Computing and Listing st-Paths in Public Transportation Networks

Theory Comput Syst (2018) 62:600–621
DOI 10.1007/s00224-016-9747-4

Computing and Listing *st*-Paths in Public Transportation Networks

Kateřina Böhmová[1] · Luca Häfliger[1] · Matúš Mihalák[2] · Tobias Pröger[1] · Gustavo Sacomoto[3,4] · Marie-France Sagot[3,4]

Published online: 11 January 2017

Abstract Given a set of directed paths (called *lines*) L, a *public transportation network* is a directed graph $G_L = (V_L, A_L)$ which contains exactly the vertices and arcs of every line $l \in L$. An st-route is a pair (π, γ) where $\gamma = \langle l_1, \ldots, l_h \rangle$ is a line sequence and π is an st-path in G_L which is the concatenation of subpaths of the lines $l_1, \ldots, l_h$, in this order. Given a threshold β, we present an algorithm for listing all *st-paths* π for which a route (π, γ) with $|\gamma| \leq \beta$ exists, and we show that the running time of this algorithm is polynomial with respect to the input and the output size. We also present an algorithm for listing all *line sequences* γ with $|\gamma| \leq \beta$ for which a route (π, γ) exists, and show how to speed it up using preprocessing. Moreover, we show that for the problem of finding an st-route (π, γ) that minimizes the number of different lines in γ, even computing an $o(\log |V|)$-approximation is NP-hard.

This work has been partially supported by the Swiss National Science Foundation (SNF) under the grant number 200021 138117/1, and by the EU FP7/2007-2013 (DG CONNECT.H5-Smart Cities and Sustainability), under grant agreement no. 288094 (project eCOMPASS). Kateř ina Böhmová is a recipient of a Google Europe Fellowship in Optimization Algorithms, and this research is supported in part by this Google Fellowship. Gustavo Sacomoto is a recipient of a grant from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement n° [247073]10 SISYPHE. Preliminary versions of parts of this paper appeared in [5, 7].

✉ Tobias Pröger
tobias.proeger@inf.ethz.ch

[1] Department of Computer Science, ETH Zürich, Zürich, Switzerland

[2] Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands

[3] INRIA Grenoble Rhône-Alpes, Montbonnot-Saint-Martin, France

[4] UMR CNRS 5558 – LBBE, Université Lyon 1, Lyon, France

Keywords Listing algorithm · Public transportation · NP-hardness

1 Introduction

Motivation Given a public transportation network (in the following called *transit network*) and two locations s and t, a common goal is to find a fastest journey from s to t, i.e. an st-journey whose travel time is minimum among all st-journeys. A fundamental feature of any public transportation information system is to provide, given s, t and a target arrival time t_A, a fastest st-journey that reaches t no later than at time t_A. This task can be solved by computing a shortest path in an auxiliary graph that also reflects time [20]. However, if delays occur in the network (which often happens in reality [14]), then the goal of computing a *robust* st-journey that likely reaches t on time, naturally arises.

The problem of finding robust journeys received much attention in the literature (for a survey, see, e.g., [1]). Recently, Böhmova et al. [4, 6] proposed the following two-stage approach for computing robust journeys. In the first step, all structural alternative length-bounded routes (ignoring time) are listed explicitly, and only after that, timetables and historic traffic data are incorporated to evaluate the robustness of each possible route. From a practical point of view it is reasonable to restrict the maximum number of transfers for two reasons: 1) the number of listed routes might be huge, hence listing all possible routes leads to a non-satisfactory running time, and 2) many routes might be unacceptable for the user, e.g., because they use many more transfers than necessary. Having a huge number of transfers is not only uncomfortable, but usually also has a negative impact on the robustness of routes, since each transfer bears a risk of missing the next connection when vehicles are delayed.

Our contribution The main contribution of the present paper are three algorithms that list all st-routes for which the number of transfers does not exceed a given threshold β. The running times of our algorithms are polynomial with respect to the sum of the input and the output size. As a subroutine of this algorithm we need to compute a route with a minimum number of transfers which is known to be solvable efficiently [20]. However, we show that finding a route with a minimum number of *different* lines cannot be approximated within $(1-\varepsilon)\ln n$ for any $\varepsilon > 0$ unless NP = P.

We note that for bus networks it is reasonable to consider directed networks (instead of undirected ones), because real-world transit networks (such as the one in the city of Barcelona) may contain one-way streets in which busses can only operate in a single direction.

Related work Listing combinatorial objects (such as paths, cycles, spanning trees, etc.) in graphs is a widely studied field in computer science (see, e.g., [2]). The currently fastest algorithm for listing all st-paths in directed graphs was presented by Johnson [16] in 1975 and runs in time $\mathcal{O}((n+m)(\kappa+1))$ where n and m are the number of vertices and arcs, respectively, and κ is the number of all st-paths (i.e., the size of the output). For undirected graphs, an optimal algorithm was presented by

Birmelé et al. [3]. A related problem is the K-shortest path problem, which asks, for a given constant K, to compute the first K distinct shortest st-paths. Yen [27] and Lawler [19] studied this problem for directed graphs. Their algorithm uses Dijkstra's algorithm [10] and can be implemented to run in time $\mathcal{O}(K(nm + n^2 \log n))$ using Fibonacci heaps [15]. For undirected graphs, Katoh et al. [18] proposed an algorithm with running time $\mathcal{O}(K(m + n \log n))$. Eppstein [13] gave an $\mathcal{O}(K + m + n \log n)$ algorithm for listing the first K distinct shortest *st-walks*, i.e., paths in which vertices are allowed to appear more than once. Recently, Rizzi et al. [22] studied a different parameterization of the K shortest path problem where they ask to list all st-paths with length at most α for a given α. The difference to the classical K shortest path problem is that the lengths (instead of the overall number) of the paths output is bounded. Thus, depending on the value of α, K might be exponential in the input size. The running time of the proposed algorithm coincides with the running time of the algorithm of Yen and Lawler for directed graphs, and with the running time of the algorithm of Katoh et al. for undirected graphs. However, the algorithm of Rizzi et al. uses only $\mathcal{O}(n+m)$ space which is linear in the input size. All these algorithms cannot directly be used for our listing problem, since we have the additional constraint to list only paths for which a route of length at most β exists, and since lines can share multiple transfers. A more detailed explanation is given in Section 6.

Xu et al. [26] studied the problem of listing all paths in a transit network that have at most two transfers. They encode the various possible situations that may occur directly into their algorithm. This algorithm is then used as a subroutine to list the k earliest-arriving paths. The same problem was addressed very recently by Vo et al. [25]. They propose a time-dependent version of Yen's algorithm [27]. To avoid the generation of similar paths, they propose a similarity measure between two paths P_1 and P_2 which essentially relates the number of common lines in both paths to the minimum number of lines that give rise to P_1 and to P_2, respectively. Nguyen et al. [21] studied an *implicit* listing problem in hypergraphs. Every hyperarc has exactly one tail t and multiple heads H, and the authors assume that for any $H' \subseteq H$, the hyperarc from t to H' is also implicitly contained. Essentially, the tail represents a stop s, and every vertex in H represents one line that visits s. Vehicles of the lines are assumed to arrive according to a given distribution. Now, for a given departure stop d and a given target stop t, one aims to identify so-called *efficient* hyperarcs (i, H') such that for every $j \in H'$ the expected travel time from j to t is smaller than the expected travel time from i to t. Hence, they do not explicitly list all possible paths, but encode the reasonable paths implicitly using efficient hyperarcs.

2 Preliminaries

Mathematical preliminaries Let $G = (V, A)$ be a directed graph. A *walk* in G is a sequence of vertices $\langle v_0, \ldots, v_k \rangle$ such that $(v_{i-1}, v_i) \in A$ for all $i \in [1, k]$. For a walk $w = \langle v_0, \ldots, v_k \rangle$ and a vertex $v \in V$, we write $v \in w$ if and only if there exists an index $i \in [0, k]$ such that $v = v_i$. Analogously, for a walk $w = \langle v_0, \ldots, v_k \rangle$ and an arc $a = (u, v) \in A$, we write $a \in w$ if and only if there exists an index $i \in [1, k]$ such that $u = v_{i-1}$ and $v = v_i$. The length of a walk $w = \langle v_0, \ldots, v_k \rangle$

is k, the number of arcs in the walk, and is denoted by $|w|$. A walk w of length $|w| = 0$ is called *degenerate*, and *non-degenerate* otherwise. For two walks $w_1 = \langle u_0, \ldots, u_k \rangle$ and $w_2 = \langle v_0, \ldots, v_l \rangle$ with $u_k = v_0$, $w_1 \cdot w_2$ denotes the *concatenation* $\langle u_0, \ldots, u_k = v_0, \ldots, v_l \rangle$ of w_1 and w_2. A *path* is a walk $\pi = \langle v_0, \ldots, v_k \rangle$ such that $v_i \neq v_j$ for all $i \neq j$ in $[0, k]$, i.e. a path is a walk without crossings. Given a path $\pi = \langle v_0, \ldots, v_k \rangle$, every contiguous subsequence $\pi' = \langle v_i, \ldots, v_j \rangle$ is called a *subpath* of π. A path $\pi = \langle s = v_0, v_1, \ldots, v_{k-1}, v_k = t \rangle$ is called an st-path. For a vertex $v \in V$, let $N_G^+(v)$ denote the out-neighborhood of v. Given two integers i, j, we define the function δ_{ij} (*Kronecker delta*) as 1 if $i = j$ and 0 if $i \neq j$.

