodes appearing in the DAG $\mathcal{U}(\hat{G}(s))$ can be a maximal chain of the original PDG $\hat{G}(s)$, and these chains can be

arbitrarily large. Nevertheless, except for the overhead of converting the original PDG $\hat{G}(s)$ first to the condensed PDG $\mathcal{C}(\hat{G}(s))$ and subsequently to the DAG $\mathcal{U}(\hat{G}(s))$, the execution time of Algorithm 6 will remain the same for all the input PDGs $\hat{G}(s)$ that are compressed to the same graphs $\mathcal{C}(\hat{G}(s))$ and $\mathcal{U}(\hat{G}(s))$.

4. Conclusions

We showed that the problem of assessing the liveness of traffic states coming from open, irreversible, dynamically routed, zone-controlled, guidepath-based transport systems can be resolved in polynomial time with respect to the size of the underlying guidepath network. Our result is very remarkable since the considered problem is equivalent to a reachability problem defined in the corresponding state space, and most reachability problems involving discrete-event dynamics similar to those that are addressed by this work, possess super-polynomial complexity with respect to the size of the underlying discrete-event system.

The polynomial nature of our result can be attributed to: (i) the graph-theoretic structures that we used for a compact representation of the traffic state s and of the discrete-event dynamics that evolve this state; (ii) the notion of the producer merger that enabled a greedy approach in the search for liveness certificates of the evaluated traffic states; and (iii) the ability to trace these mergers efficiently in the aforementioned graphs. Furthermore, it is worth mentioning that the notion of the producer merger was inspired by the seminal paper of Gold [36], who employed a similar concept in the development of efficient algorithms for assessing state liveness in the context of sequential resource allocation.

The theoretical developments of the work were complemented with some experimental results that highlight the ability of the developed algorithm to assess the liveness of traffic states coming from some very large instantiations of the considered transport systems with extremely small computational times.

Finally, our work also has significant implications for the additional problem of traffic-state liveness assessment that is defined in the context of *closed*, irreversible, dynamically routed, zone-controlled, guidepath-based transport systems. The work of [13] has shown that, for these transport systems, traffic-state liveness is characterized by a result similar to that of Theorem 11; more specifically, as long as the underlying transport system satisfies some structural conditions necessary for exhibiting live behavior, a given traffic state s is live if and only if it is co-reachable to a traffic state s' that is chained (according to Definition 4). But then, for closed, irreversible, dynamically routed, zone-controlled, guidepath-based transport systems with a guidepath network G that satisfies Condition 25, the corresponding decision problem of assessing the liveness of any given traffic state s can be resolved by Algorithm 5, once the part that involves the detection of p-mergers defined on paths leading to the home node n_h has been removed. Indeed, no other parts of Algorithm 5 and its supporting developments make a substantial use of the home node n_h and its infinite capacity, which is the structural element differentiating the considered classes of open and closed transport systems in the context of the employed representations. On the other hand, the presence of the home node n_h plays a more central role in the specification of Algorithm 6, and therefore, the potential adaptation of this algorithm to closed transport systems is a more complicated issue that needs further investigation.

Another issue that can be an interesting subject for further investigations, concerns the potential enhancement of the main iteration of Algorithms 5 and 6 with additional efficient searches for producer mergers, that might expedite even more the identification of such mergers, and result in even faster empirical execution times.

Finally, at a more general level, it is also interesting to explore (i) the possibility of explaining the polynomial-time nature of the presented result based on certain features of the underlying decision problem that might pertain to the broader theory of (DES-theoretic) reachability analysis and algorithms, and (ii) the potential that the presented developments might hold for other similar applications. All the aforementioned issues are part of our future endeavors in this area.

Appendix A. The Master Theorem of Complexity Theory

This appendix provides a statement of the Master Theorem in complexity theory [33].

Theorem 54. *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then, $T(n)$ can be bounded asymptotically as follows.