Lines and transit networks Given a set of non-degenerate paths (called *lines*) L, the *transit network induced by* L is the graph $G_L = (V_L, A_L)$ where V_L contains exactly the vertices v for which L contains a line l with $v \in l$, and A_L contains exactly the arcs a for which L contains a line l with $a \in l$. This definition is similar to the definition of the *station graph* in [23], and it does not include travel times or timetables since we are only interested in the structure of the network. The modeling differs from classical graph-based models like the *time-expanded* or the *time-dependent* model which incorporate travel times explicitly by adding additional vertices or cost functions in the arcs, respectively (see, e.g., [8, 20] for more information on these models). However, for finding robust routes with the approach in [6], the above definition is sufficient since travel times are integrated at a later stage. In the following, let $M_L = \sum_{l \in L} |l|$ denote the sum of the lengths of all lines. In the rest of this paper, we omit the index L from V_L, A_L and M_L to simplify the notation.

Given a path $\pi = \langle v_0, \ldots, v_k \rangle$ in G_L and a sequence of lines $\gamma = \langle l_1, \ldots, l_h \rangle$, we say that the pair (π, γ) is a *route* if π is equal to the concatenation of non-degenerate subpaths $\pi_1, \ldots, \pi_h$ of the lines $l_1, \ldots, l_h$, in this order. Notice that a line might occur multiple times in γ (see Fig. 1); however, we assume that any two consecutive lines in γ are different. For every $i \in \{1, \ldots, h-1\}$, we say that a *line change* between the lines l_i and l_{i+1} occurs. The *length* of the route (π, γ) is $|\gamma|$, i.e. the number of line changes plus one. Given two vertices $u, v \in V$, a uv-route is a route (π, γ) such that π is a uv-path. A *minimum* uv-route has smallest length among all uv-routes in G_L, and we define the *L-distance* $d_L(u, v)$ from u to v as the length of a minimum uv-route. For a path π and a line $l \in L$, let $l - \pi$ be the union of (possibly degenerate) paths that we obtain after removing every vertex $v \in \pi$ and its adjacent

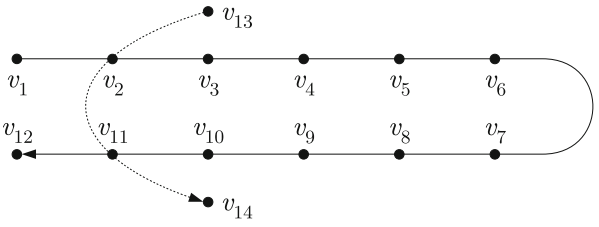

Fig. 1 A transit network with one-way streets induced by a line $l_1 = \langle v_1, \ldots, v_{12} \rangle$ (*solid*) and a line $l_2 = \langle v_{13}, v_2, v_{11}, v_{14} \rangle$ (*dotted*). To travel from $s = v_1$ to $t = v_{12}$, it is reasonable to use l_1 until v_2, after that use l_2 from v_2 to v_{11} and from there finally use l_1 again

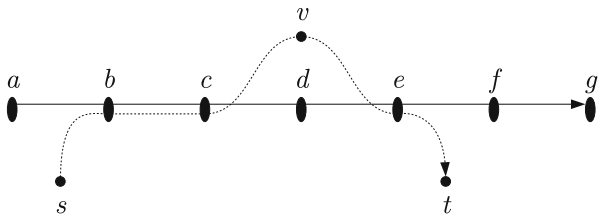

Fig. 2 Let $l = \langle a, b, c, d, e, f, g\rangle$ be a line (*solid*) and $\pi = \langle s, b, c, v, e, t\rangle$ be a path (*dotted*). Then, $l - \pi = \langle a, d, f, g\rangle$ is the disjoint union of the degenerate lines $\langle a\rangle$ and $\langle d\rangle$, and the non-generated line $\langle f, g\rangle$

arcs from l (see Fig. 2). For simplicity, we also call each of these unions of paths a *line*, although they might be disconnected and/or degenerated. However, we note that all algorithms in this paper also work for disconnected and/or degenerate lines. Given a path π and a set L of lines, let $L - \pi = \{l - \pi \mid l \in L\}$ denote the set of all lines in which every vertex from π has been removed. Analogously to our previous definitions, given a path π and a graph G, we define $G - \pi$ as the graph from which every vertex $v \in \pi$ and its adjacent arcs have been removed.

Problems An algorithm that systematically lists all or a specified subset of solutions of a combinatorial optimization problem is called a *listing algorithm*. The *delay* of a listing algorithm is the maximum of the time elapsed until the first solution is output and the times elapsed between any two consecutive solutions are output [17, 22].

Problem 1 (Finding a minimum st-route) Given a transit network $G_L = (V, A)$ and two vertices $s, t \in V$, find a minimum route from s to t.

Problem 2 (Finding an st-route with a minimum number of different lines) Given a transit network $G_L = (V, A)$ and two vertices $s, t \in V$, find a route from s to t that uses a minimum number of different lines from L.

Notice that, although Problems 1 and 2 sound similar, they are in general not equivalent. Figure 3 shows an example for a transit network in which the optimal solutions of the problems differ.

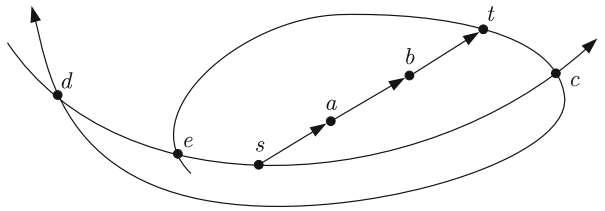

Fig. 3 A transit network induced by the lines $l_1 = \langle s, a\rangle$, $l_2 = \langle a, b\rangle$, $l_3 = \langle b, t\rangle$, $l_4 = \langle d, e, s, c\rangle$ and $l_5 = \langle e, t, c, d\rangle$. The route $r_1 = (\langle s, a, b, t\rangle, \langle l_1, l_2, l_3\rangle)$ is an optimal solution for Problem 1. It uses three different lines and two transfers. However the optimal solution for Problem 2 is the route $r_2 = (\langle s, c, d, e, t\rangle, \langle l_4, l_5, l_4, l_5\rangle)$ which uses only two different lines but three transfers

A natural listing problem is to list all possible st-routes. However, this formulation has the disadvantage that the number of possible solutions is huge, and that there might exist many redundant solutions since a path π can give rise to multiple distinct routes (e.g., if some arc of π is shared by two lines) and vice versa. Moreover, from a practical point of view, also routes that contain many line changes are undesirable. Thus, we formulate the following two listing problems.

Problem 3 (Listing β-bounded line sequences) Given a transit network $G_L = (V, A)$, two vertices $s, t \in V$, and $\beta \in \mathbb{N}$, output all line sequences γ such that there exists at least one route (π, γ) with length at most β.

Problem 4 (Listing β-bounded st-paths) Given a transit network $G_L = (V, A)$, two vertices $s, t \in V$, and $\beta \in \mathbb{N}$, output all st-paths π such that there exists at least one route (π, γ) with length at most β.

3 Finding an Optimal Solution

In this section we discuss solutions to the Problems 1 and 2. As a preliminary observation we show that for *undirected* lines (i.e., undirected connected graphs where every vertex has degree 2 or smaller) and *undirected* transit networks, the problems are equivalent and can be solved in time $\Theta(M)$. Essentially they are easy because lines can always be traveled in both directions. Of course, this does not hold in the case of directed graphs (see Fig. 3). While Problem 1 can be solved in time $\Theta(M)$ using Dial's (implementation of Dijkstra's) algorithm [9] on an auxiliary graph similar to the one presented in [20], Problem 2 turns out to be NP-hard to approximate.