1. *If $f(n) = O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.*
2. *If $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a} log_2 n)$.*
3. *If $f(n) = \Omega(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

Appendix B. Some important notation, concepts and terminology used in the manuscript

- $\mathcal{A}$: The set of the *agents* circulating in the considered transport system. A generic element of A typically is denoted by a.
- $G = (V, E \cup \{h\}, \xi)$: A multigraph defining the *guidepath network* of the considered transport system. Elements of E correspond to *zones* of the guidepath network that can accommodate no more than one agent at a time. h is a self-loop that represents the *home zone* of the guidepath network; this zone has infinite accommodating capacity. A generic zone other than the home zone is typically denoted by e. Also, a generic vertex of G is denoted by v, while v_h is the particular vertex that is the terminal vertex for h.
- $\Sigma_a = \langle e_i \in E \setminus \{h\} \rangle$: A list of edges specifying the (remaining) *visitation requirements* for agent $a \in \mathcal{A}$ at some arbitrary time point.
- Φ : The *finite state automaton (FSA)* abstracting the traffic dynamics of the considered transport system for the needs of the paper developments.
- S : The *state set* of Φ. A generic element of S is denoted by s. A *state* $s \in S$ is defined by (i) the agent distribution on the edges (or zones) of the guidepath network G, and (ii) the orientation of the agent motion on these edges (but since h is a self-loop, the agent orientation on it is indifferent).
- $\epsilon(a; s)$: The edge occupied by agent a in state s.
- Q : The *event set* of Φ. An *event* $q \in Q$ corresponds to the advancement of a single agent from its current zone $\epsilon(a; s)$ to a zone e' that is incident to the head of $\epsilon(a; s)$ and it is free in state s.
- Q^* : The *Kleene closure* of Q, i.e., the set containing all the finite sequences of elements of Q. A generic element of Q^* is denoted by σ.
- f : The *extended transition function* of Φ defined on $S \times Q^*$.
- s_h : The *home state* of Φ; .i.e., the state where all agents are located at the home zone h.
- S_l : The *set of live states* of Φ; i.e., states for which there exists an event sequence $\sigma \in Q^*$ such that $f(s, \sigma) = s_h$.
- $\hat{G}(s)$: The partially directed graph (PDG) representing state s.
- A *path* π in $\hat{G}(s)$ is a sequence $\pi = \langle v_0, e_1, v_1, e_2, \ldots, e_n, v_n \rangle$, $n \geq 0$, where (i) the subsequence $\langle v_i,\ i = 0, \ldots, n \rangle$ consists of distinct vertices of $\hat{G}(s)$, (ii) for all $i = 1, \ldots, n$, e_i is an edge connecting v_{i-1} and v_i, and (iii) if edge e_i is directed, then its direction is from v_{i-1} to v_i; that is, the sense of direction of edges in π is consistent with the direction of motion implied by the ordering of the path vertices. Also, a *cycle* c of $\hat{G}(s)$ has a structure similar to that of a path, but it contains at least one edge and the starting and the ending vertices, v_0 and v_n, are coinciding. A *joint* between two cycles is a path that belongs to both cycles. A *pass* between two cycles c and c' is a path π such that its first vertex lies on c, its last vertex lies on c', and all edges of π are undirected and do not belong to c, c', or any other directed cycle of $\hat{G}(s)$.
- ch : A *chain* of the PDG $\hat{G}(s)$; c.f. Definition 4 for a complete characterization of this concept.
- $C(\hat{G}(s))$: The *condensation* of the PDG $\hat{G}(s)$; i.e., the PDG induced by $\hat{G}(s)$ by collapsing each maximal chain in it to a single node. Sometimes, a generic instance of such a condensed PDG is denoted by C.

- u : A *u-component* of the PDG $C(\hat{G}(s))$; i.e., a maximal connected subgraph C_u of $C(\hat{G}(s))$ that contains no directed edges.

- $\mathcal{U}(\hat{G}(s))$: The directed acyclic graph (DAG) obtained from the condensation $C(\hat{G}(s))$ by collapsing each single u-component of $C(\hat{G}(s))$ to a single vertex. The vertex set of $\mathcal{U}(\hat{G}(s))$ is denoted by $\mathcal{N}$, its elements are referred to as "nodes", and they are denoted by n. The edge set of $\mathcal{U}(\hat{G}(s))$ is denoted by $\mathcal{E}$. Furthermore, the finally employed version of $\mathcal{U}(\hat{G}(s))$ is a labeled DAG that is obtained by (i) reducing each maximal path π of the original DAG $\mathcal{U}(\hat{G}(s))$ leading from some n' to some node n and containing no branching nodes in it, into single directed edge $e = (n', n)$, and (ii) assigning the length of path π as a label $w(e)$ to edge e. Sometimes, a generic instance of $\mathcal{U}(\hat{G}(s))$ is denoted by $\mathcal{G}$.