Theorem 1 *If all lines in L are undirected and G_L is the undirected induced transit network, then Problems 1 and 2 coincide and can be solved in time $\Theta(M)$ where $M = \sum_{l \in L} |l|$ is the input size.*

Proof Let $r = (\pi, \gamma)$ with $\pi = (\pi_1, \ldots, \pi_h)$ and $\gamma = (l_1, \ldots, l_h)$ be an optimal solution to Problem 2. We first show that there always exists an optimal solution $\bar{r} = (\bar{\pi}, \bar{\gamma})$ that uses every line in $\bar{\gamma}$ exactly once. Suppose that some line l occurred multiple times in γ. Let i be the smallest index such that $l_i = l$, and let j be the largest index such that $l_j = l$. Let v be the first vertex on π_i (i.e., the first vertex on the subpath served by the first occurrence of l), and let w be the last vertex on π_j (i.e., the last vertex on the subpath served by the last occurrence of l). Let π_{sv} be the subpath of π starting in s and ending in v, π_{vw} be a subpath of l from v to w, and π_{wt} be the subpath of π starting in w and ending in t. The route $r' = (\pi', \gamma')$ with $\pi' = \pi_{sv} \cdot \pi_{vw} \cdot \pi_{wt}$ and $\gamma' = (l_1, \ldots, l_{i-1}, l, l_{j+1}, \ldots, l_h)$ is still an st-route, it uses the line l exactly once, and overall it does not use more different lines than r does. Thus, repeating the above argument for every line l that occurs multiple times, we obtain a route $\bar{r} = (\bar{\pi}, \bar{\gamma})$ which uses every line in $\bar{\gamma}$ exactly once and which is still an optimal solution to Problem 2.

The above argument can also be applied to show that every optimal solution (π, γ) to Problem 1 uses every line in γ exactly once. Now it easy to see that Problem 1 has a solution with exactly k line changes if and only if Problem 2 has a solution with exactly $k+1$ different lines. Therefore, Problems 1 and 2 are equivalent. They can efficiently be solved as follows. For a given transit network $G_L = (V, A)$, consider the vertex-line incidence graph $G' = (V \cup L, A')$ where

$$A' = \{\{v, l\} \mid v \in V \wedge l \in L \wedge \text{line } l \text{ contains vertex } v\}. \tag{1}$$

Breadth-first search can be used to find a shortest st-path $\langle s, l_1, v_1, \ldots, v_{k-1}, l_k, t\rangle$ in G'. Let $\gamma = (l_1, \ldots, l_k)$ be the sequence of lines in this path. Now we use a simple greedy strategy to find a path π in the transit network G_L such that π is the concatenation of subpaths of $l_1, \ldots, l_k$: we start in s, follow l_1 in an arbitrary direction until we find the vertex v_1; if v_1 is not found, we traverse l_1 in the opposite direction until we find v_1. From v_1 we search v_2 on line l_2, and continue correspondingly until we reach t on line l_k. Now the pair (π, γ) is a route with a minimum number of transfers (and, with a minimum number of different lines).

We have $|V \cup L| \in \mathcal{O}(M)$ and $|A'| \in \Theta(M)$, thus the breadth-first search runs in time $\Theta(M)$. Furthermore, G' can be constructed from G_L in time $\Theta(M)$. Thus, for undirected lines and undirected transit networks, Problems 1 and 2 can be solved in time $\Theta(M)$. □

To solve Problem 1 for a *directed* transit network $G_L = (V, A)$, one can construct a weighted auxiliary graph $\Gamma[G_L] = (V[\Gamma], A[\Gamma])$ such that $V \subseteq V[\Gamma]$, and for any two vertices $s, t \in V$ the cost of a shortest st-path in $\Gamma[G_L]$ is exactly $d_L(s, t)$. For a given vertex $v \in V$, let $L_v \subseteq L$ be the set of all lines that contain v. We add every vertex $v \in V$ to $V[\Gamma]$. Additionally, for every vertex $v \in V$ and every line $l \in L_v$, we create a new vertex v_l and add it to $V[\Gamma]$. The set $A[\Gamma]$ contains three different types of arcs:

1) For every arc $a = (u, v)$ in a line l, we create a *traveling* arc (u_l, v_l) with cost 0. These arcs are used for traveling along a line l.
2) For every vertex v and every line $l \in L_v$, we create a *boarding* arc (v, v_l) with cost 1. These arcs are used to board the line l at vertex v.
3) For every vertex v and every line $l \in L_v$, we create a *leaving* arc (v_l, v) with cost 0. These arcs are used to leave the line l at vertex v.

This construction is similar to the *train-route graph* for the *Minimum Number of Transfers Problem* described by Müller-Hannemann et al. [20] except that we penalize boarding arcs instead of leaving arcs. We nevertheless describe and analyze it explicitly because it will be used as a subroutine in the listing algorithms in the Sections 4 and 6, and the details of the construction are important for the running time analysis of our listing algorithms.

Theorem 2 *Problem 1 is solvable in time $\Theta(M)$ where $M = \sum_{l \in L} |l|$ is the input size.*

Proof Let $G_L = (V, A)$ be a transit network and $s, t \in V$ be arbitrary. We compute the graph $\Gamma[G_L]$ and run Dial's algorithm [9] on the vertex s. Let π_{st} be a

shortest st-path in $\Gamma[G_L]$. It is easy to see that the cost of π_{st} is exactly $d_L(s,t)$ [20]. Furthermore, π_{st} induces an st-path in G_L by replacing every traveling arc (v_l, w_l) by (v, w), and ignoring the arcs of the other two types [23]. Analogously the line sequence can be extracted from π_{st} by considering the lines l of all boarding arcs (v, v_l) in π_{st} (or, alternatively, by considering the lines l of all leaving arcs (v_l, v) in π_{st}).

For every vertex v served by a line l, $\Gamma[G_L]$ contains at most two vertices (namely, v_l and v), thus we have $|V[\Gamma]| \in \mathcal{O}(M)$. Furthermore, $A[\Gamma]$ contains every arc a of every line, and exactly two additional arcs for every vertex v_l. Thus we obtain $|A[\Gamma]| \in \mathcal{O}(M)$. Since the largest arc weight is $C = 1$ and Dial's algorithm runs in time $\mathcal{O}(|V[\Gamma]|C + |A[\Gamma]|)$, Problem 1 can be solved in time $\mathcal{O}(M)$. □

In contrast to the previous Theorem, we will show now that finding a route with a minimum number of *different* lines is NP-hard to approximate.

Theorem 3 *Problem 2 is NP-hard to approximate within* $(1-\varepsilon)\ln n$ *for any* $\varepsilon > 0$ *unless NP* $=$ *P.*

Proof We construct an approximation preserving reduction from SETCOVER. The reduction is similar to the one presented in [28] for the minimum-color path problem. Given an instance $I = (X, \mathcal{S})$ of SETCOVER, where $X = \{x_1, \ldots, x_n\}$ is the ground set, and $\mathcal{S} = \{S_1, \ldots, S_m\}$ is a family of subsets of X, the goal is to find a minimum cardinality subset $\mathcal{S}' \subseteq \mathcal{S}$ such that the union of the sets in $\mathcal{S}'$ contains all elements from X.

We construct from I a set of lines L that induces a transit network $G_L = (V, A)$ as follows. See Fig. 4 along with the construction. The set L consists of $m+1$ lines and induces $2n$ vertices. The vertex set $V = \{v_1^a, v_1^b, v_2^a, v_2^b, \ldots, v_n^a, v_n^b\}$ contains two vertices v_i^a and v_i^b for each element x_i of the ground set X. Let $V^O = \langle v_1^a, v_1^b, \ldots, v_n^a, v_n^b\rangle$ be the order naturally defined by V. The set of lines $L = \{l_1, \ldots, l_m, l_{aux}\}$ contains one line for each set in $\mathcal{S}$, and one additional auxiliary line l_{aux}. For a set $S_i \in \mathcal{S}$, consider the set of vertices that correspond to the elements in S_i and order them according to V^O to obtain the sequence $\langle v_{i_1}^a, v_{i_1}^b, v_{i_2}^a, v_{i_2}^b, \ldots, v_{i_r}^a, v_{i_r}^b\rangle$. Now we define the line l_i as $\langle v_{i_r}^a, v_{i_r}^b, v_{i_{(r-1)}}^a, v_{i_{(r-1)}}^b, \ldots, v_{i_1}^a, v_{i_1}^b\rangle$. The auxiliary line l_{aux} is defined as $\langle v_{n-1}^b, v_n^a, v_{n-2}^b, v_{n-1}^a, \ldots, v_1^b, v_2^a\rangle$. Observe that the set of arcs A induced by L contains two types of arcs. First, there are arcs of the form (v_i^a, v_i^b) or of the form (v_i^b, v_{i+1}^a) for some $i \in [1, n]$. These are the only arcs in A whose direction agrees with the order V^O, and we refer to them as *forward* arcs. Second, for all the other

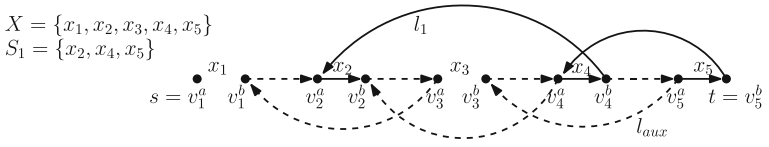

Fig. 4 The correspondence between a set $S_1 \subseteq X$ and a line l_i of the transit network

arcs $(u, v) \in A$ we have $u > v$ with respect to the order V^O, and we refer to these arcs as *backward* arcs. We note that every line l_i is constructed so that the forward arcs of l_i correspond to those elements of X that are contained in S_i, and the backward arcs connect the forward arcs, in the order opposite to V^O, thus making the lines connected. The auxiliary line l_{aux} consists of all the forward arcs of the form (v_i^b, v_{i+1}^a), that are again connected in the opposite order by backward arcs.