- n_h : The node of the DAG $\mathcal{U}(\hat{G}(s))$ corresponding to the subgraph of the guidepath network G that contains the home zone h.

- $\zeta(ch)$: The *capacity of chain ch* of $C(\hat{G}(s))$; i.e., the number of free edges that belong on cycles of the subgraph of the PDG $\hat{G}(s)$ corresponding to chain ch.

- $\chi(n)$: The *capacity of node n* of the DAG $\mathcal{U}(\hat{G}(s))$; this notion is induced from the notion of the chain capacity $\zeta(ch)$ according to Defintiion 8.

- An edge $e = (n', n)$ of the DAG $\mathcal{U}(\hat{G}(s))$ defines a *basic merger* according Definition 12. This merger is feasible if $\chi(n) \geq w(e)$, and its execution collapses edge e to a single node n'' with capacity $\chi(n'') = \chi(n') + \chi(n) - w(e)$.

- A (feasible) *macro-merger* $\mathcal{M} = \langle e_1, e_2, \ldots, e_n \rangle$ defined in the DAG $\mathcal{U}(\hat{G}(s))$ is a sequence of edges that (i) induce a weakly connected sub-DAG of $\mathcal{U}(\hat{G}(s))$, (ii) edge e_1 defines a feasible basic merger in the DAG $\mathcal{U}(\hat{G}(s))$, and (iii) every other edge e_i, $i = 2, \ldots, n$, is a basic feasible merger in the DAG $\mathcal{G}(e_1, \ldots, e_{i-1})$ that is obtained from the execution of the basic mergers $e_1, \ldots, e_{i-1}$. The node resulting from the execution of macro-merger $\mathcal{M}$ is denoted by $n(\mathcal{M})$. $\mathcal{M}$ is a *producer merger* if $\chi(n(M)) \geq \chi(n)$ for every node n of the DAG $\mathcal{U}(\hat{G}(s))$ that is merged into node $n(\mathcal{M})$.

- $\mathcal{M} + \mathcal{M}'$: The macro-merger obtained from the *concatenation* of the edge sequences that define the macro-mergers $\mathcal{M}$ and $\mathcal{M}'$.

- *t-mergers, p-mergers* and *c-mergers* are macro-mergers where the merged subgraphs of the DAG $\mathcal{U}(\hat{G}(s))$ are, respectively, in-trees, single paths or possessing the circular structure that is depicted in the right part of Figure 2. Also, c.f. Definitions 15, 20 and 23.

- $\mathcal{T}(n, e)$: A sub-DAG of the DAG $\mathcal{U}(\hat{G}(s))$ that is induced by a terminal node n of $\mathcal{U}(\hat{G}(s))$ and an edge e that is incident to node n. $\mathcal{T}(n, e)$ consists of all the maximal paths of $\mathcal{U}(\hat{G}(s))$ that lead to node n via edge e, and have interior nodes with an out-degree of one.

- $\mathcal{T}(n)$: A sub-DAG of the DAG $\mathcal{U}(\hat{G}(s))$ that is induced by a terminal node n of $\mathcal{U}(\hat{G}(s))$ by taking the union of the DAGs $\mathcal{T}(n, e)$ for all the edges e incident to node n.

- *In-tree* $\tilde{\mathcal{T}}$: A rooted tree $\mathcal{T}$ with directed edges that point towards the root.

- $\tilde{\mathcal{T}}(n)$: The in-tree obtained from the DAG $\mathcal{T}(n)$ by removing all its source nodes.

- $\widehat{\mathcal{U}(\hat{G}(s))}$: The undirected graph that is induced by the DAG $\mathcal{U}(\hat{G}(s))$.

- A *component* of the graph $\widehat{\mathcal{U}(\hat{G}(s))}$ is a maximal connected subgraph.