Now, for $s = v_1^a$ and $t = v_n^b$, we show that an st-route with a minimum number of different lines in the given transit network G_L provides a minimum SETCOVER for I, and vice versa. Since t is after s in the order V^O, and the only forward arcs in G_L are of the form (v_i^a, v_i^b) or (v_i^b, v_{i+1}^a) for some i, it follows that any route from s to t in G_L goes via all the vertices, in the order V^O. Recall that we defined a route as a pair (π, γ) where π is a *path* and γ is a sequence of lines. Since paths are not allowed to revisit vertices, it even follows that *every* st-route (π, γ) consists of the path $\pi = \langle v_1^a, v_1^b, v_2^a, v_2^b, \ldots, v_n^a, v_n^b \rangle$, and never uses a backward arc (otherwise, there would be a vertex that is visited at least twice, contradicting the assumption that π is a path). In particular, the st-route that minimizes the number of different lines follows this path π. Clearly, l_{aux} must be used in every st-route, as it represents the only way to reach v_{i+1}^a from v_i^b. Now, if a line l_i is used in the st-route r, all the forward arcs in l_i correspond to the arcs (v_i^a, v_i^b) of the path in r and in this way the line l_i "covers" these arcs. Since there is a one to one mapping between the lines $l_1, \ldots, l_m$ and the sets in $\mathscr{S}$, by finding an st-route with $k + 1$ different lines, one finds a solution of size k to the original SETCOVER. Similarly each solution of size k to the original SETCOVER can be mapped to an st-route with $k + 1$ lines. Thus our reduction is approximation preserving (up to an addend $+1$), and based on the inapproximability of SETCOVER [11] this concludes the proof. □

4 Listing all β-bounded Line Sequences

A solution to Problem 3 In [6], the following algorithm for solving Problem 3 is proposed. In a first step, an undirected auxiliary graph G_I is built that contains a vertex for every line $l \in L$, and two vertices l_i and l_j are directly connected if and only if l_i and l_j have at least one common stop. Then a modified breadth-first search is used to compute all line sequences γ for which there exists a route (w, γ) with length at most β. When this modified breadth first search visits an arc (l_i, l_j), it proceeds with l_j only if a transfer from l_i to l_j is possible. We note that unlike in the original definition of a route, in this section we allow the concatenation of the subpaths of the lines in γ to repeat vertices, i.e., here it is sufficient that w is a walk in G_L instead of a path. To avoid the explicit construction of the graph G_I as proposed in [6], we reformulate the algorithm as follows.

The algorithm works recursively and obtains as parameters a partial line sequence $\gamma = \langle l_1, \ldots, l_k \rangle$, $k \leq \beta$, and the earliest vertex u on l_k which can be reached from l_{k-1} among all possible st-routes (π, γ); see Fig. 5 for an example. If l_k visits t after u, then we output the line sequence γ. Moreover, if $k < \beta$, we also check whether γ can be extended. Figure 1 gave an example of a transit network where it is reasonable

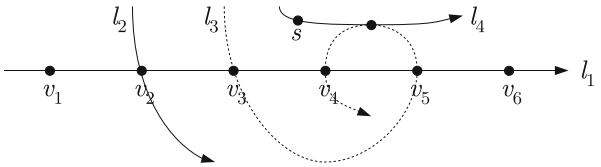

Fig. 5 The earliest transfer from l_3 to l_1 is v_3. However, the earliest transfer using the route $\langle l_4, l_3, l_1\rangle$ is v_4. We have $\nu(l_1, l_2, v_1) = v_2$ and $\nu(l_1, l_2, v_k) = \infty$ for every $k \neq 1$. We have $\nu(l_1, l_3, v_5) = \nu(l_1, l_3, v_6) = \infty$, $\nu(l_1, l_3, v_3) = \nu(l_1, l_3, v_4) = v_5$ and $\nu(l_1, l_3, v_1) = \nu(l_1, l_3, v_2) = v_3$

to extend the current line sequence even if t is reachable via l_k. For extending γ, we compute a set L' of possible line candidates l that can be reached from l_k after u, and also the transfer vertices v_l by which l is reached as early as possible (i.e., there is no vertex v'_l visited by l_k after u that l visits prior to v_l). This can be done by considering the successors of u on l_k with increasing distance from u, and keeping track of the optimal transfer from l_k. Now, for every $l \in L'$, we perform a recursive call with the line sequence $\gamma' = \langle l_1, \ldots, l_k, l\rangle$ and the transfer vertex v_l.

Already the original algorithm (that explicitly constructs the auxiliary graph G_I) works reasonably fast for medium-sized urban transit networks such as the one in Zürich [4], at least if β is relatively small. However observe that the theoretical worst-case running time of the algorithm might be $\Omega(\Delta^\beta)$ where Δ is the largest degree of a vertex in G_I, even if only few (or even none at all) line sequences are output. This is unacceptable from a theoretical point of view, and it might also be unacceptable in a real-world application when both the network and β are large.

An improved solution to Problem 3 To improve the above algorithm, we use an idea similarly used by Rizzi et al. [22] who perform a recursive call only if such a call is guaranteed to output eventually at least one solution. To make use of this idea in the context of listing line sequences, for the transit network G_L, two vertices $v, t \in V$ and a line $l \in L_v$ (the set of all lines that visit v), we define $d^L_{G_L}(v, t, l)$ to be the L-distance from v to t in $G_L = (V, A)$ such that l_j is the first line used. Observe that for a fixed target t, all values $d^L_{G_L}(v, t, l)$ for every $v \in V$ and every $l \in L_v$ can easily be computed using the solution to Problem 1: one simply considers the reverse graph $(G_L)^R$ of G_L (with all the arcs and lines in L reversed), computes $\Gamma[(G_L)^R]$ and runs Dial's algorithm on the vertex t. Then, the length of a shortest path in $\Gamma[(G_L)^R]$ from t to v_{l_j} is exactly $d^L_{G_L}(v, t, l_j)$. A similar idea (traversing the graph $\Gamma[(G_L)^R]$ from the target stop t backwards to compute for every stop v the length of a shortest vt-route) was used in [12]; however, it was not used in the context of listing all length-bounded line sequences but as a speedup heuristic in a multi-criteria version of Dijkstra's algorithm to avoid the generation of unnecessary labels. Here, we also go one step further and take the first line of the corresponding route into account.

To obtain a polynomial-delay listing algorithm for Problem 3, we modify the algorithm described in the previous paragraph as follows. We first compute the values $d^L_{G_L}(v, t, l)$ for every vertex v and every line l that contains v (using the solution to Problem 1). After that, we execute the recursive listing algorithm exactly as described before, but we recursively extend γ by a line l only if $d^L_{G_L}(v_l, t, l) \leq \beta - k$. In such a

case there exists a $v_l t$-route starting with the line l, and extending γ by l gives a route of length at most β. Algorithm 1 shows the details. To solve Problem 3, it is sufficient to invoke LISTLINESEQUENCES$(s, \langle l \rangle)$ for every $l \in L_s$ where $d^L_{G_L}(s, t, l) \leq \beta$. As the following theorem shows, this modified listing algorithm has a polynomial delay.

Algorithm 1 LISTLINESEQUENCES$(u, \langle l_1, \ldots, l_k \rangle)$

```
1 Compute d^L_{G_L}(v,t,l) for each v ∈ G_L and l ∈ L
2 for l ∈ L do v_l ← ∞
3 for each successor v of u on l_k in increasing order do
4     if v = t then OUTPUT(⟨l_1, ..., l_k⟩)
5     for l ∈ L_v do
6         if v_l = ∞ or l visits v earlier than v_l then v_l ← v
7 for l ∈ L do
8     if v_l ≠ ∞ and d^L_{G_L}(v_l,t,l) ≤ β − k then
9         LISTLINESEQUENCES(v_l, ⟨l_1, ..., l_k, l⟩)
```

Theorem 4 *Algorithm 1 has delay $\mathcal{O}(\beta M)$, and its overall time complexity is $\mathcal{O}(\beta M \cdot \kappa)$, where $M = \sum_{l \in L} |l|$ is the input size and κ is the number of returned solutions. Moreover, the space complexity is $\mathcal{O}(M)$.*

Proof As before, step 1 can be computed using $\mathcal{O}(M)$ operations and requires $\mathcal{O}(M)$ space. Steps 3–6 can also be implemented to run in time $\mathcal{O}(M)$, as every step 6 takes only constant time and is performed at most once for every vertex v and every line l containing v. Since the recursive calls in step 9 are only performed if it is guaranteed to output a solution, we observe (similar to [22]) that the height of the recursion tree is bounded by $\mathcal{O}(\beta)$, hence the delay of the algorithm is $\mathcal{O}(\beta M)$. The time complexity of $\mathcal{O}(\beta M \cdot \kappa)$ immediately follows. □

As in [6], we assumed in the above running time analysis that the test whether a given vertex v is visited by a line l can be performed in constant time using suitable hash tables. The same is true for the test whether a line visits a vertex v earlier than some other vertex w. Moreover, when the algorithm is invoked with the parameters $\gamma = \langle l_1, \ldots, l_k \rangle$ and u, and if additionally t is visited by l_k after u, then in practice the successors of t on l_k do not have to be visited any more in step 3 of the Algorithm 1, hence the current call of the algorithm can terminate after γ is output in step 4.