- $\widehat{\mathcal{U}(\hat{G}(s))} - n$: The graph obtained from $\widehat{\mathcal{U}(\hat{G}(s))}$ by removing node n and all the edges incident to n. Node n is an *articulation node* of $\widehat{\mathcal{U}(\hat{G}(s))}$ if the graph $\widehat{\mathcal{U}(\hat{G}(s))} - n$ has more components than the graph $\widehat{\mathcal{U}(\hat{G}(s))}$. We extend the characterization of an articulation node to the nodes of the DAG $\mathcal{U}(\hat{G}(s))$.

- $\widehat{\mathcal{U}(\hat{G}(s))} - e$: The graph obtained from $\widehat{\mathcal{U}(\hat{G}(s))}$ by removing edge e from it. Edge e is a *bridge* (edge) of $\widehat{\mathcal{U}(\hat{G}(s))}$ if $\widehat{\mathcal{U}(\hat{G}(s))} - e$ has more components than the graph $\widehat{\mathcal{U}(\hat{G}(s))}$. We extend the characterization of a bridge edge to the edges of the DAG $\mathcal{U}(\hat{G}(s))$.

- A *block* B of the DAG $\mathcal{U}(\hat{G}(s))$: A maximal connected subgraph of the undirected graph $\widehat{\mathcal{U}(\hat{G}(s))}$ that has no articulation vertex. We extend this concept to the DAG $\mathcal{U}(\hat{G}(s))$. The blocks B of $\mathcal{U}(\hat{G}(s))$ are either bridge edges, characterized as *bridge blocks*, or biconnected subgraphs, characterized as *bi-blocks*.

- The *block-cutpoint tree* of the DAG $\mathcal{U}(\hat{G}(s))$: A tree representing the decomposition of the the DAG $\mathcal{U}(\hat{G}(s))$ to its blocks. The nodes of this tree are the blocks of the DAG $\mathcal{U}(\hat{G}(s))$ and its articulation nodes, and the edges link each block with its articulation nodes. Hence, the block-cutpoint tree is a bipartite graph. This concept is demonstrated in Figure 7 and the accompanying discussion. Also, the block-cutpoint tree of the DAG $\mathcal{U}(\hat{G}(s))$ is treated as a rooted tree, with the root being either the home node n_h itself, if this node is an articulation node, or the block that contains node n_h, otherwise.

- $\mathcal{G}_B$: The sub-DAG corresponding to block B of DAG $\mathcal{G}$.

- A block B of DAG $\mathcal{G}$ is further classified as *type-I*, if its parent articulation node n is a terminal node of $\mathcal{G}_B$. Otherwise, it is *type-II*.

References

[1] G. Daugherty, S. Reveliotis, G. Mohler, Optimized multi-agent routing for a class of guidepath-based transport systems, IEEE Trans. on Automation Science and Engineering 16 (2019) 363–381.

[2] W. L. Maxwell, J. A. Muckstadt, Design of automatic guided vehicle systems, IIE Trans. 14 (1982) 114–124.

[3] T. Ganesharajah, N. G. Hall, C. Sriskandarajah, Design and operational issues in AGV-served manufacturing systems, Annals of OR 76 (1998) 109–154.

[4] I. F. A. Vis, Survey of research in the design and control of automated guided vehicle systems, European Journal of Operational Research 170 (2006) 677–709.

[5] S. S. Heragu, Facilities Design (3rd ed.), CRC Press, 2008.

[6] M. Weiss, Semiconductor factory automation, Solid State Technology (1996) 89–96.

[7] J. Nestel-Patt, Semiconductor manufacturing: the final automation wave, Robotics World Spring (1998) 32–35.

[8] D. Pillai, The future of semiconductor manufacturing: Factory integration breakthrough opportunities, IEEE Robotics & Automation Magazine 13-4 (2006) 16–24.

[9] S. A. Reveliotis, Conflict resolution in AGV systems, IIE Trans. 32(7) (2000) 647–659.

[10] N. Wu, M. Zhou, Resource-oriented Petri nets in deadlock avoidance of AGV systems, in: Proceedings of the ICRA'01, IEEE, 2001, pp. 64–69.

[11] M. P. Fanti, Event-based controller to avoid deadlock and collisions in zone-controlled AGVS, Intl. Jrnl Prod. Res. 40 (2002) 1453–1478.

[12] D. Y. Liao, M. Jeng, M. Zhou, Petri net modeling and Lagrangian relaxation approach to vehicle scheduling in 300mm semiconductor manufacturing, in: IEEE Intl. Conf. on Robotics & Automation, 2004.