A faster algorithm with preprocessing From a practical point of view, an overall running time of $\mathcal{O}(\beta M \cdot \kappa)$ is still undesirable. Algorithm 1 has delay $\mathcal{O}(\beta M)$ for two reasons: 1) initially we compute the values $d^L_{G_L}(\cdot, t, \cdot)$, and 2) for every partial line sequence $\langle l_1, \ldots, l_k \rangle$ we investigate all possible transfer vertices from l_k to other lines to find the optimal one for every line. Issue 1) can easily be solved by computing the values $d^L_{G_L}(v, t, l)$ for every $v, t \in V$ and every line $l \in L$ in advance and then storing them. Since for every t there are at most $\mathcal{O}(M)$ many values $d^L_{G_L}(\cdot, t, \cdot)$ and

all of them can be computed in time $\mathcal{O}(M)$, we need overall time $\mathcal{O}(M|V|)$. To solve issue 2), we precompute for every line l, every vertex v on l and every line $l' \neq l$ the vertex $f(l, l', v)$ which is visited by l after v and by which l' is reached as early as possible (on l'). If no such a vertex exists, we set $f(l, l', v) = \infty$. For every line $l = \langle v_1, \ldots, v_k \rangle \in L$, the values $f(l, \cdot, \cdot)$ can be computed as follows. We consider the vertices $v_k, \ldots, v_1$ in this order. We set $f(l, l', v_k) = \infty$ for every $l' \in L$. After that, considering a vertex v_i, we set

$$f(l, l', v_i) = \begin{cases} v_{i+1} & \text{if } l' \in L_{v_{i+1}} \text{ and } f(l, l', v_{i+1}) = \infty \\ v_{i+1} & \text{if } l' \in L_{v_{i+1}} \text{ and } l' \text{ visits } v_{i+1} \text{ before } f(l, l', v_{i+1}) \\ f(l, l', v_{i+1}) & \text{otherwise} \end{cases} \tag{2}$$

Since the computation of each entry requires only constant time, the values $f(l, l', v)$ can be computed using time and space $\mathcal{O}(M|L|)$. Hence, for preprocessing time and space $\mathcal{O}(M(|V| + |L|))$ suffice. Now, however, st-route listing queries can be performed much faster using the following algorithm. Figure 5 gives an example for a transit network and the corresponding values $v(l, l', v_i)$.

Algorithm 2 ListLineSequences$(u, \langle l_1, \ldots, l_k \rangle)$

```
1 if l_k visits t after u then Output(⟨l_1, ..., l_k⟩)
2 for l ∈ L do
3     v_l ← f(l_k, l, u)
4     if v_l ≠ ∞ and d^L_{G_L}(v_l, t, l) ≤ β − k then
5         ListLineSequences(v_l, ⟨l_1, ..., l_k, l⟩)
```

Theorem 5 *The values $d^L_{G_L}(v, t, l)$ and $f(l, l', v)$ can be precomputed using time and space $\mathcal{O}(M(|V| + |L|))$. Assuming that these values have been precomputed, Algorithm 2 has delay $\mathcal{O}(\beta|L|)$, and its overall time complexity is $\mathcal{O}(\beta|L| \cdot \kappa)$, where $|L|$ is the number of lines and κ is the number of returned solutions.*

Proof The straightforward proof is similar to the proof of Theorem 4. □

To see the speedup, remember that Algorithm 1 has a delay of $\mathcal{O}(\beta M)$ while Algorithm 2 has a delay of only $\mathcal{O}(\beta|L|)$. In real transit networks, $M = \sum_{l \in L} |l|$ is usually way larger than $|L|$ is.

5 Modeling Consequences: Line Sequences vs. Routes

In the following we discuss some consequences of the way how we modeled transit networks. In particular we consider situations that arise when modeling the transit network of Zürich, because this network will be used later as a basis for the experimental evaluation of our algorithms. The most critical point in our model is our definition of a line as an (ordered) sequence of stops. Although no two lines have the same stop sequence, they may have the same identifier (such as "Bus 31") in reality

and might therefore be considered equivalent from a travelers's perspective. Figure 6 gives an example for various lines that might be operated under the same identifier which we now explain in more detail.

- *Standard realization of a line.* In most cases, a line $l = \langle v_1, \ldots, v_k \rangle$ is realized throughout the whole day with high frequency. This line corresponds to $l_{standard}$ in Fig. 6. Often there exists a similar line (a backward line) in the opposite direction $l' = \langle v_k, \ldots, v_1 \rangle$ which contains the same stops as l but in reverse order. In some rare cases, the stop sequence of the backward line slightly differs. This mostly happens when busses travel along one-way streets.
- *Standard realization changes over the day.* In most cities there exist observable patterns of how people use public transportation. During the work days, there are clearly visible peak hours at which people commute to or from work. The planned schedule of public transportation services usually tries to react to increased or decreased demand. As a consequence, the planned frequency of a line may change during the day. Sometimes, however, also the realization of the line itself changes, e.g. because in the evening certain stops are completely left out and some other stops are visited instead. Hence we have to introduce a new line to model such a situation. In Fig. 6, $l_{evening}$ is an example for such a line.

  Such lines clearly introduce difficulties in the context of finding *robust* routes: if, at some point, the realization of a line changes from $l_{standard}$ to $l_{evening}$ and a traveler misses the last trip of $l_{standard}$, then following the earlier computed route does not lead to an arrival at the target stop on that day. In reality, however, if one does not want to travel to one of the stops that is left out, one could have just used $l_{evening}$ instead.
- *A vehicle comes from or goes to the depot.* At the beginning of an operational day (and shortly before the beginning of the rush hours), vehicles come from the depot to start the service. Analogously, at the end of the operational day (and after the rush hours) vehicles travel back to the depot. Often these lines can also be used for traveling, but their line sequence may differ considerably from the standard realization. The low frequency of such lines makes an inclusion of them into a robust route highly undesirable. For example, consider Fig. 6 and assume that we want to travel from y to f. It might be tempting to use l_2 until a, and from there use l_{depot}. Now, if the planned trip of l_2 is slightly delayed and the

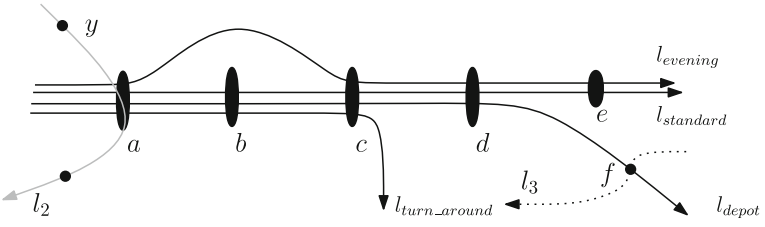

Fig. 6 Different lines with the same identifier. The line $l_{standard}$ describes the most frequent realization. The line $l_{evening}$ leaves a stop out, but this happens only in the evening. Towards the end of the operational day, the line l_{depot} goes back to the depot. Sometimes early in the morning, busses by plan turn around in advance such that there is a fast and good coverage of vehicles. An example for such a line is l_{turn_around}

transfer to l_{depot} fails, one cannot continue the recommended journey until the next trip of l_{depot} arrives, which might be only a few hours later (or even only on the next day).

– *Cyclic lines.* In some rare cases, a line l does not have a corresponding backward line going in the opposite direction. Instead, after visiting the last stop on l the realizing vehicle travels back to the first stop of l, hence l has a cyclic topology. Such a situation is rather difficult to capture, because in a journey it might be necessary to take two consecutive trips of the same line. Although the trip is changed, the transfer should not be counted because it is the same physical vehicle. One way out might be to concatenate l with itself, but then we violate our previously made assumption that every line visits every stop of the network at most once. This assumption was not used in Algorithm 1, but it is crucial for the correctness of Algorithm 3 that is discussed in the next section.

  However, the algorithms itself can be modified to handle such a situation. For that, we could also give an additional number as a parameter to each recursive call that counts how many real transfers have already been used. At the end of a cyclic line we also must allow a transfer to the line itself, and the aforementioned counter is increased only if the current transfer to a "new" line is also a transfer in reality. Moreover, we would have to modify the auxiliary graph $\Gamma[G_L]$: For every cyclic line $l = \langle v, \ldots, w \rangle$ (where v is visited directly after w again), we have to insert an additional arc from v_l to w_l (with cost 0). A similar modification can be used to model the situation when vehicles serve multiple lines $l_1, l_2, l_3, \ldots$ one after another, and no physical transfer from l_i to l_{i+1} is necessary to change the line (because one can simply stay in the vehicle). One just has be careful that such a "transfer" can take place only at the end of a line. However, adapting the implementation to consider such special situations might be more a technical than a conceptional problem. Therefore we don't discuss this issue further.

– *A vehicle turns back in advance.* Shortly after the beginning of an operational day, one especially tries to spread vehicles as fast as possible such that many stops are covered. Therefore, vehicles sometimes are planned to turn around in advance such that the most important stops of that line are covered already early in the morning. Since this behavior can be found in the planned timetable, we introduce an additional line (e.g., l_{turn_around} in Fig. 6).

Beside the fact that these lines are operated under the same identifier, they also have many common stops. In urban areas one generally can often observe that different lines have more than just one common stop. Assume, e.g., that two lines l_i and l_j have the common stops $v_1, \ldots, v_k$. From a traveler's perspective, when traveling from any of these stops to any other of these stops, one may either use l_i or l_j, whatever arrives first. Figure 7 shows a real-life example from Zürich, where much more line sequences than actual paths in G_L exist. Hence, considering paths in transit networks instead of line sequences seems to be more natural because it allows to consider different lines as one joint line. In some way, computing journeys from paths was already proposed in [24]; there, however, paths are not listed explicitly, and they are bounded by their geographical length instead of by the number of transfers.

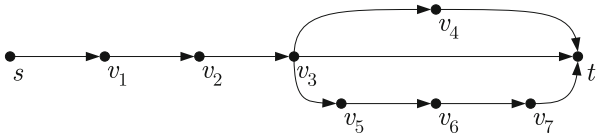

Fig. 7 An extract of the transit network in Zürich when traveling from s = Sihlpost to t = ETH/Universitätsspital. The lines are $l_{PB} = \langle v_3, t\rangle$, $l_3 = l_{31} = \langle s, v_1, v_2, v_3, v_5, v_6\rangle$, $l_6 = \langle v_2, v_3, v_4, t\rangle$, $l_9 = \langle v_6, v_7, t\rangle$, $l_{10} = \langle v_1, v_2, v_3, v_4, t\rangle$, and $l_{14} = \langle s, v_1, v_2\rangle$. There is no direct connection between s and t, but there are 10 line sequences γ with $|\gamma| = 2$ for which G_L contains a route (π, γ). However, observe that only three line sequences differ substantially (i.e., have different paths in the network)

6 Listing all β-bounded Paths

Motivation We saw in the previous section that listing all length-bounded paths in G_L seems more natural than listing all st-routes. A naïve approach for solving Problem 4 is to use Algorithm 1 or 2 to generate all feasible line sequences γ and then to compute the corresponding paths (there might be more than one) for each feasible γ. However, this approach does not only have the disadvantage of a possibly huge running time, also for every path π there might be exponentially many line sequences γ such that (π, γ) is route in G_L. Since we want to output every path π at most once, we would also need to store $\Omega(\kappa)$ many paths. To some extent, such an approach was used by Xu et al. [26] whose algorithm outputs each path as often as some route realizes this path (instead of just once), and also only lists paths with no more than two transfers.

Another straightforward idea to solve Problem 4 might be to construct an auxiliary graph from G_L and then use one of the well-known algorithms for listing paths, e.g., Yen's algorithm. For example, one could create a directed graph $G_X = (V, A_X)$ where A_X contains an arc between v and w if there exists a line that visits v before w. Any path between s and t in G_X of length at most β induces an st-path in G_L. However, as before, exponentially many paths in G_X might correspond to one path in G_L which again might lead to an exponential gap between the running time and the sum of the input and the output size.

Improved idea for Problem 4 Let $\mathscr{P}_{st}^{\beta}(L)$ denote the set of all st-paths π such that there exists a route (π, γ) with length at most β in the transit network G_L. To obtain a polynomial delay algorithm that uses only $\mathscr{O}(M)$ space, we use the so-called *binary partition method* described in [3, 22]: The transit network G_L is traversed in a depth first search fashion starting from s, and the solution space $\mathscr{P}_{st}^{\beta}(L)$ is recursively partitioned at every call until it contains exactly one solution (i.e., one path) that is then output.

When the algorithm considers a partial su-path π_{su}, we first check whether $u = t$. In that case, π_{su} is output. Otherwise, we compute the graph G' that is the transit network G_L from which all vertices (and all adjacent arcs) in π_{su} are removed. To bound the running time of the algorithm we maintain the invariant that the current partition (i.e., the paths in $\mathscr{P}_{st}^{\beta}(L)$ with prefix π_{su}) contains at least one solution. More concretely, we require that G' contains at least one ut-path π_{ut} that extends

π_{su} so that $\pi_{su} \cdot \pi_{ut} \in \mathscr{P}^{\beta}_{st}(L)$. The idea behind this algorithm is similar to the one in [22] for listing all α-bounded paths; here, however, new ideas to maintain the invariant are necessary because our objective is to list only paths π for which a length-bounded route (π, γ) in G_L exists (instead of listing all paths whose length itself is bounded).

Checking whether to recurse or not Let π_{su} be the su-path that the algorithm currently considers, $L' = L - \pi_{su}$, $G' = G_L - \pi_{su} = G_{L'}$, and $v \in N^+_{G_L}(u) \cap G'$, i.e., v is a neighbor of u that is not contained in π_{su}. We recursively continue on $\pi_{su} \cdot (u, v)$ only if the invariant (I) is satisfied, i.e., if $\mathscr{P}^{\beta}_{st}(L)$ contains a path with prefix $\pi_{su} \cdot (u, v)$.

Let $d_{G_L}(\pi_{su}, (u, v), l_i)$ be the length of a minimum route $(\pi_{su} \cdot (u, v), \gamma)$ in G_L such that l_i is the last line of γ. Let $d^{L'}_{G'}(v, t, l_j)$ be the L'-distance from v to t in G' such that l_j is the first line used. For a vertex $v \in V$, let $L_v \subseteq L$ be the set of all lines that contain an outgoing arc from v. Analogously, for an arc $(u, v) \in A$, let $L_{(u,v)}$ be the set of all lines that contain (u, v). Now, the set $\mathscr{P}^{\beta}_{st}(L)$ contains a path with prefix $\pi_{su} \cdot (u, v)$ if and only if

$$\min \left\{ d_{G_L}(\pi_{su}, (u, v), l_i) - \delta_{ij} + d^{L'}_{G'}(v, t, l_j) \mid l_i \in L_{(u,v)} \text{ and } l_j \in L_v \right\} \leq \beta \,. \quad (3)$$

Basically, $\min\{d_{G_L}(\pi_{su}, (u, v), l_i) - \delta_{ij} + d^{L'}_{G'}(v, t, l_j) \mid l_i \in L_{(u,v)} \text{ and } l_j \in L_v\}$ is the length of the minimum route that has prefix $\pi_{su} \cdot (u, v)$.

Computing $d_{G_L}(\pi_{su}, (u, v), l_i)$ and $d^{L'}_{G'}(v, t, l_j)$ We can use the solution for Problem 1 to compute the values $d_{G_L}(\pi_{su}, (u, v), l_i)$ and $d^{L'}_{G'}(v, t, l_j)$. The values $d_{G_L}(\pi_{su}, (u, v), l_i)$ need to be computed only for arcs $(u, v) \in A$ with $v \notin \pi_{su}$ (i.e., only for arcs from u to a vertex $v \in N^+_{G_L}(u) \cap G'$), and only for lines $l_i \in L_{(u,v)}$. Consider the graph G'' that contains every arc from π_{su} and every arc $(u, v) \in A$ with $v \notin \pi_{su}$, and that contains exactly the vertices incident to these arcs. Now we compute $H = \Gamma[G'']$ and run Dial's algorithm on the vertex s. For every $v \in N^+_{G_L}(u) \cap G'$ and every line $l_i \in L_{(u,v)}$, the length of a shortest path in H from s to v_{l_i} is exactly $d_{G_L}(\pi_{su}, (u, v), l_i)$. For computing $d^{L'}_{G'}(v, t, l_j)$, we can consider the L'-distances from t in the reverse graph G'^R (with all the arcs and lines in L' reversed). Considering G' instead of G_L ensures that lines do not use vertices that have been deleted in previous recursive calls of the algorithm. Thus we compute $\Gamma[G'^R]$ and run Dial's algorithm on the vertex t. Then, the length of a shortest path in $\Gamma[G'^R]$ from t to v_{l_j} is exactly $d^{L'}_{G'}(v, t, l_j)$.

Algorithm Algorithm 3 shows the details of the aforementioned approach. To limit the space consumption of the algorithm, we do not pass the graph G' as a parameter to the recursive calls, but compute it at the beginning of each recursive call from the current prefix π_{su}. For the same reason, we do not perform the recursive calls immediately in step 8, but first create a list $V_R \subseteq V$ of vertices for which the invariant (I) is satisfied, and only then recurse on $(v, \pi_{su} \cdot (u, v))$ for every $v \in V_R$. To list all paths in $\mathscr{P}^{\beta}_{st}$, we invoke ListPaths$(s, \langle s \rangle)$.