[13] E. Roszkowska, S. Reveliotis, On the liveness of guidepath-based, zoned-controlled, dynamically routed, closed traffic systems, IEEE Trans. on Automatic Control 53 (2008) 1689–1695.

[14] T. Standley, Finding optimal solutions to cooperative pathfinding problems, in: Proc. AAAI 2010, 2010.

[15] Q. Sajid, R. Luna, K. E. Bekris, Multi-agent path finding with simultaneous execution of single-agent primitives, in: 5th Symposium on Combinatorial Search, 2012.

[16] J. Yu, S. M. LaValle, Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics, IEEE Trans. on Robotics 32 (2016) 1163–1177.

[17] H. Ma, C. Tovey, G. Sharon, S. Kumar, S. Koenig, Multi-agent path finding with payload transfers and the package-exchange robot-routing problem, in: AAAI 2016, 2016, pp. 3166–3173.

[18] R. M. Wilson, Graph puzzles, homotopy, and the alternating group, Journal of Combinatorial Theory, B 16 (1974) 86–96.

[19] D. Kornhauser, G. Miller, P. Spirakis, Coordinating pebble motion on graphs, the diameter of permutation groups, and applications, in: Proc. IEEE Symp. Found. Comput. Sci., IEEE, 1984, pp. 241–250.

[20] V. Auletta, A. Monti, M. Parente, P. Persiano, A linear-time algorithm for the feasibility of pebble motion on trees, Algorithmica 23 (1999) 223–245.

[21] P. Surynek, Towards optimal cooperative path planning in hard setups through satisfiability solving, in: Proc. 12th Pacific Rim Int. Conf. Artif. Intell., 2012, pp. 564–576.

[22] G. Daugherty, Multi-agent routing in shared guidepath networks, Ph.D. thesis, Georgia Tech, Atlanta, GA (2017).

[23] S. Reveliotis, On the state liveness of some classes of guidepath-based transport systems and its computational complexity, Automatica 113.

[24] S. Reveliotis, An MPC scheme for traffic coordination in open and irreversible, zone-controlled, guidepath-based transport systems, IEEE Trans. on Automation Science and Engineering 17 (2020) 1528–1542.

[25] S. Reveliotis, T. Masopust, Some new results on the state liveness of open guidepath-based traffic systems, in: 27th Mediterranean Conference on Control and Automation (MED 2019), IEEE, 2019.
[26] B. H. Krogh, L. E. Holloway, Synthesis of feedback control logic for discrete manufacturing systems, Automatica 27 (1991) 641–651.
[27] B. A. Brandin, The real-time supervisory control of an experimental manufacturing cell, IEEE Trans. on Robotics and Automation 12 (1996) 1–14.
[28] W. M. Wonham, Supervisory control of discrete event systems, Tech. Rep. ECE 1636F / 1637S 2006-07, Electrical & Computer Eng., University of Toronto (2006).
[29] S. Reveliotis, E. Roszkowska, On the complexity of maximally permissive deadlock avoidance in multi-vehicle traffic systems, IEEE Trans. on Automatic Control 55 (2010) 1646–1651.
[30] J. Girault, J.-J. Loiseau, O. H. Roux, On-line compositional controller synthesis for AGV, Discrete Event Dynamic Systems: Theory and Applications 26 (2016) 583–610.
[31] S. Reveliotis, T. Masopust, Efficient liveness assessment for traffic states in open, irreversible, dynamically routed, zone-controlled guidepath-based transport systems, IEEE Trans. on Automatic Control 65 (2020) 2883–2898.
[32] S. Reveliotis, T. Masopust, Efficient assessment of state liveness in open, irreversible, dynamically routed, zone-controlled guidepath-based transport systems: The general case, in: 15th Workshop on Discrete Event Systems (WODES 2020), IFAC, 2020.
[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 2nd ed., MIT Press, Boston, MA, 2001.
[34] D. B. West, Introduction to Graph Theory (2nd ed.), Prentice Hall, Upper Saddle River, N.J., 2001.
[35] R. E. Tarjan, A note on finding the bridges of a graph, Information Processing Letters (1974) 161–162.
[36] E. M. Gold, Deadlock prediction: Easy and difficult cases, SIAM Journal of Computing 7 (1978) 320–336.