Algorithm 3 ListPaths(u, π_{su})

```
1  if u = t then Output(π_su); return
2  L' ← L − π_su; G' ← G_L − π_su
3  Compute d_{G_L}(π_su, (u,v), l_i) for each v ∈ N⁺_{G_L}(u) ∩ G' and l_i ∈ L_(u,v)
4  Compute d^{L'}_{G'}(v,t,l_j) for each v ∈ N⁺_{G_L}(u) ∩ G' and l_j ∈ L_v
5  V_R ← ∅
6  for v ∈ N⁺_{G_L}(u) ∩ G' do
7      d ← min{d_{G_L}(π_su, (u,v), l_i) + d^{L'}_{G'}(v,t,l_j) − δ_ij | l_i ∈ L_(u,v) and l_j ∈ L_v}
8      if d ≤ β then V_R ← V_R ∪ {v}
9  for v ∈ V_R do
10     ListPaths(v, π_su · (u,v))
```

Theorem 6 *Algorithm 3 has delay $\mathcal{O}(nM)$, where n is the number of vertices in G_L and $M = \sum_{l \in L} |l|$ is the input size. The overall time complexity is $\mathcal{O}(nM \cdot \kappa)$, where κ is the number of returned solutions. Moreover, the space complexity is $\mathcal{O}(M)$.*

Proof We first analyze the cost of a given call to the algorithm without including the cost of the recursive calls performed inside. Theorem 2 states that steps 3 and 4 can be performed in time $\mathcal{O}(M)$. We will now show that steps 6–8 can be implemented in time $\mathcal{O}(M)$. Notice that for a fixed prefix π_{su} and a fixed vertex $v \in N^+_{G_L}(u) \cap G'$, for computing the minimum in step 7, we need to consider only the values $d_{G_L}(\pi_{su}, (u, v), l_i)$ that are minimum among all $d_{G_L}(\pi_{su}, (u, v), \cdot)$, and only the values $d^{L'}_{G'}(v, t, l_j)$ that are minimum among all $d^{L'}_{G'}(v, t, \cdot)$. Let $\Lambda_v \subseteq L_{(u,v)}$ be the list of all lines l_i for which $d_{G_L}(\pi_{su}, (u, v), l_i)$ is minimum among all $d_{G_L}(\pi_{su}, (u, v), \cdot)$. Analogously, let $\Lambda'_v \subseteq L_v$ be the list of all lines l_j for which $d^{L'}_{G'}(v, t, l_j)$ is minimum among all $d^{L'}_{G'}(v, t, \cdot)$. Let

$$\mu_v = \min \left\{ d_{G_L}(\pi_{su}, (u, v), l_i) \mid l_i \in \Lambda_v \right\} \tag{4}$$

$$\mu'_v = \min \left\{ d^{L'}_{G'}(v, t, l_j) \mid l_j \in \Lambda'_v \right\} \tag{5}$$

be the minimum values of $d_{G_L}(\pi_{su}, (u, v), \cdot)$ and $d^{L'}_{G'}(v, t, \cdot)$, respectively. Both values as well as the lists Λ_v and Λ'_v can be computed in steps 3 and 4, and their computation only takes overall time $\mathcal{O}(M)$. Now the expression in step 7 evaluates to $\mu_v + \mu'_v$ if $\Lambda_v \cap \Lambda'_v = \emptyset$, and to $\mu_v + \mu'_v - 1$ otherwise. Assuming that Λ_v and Λ'_v are ordered ascendingly by the index of the contained lines l_i, it can easily be checked with $|\Lambda_v| + |\Lambda'_v| \leq |L_{(u,v)}| + |L_v|$ many comparisons if their intersection is empty or not. Using this method, each of the values $d_{G_L}(\pi_{su}, \cdot, \cdot)$ and $d^{L'}_{G'}(\cdot, t, \cdot)$ is accessed exactly once (when computing Λ_v and Λ'_v), and since each of these values has a unique corresponding vertex in the graphs H and $\Gamma[G'^R]$, there exist at most $\mathcal{O}(M)$ many such values. Thus, the running time of the steps 6–8 is bounded by $\mathcal{O}(M)$ which is also an upper bound on the running time of Algorithm 3 (ignoring the recursive calls).

To prove the delay of $\mathcal{O}(nM)$, we again use an argument similar to one in [22]: since every recursive call appends one vertex to the current partial path π_{du} and the length of π_{du} is bounded by n, the height of the recursion tree is bounded by n. Hence, after a path is output, at most n many previous recursive calls terminate and at most n many new recursive calls are invoked until the next path is output.

For analyzing the space complexity, observe that both L' and G' as well as all values $d_{G_L}(\pi_{su}, (u, v), l_i)$ and $d_{G'}^{L'}(v, t, l_j)$ can be removed from the memory after step 8 since they are not needed any more. Thus, we only need to store the lists V_R between the recursive calls. Consider a path in the recursion tree, and for each recursive call i, let u^i be the vertex u and V_R^i be the list V_R of the i-th recursive call. Since V_R^i contains only vertices adjacent to u^i and u^i is never being considered again in *any* succeeding recursive call $j > i$, we have

$$\sum_i |V_R^i| \leq |A_L|, \tag{6}$$

which proves the space complexity of $\mathcal{O}(M)$. □

7 Experimental Evaluation

We implemented Algorithm 3 to investigate its running time and the number of paths it computes. We used the data from the transit network of Zürich, Switzerland, including trams and busses. The network has 401 stops and 292 lines (of which many are operated under the same identifier, see Section 5 for an explanation on the reasons). The algorithm was implemented in Java 8, and it was executed on one core of an Intel Core i7-6700 CPU. Even though most modern CPUs have more than one core and the algorithm seems to be easily parallelizable, we implemented only a serial version of the algorithm for two reasons: first, our goal was to investigate how fast the algorithm can be even under "bad" conditions, for example when the algorithm was used in a web application and there were more queries than available cores. Moreover, in practical applications it might be sufficient to compute only few (e.g., 10) alternative paths. Since the algorithm turned out to perform sufficiently well when only few paths are computed (see below), the additional overhead might reduce the speedup achieved by the parallelization.

We generated 1'000 st-pairs (with $s \neq t$ and $d_L(s, t) \leq 3$) uniformly at random, and computed the set $\mathscr{P}_{st}^{\beta}(L)$ for every $\beta \in \{d_L(s, t), d_L(s, t) + 1, d_L(s, t) + 2\}$ (i.e., we computed all paths along routes with a minimum number of transfers, all paths along routes that may use one additional transfer and all paths along routes that may use two additional transfers). The sets $\mathscr{P}_{st}^{\beta}(L)$ contained between 1 and 8,365 paths, and the running times for computing these paths ranged between less than one millisecond and 202 seconds. Figure 8 (left) shows a more detailed overview. For each of the 3,000 experimental outcomes (1,000 st-pairs with three values of β each), the figure contains a point of this outcome where the x-coordinate denotes the number of solutions and the y-coordinate the corresponding running time of the algorithm (in seconds). One can clearly see that the running time and the number of

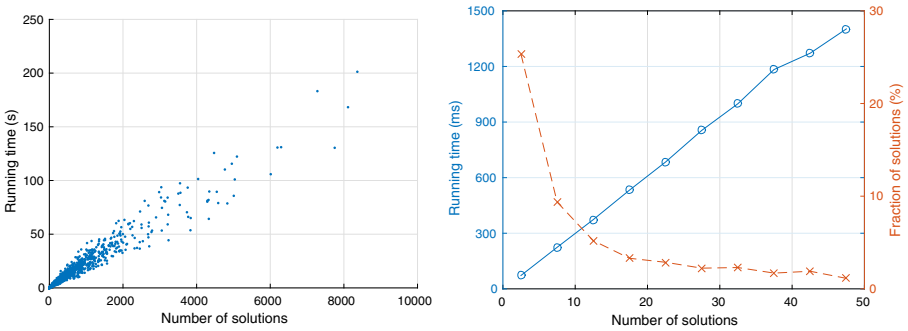

Fig. 8 The running time of the algorithm in comparison to the number of solutions output (*left*). Average running times and the correspoding fraction of outcomes when outcomes are grouped by the number of computed solutions in buckets of length 5 (*right*)

solutions are correlated, and that there are relatively few outcomes where the running time exceeds a minute.

From a practical point of view, even query times in the magnitude of seconds are unacceptable. We therefore grouped all outcomes according to the number of solutions computed into buckets of 5 solutions, i.e., the first bucket contains all outcomes where the number of computed solutions is not larger than 5, the second bucket contains all outcomes where the number of computed solutions ranges between 6 and 10, and so on. Then, for every bucket we computed the average running time of the outcomes in this bucket and also related their number to the overall number of outcomes. Figure 8 (right) shows the results for the outcomes which computed not more than 50 paths. First, one again can observe a linear correlation between the number of computed solutions and the corresponding running times. The average running time remains below 1.5s and even below 250ms when 10 or less paths are computed. Notice that more than 25 % of all outcomes had 5 or less paths, and fewer than 45 % had more than 50 paths.

To investigate how often this is the case, we grouped our $1'000$ st-pairs into three groups containing all pairs with $d_L(s,t) = 1$ (i.e., pairs between which a direct connection exists), $d_L(s,t) = 2$ (i.e., pairs which require at least one transfer) and $d_L(s,t) = 3$ (i.e., pairs which require at least two transfers). We note that in Zürich there are only few pairs that require more than three transfers. Now, for each group, we computed the minimum, average and maximum number as well as the median, the 10th and the 90th percentile of solutions over all outcomes that use zero, one and two additional transfers, respectively. Figure 9 shows the results for the st-pairs with $d_L(s,t) = 1$ (left), $d_L(s,t) = 2$ (middle) and $d_L(s,t) = 3$ (right). One can observe that allowing one more transfer than necessary leads to a moderate increase of the number of computed paths while allowing two additional transfers increases the number of computed paths drastically.

Again, we should take a closer look at the concrete numbers. We observed that for all st-pairs with $d_L(s,t) = 1$, the maximum number of computed paths with $\beta = 2$ is 39. Moreover, the median number of computed paths with $\beta = 3$ is 38, while the average is below 65. For the st-pairs with $d_L(s,t) = 2$, the median and

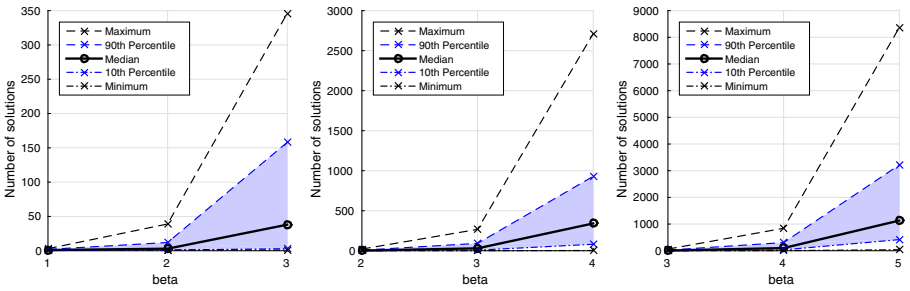

Fig. 9 Minimum and maximum as well as the median, the 10th and the 90th percentile of solutions over all outcomes that use zero, one and two additional transfers. The st-pairs have $d_L(s,t) = 1$ (*left*) $d_L(s,t) = 2$ (*middle*) and $d_L(s,t) = 3$ (*right*)

average number of solutions with $\beta = 3$ are 30 and 41.27, respectively, and for the st-pairs with $d_L(s,t) = 3$, the median number of solutions with $\beta = 3$ is 7 while the average is 10.29. Although alternative ways to travel are clearly useful, in practice it seems unlikely that one needs more than 10 alternative solutions. Hence, it seems that allowing three transfers is likely to give a fair compromise between the number of solutions output and the running time. A more sophisticated way might be the following: one first computes $\mathscr{P}_{st}^{\beta}(L)$ for $\beta = d_L(s,t)$, and if the number of computed paths is below a fixed threshold (e.g., 10), one successively increases β until the number of solutions is sufficiently large. Since also in practice the running time seems to linearly depend on the number of computed solutions, this might be feasible for real-life applications.

8 Conclusion

In this paper, we studied the problem of computing and listing st-paths in transit networks. As a first result, we showed how the existing listing algorithm in [6] for listing all line sequences γ for which a length-bounded route along γ exists can be modified such that the worst-case running time becomes polynomial in the sum of the input and the output size. Moreover, we argued that many routes might correspond to the same path in the transit network, and might therefore be considered equivalent from a traveler's perspective. Therefore we proposed an algorithm to compute the set of all paths in the transit network for which a suitable route using at most β lines exist. The running time of this algorithm again is polynomial in the sum of the input and the output size. Our preliminary experimental study showed that the real running time of the algorithm indeed linearly depends on the number of solutions output, and we also saw that the algorithm is sufficiently fast for real applications if the number of computed paths is small (i.e., not above 10). However, it would certainly be worthwhile to further study whether heuristical speedup techniques (such as pruning based on the geography of the stops) helps to decrease the running time, and how such techniques influence the quality of the solutions.

Acknowledgements We thank the anonymous reviewers for pointing out how the running times of our listing algorithms can be improved by a factor of $\Theta(\log M)$ and for providing additional valuable feedback. Furthermore we thank Peter Widmayer for many helpful discussions.

References

1. Bast, H., Delling, D., Goldberg, A.V., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., Werneck, R.F.: Route planning in transportation networks (2015). CoRR arXiv:1504.05140
2. Bezem, G., Leeuwen, J.V.: Enumeration in graphs. Tech. Rep. RUU-CS-87-07. Utrecht University (1987)
3. Birmelé, E., Ferreira, R.A., Grossi, R., Marino, A., Pisanti, N., Rizzi, R., Sacomoto, G.: Optimal listing of cycles and st-paths in undirected graphs. In: SODA 2013, pp. 1884–1896 (2013)
4. Böhmová, K., Mihalák, M., Neubert, P., Pröger, T., Widmayer, P.: Robust routing in urban public transportation: Evaluating strategies that learn from the past. In: ATMOS 2015, pp. 68–81 (2015)
5. Böhmová, K., Mihalák, M., Pröger, T., Sacomoto, G., Sagot, M.: Computing and listing st-paths in public transportation networks. In: CSR 2016, pp. 102–116 (2016)
6. Böhmová, K., Mihalák, M., Pröger, T., Šrámek, R., Widmayer, P.: Robust routing in urban public transportation: How to find reliable journeys based on past observations. In: ATMOS 2013, pp. 27–41 (2013)
7. Böhmová, K., Mihalák, M., Pröger, T., Widmayer, P.: Modelling urban public transportation networks to support robust routing. In: CASPT 2015 (2015)
8. Brodal, G.S., Jacob, R.: Time-dependent networks as models to achieve fast exact time-table queries. Electr. Notes Theor. Comput. Sci **92**, 3–15 (2004)
9. Dial, R.B.: Algorithm 360: shortest-path forest with topological ordering. Commun. ACM **12**(11), 632–633 (1969)
10. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
11. Dinur, I., Steurer, D.: Analytical approach to parallel repetition. In: STOC 2014, pp. 624–633 (2014)
12. Disser, Y., Müller-Hannemann, M., Schnee, M.: Multi-criteria shortest paths in time-dependent train networks. In: WEA 2008, pp. 347–361 (2008)
13. Eppstein, D.: Finding the k shortest paths. SIAM J. Comput. **28**(2), 652–673 (1998)
14. Firmani, D., Italiano, G.F., Laura, L., Santaroni, F.: Is timetabling routing always reliable for public transport? In: ATMOS 2013, pp. 15–26 2013. doi:10.4230/OASIcs.ATMOS.2013.15
15. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3), 596–615 (1987)
16. Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM J. Comput. **4**(1), 77–84 (1975)
17. Johnson, D.S., Papadimitriou, C.H., Yannakakis, M.: On generating all maximal independent sets. Inf. Process. Lett. **27**(3), 119–123 (1988)
18. Katoh, N., Ibaraki, T., Mine, H.: An efficient algorithm for *k* shortest simple paths. Networks **12**(4), 411–427 (1982)
19. Lawler, E.L.: A procedure for computing the *k* best solutions to discrete optimization problems and its application to the shortest path problem. Mgmt. Sci **18**, 401–405 (1972)
20. Müller-Hannemann, M., Schulz, F., Wagner, D., Zaroliagis, C.D.: Timetable information: Models and algorithms. In: ATMOS 2004, pp. 67–90 (2004)
21. Nguyen, S., Pallottino, S., Gendreau, M.: Implicit enumeration of hyperpaths in a logit model for transit networks. Transp. Sci. **32**(1), 54–64 (1998)
22. Rizzi, R., Sacomoto, G., Sagot, M.: Efficiently listing bounded length st-paths. In: IWOCA 2014, pp. 318–329 (2014)
23. Schulz, F., Wagner, D., Zaroliagis, C.D.: Using multi-level graphs for timetable information in railway systems. In: ALENEX 2002, pp. 43–59 (2002)
24. Tulp, E., Siklóssy, L.: Trains, an active time-table searcher. In: ECAI 1988, pp. 170–175 (1988)
25. Vo, K.D., Pham, T.V., Nguyen, H.T., Nguyen, N., Hoai, T.V.: Finding alternative paths in city bus networks. In: IC3INA 2015, pp. 34–39 (2015)
26. Xu, W., He, S., Song, R., Chaudhry, S.S.: Finding the K shortest paths in a schedule-based transit network. Comput. OR **39**(8), 1812–1826 (2012)

27. Yen, J.Y.: Finding the k shortest loopless paths in a network. Mgmt. Sci. **17**, 712–716 (1971)
28. Yuan, S., Varma, S., Jue, J.P.: Minimum-color path problems for reliability in mesh networks. In: INFOCOM 2005, pp. 2658–2669 (2005)