On finding k earliest arrival time journeys in public transit networks

David Coudert, Ali Al Zoobi, Arthur Finkelstein

On finding k earliest arrival time journeys in public transit networks*

David Coudert[1], Ali Al Zoobi[1], and Arthur Finkelstein[2]

[1]Université Côte d'Azur, Inria, I3S, CNRS, France
[2]INSTANT System

June 18, 2021

Abstract

Journey planning in (schedule-based) public transit networks has attracted interest from researchers in the last decade. In particular, many algorithms aiming at efficiently answering queries of journey planning have been proposed. However, most of the proposed methods give the user a single or a limited number of journeys in practice, which is undesirable in a transportation context.

In this paper, we consider the problem of finding k earliest arrival time journeys in public transit networks from a given origin to a given destination, i.e, an earliest arrival journey from the origin to the destination, a second earliest arrival journey, *etc.* until the k^{th} earliest arrival journey.

For this purpose, we propose an algorithm, denoted by Yen - Public Transit (Y-PT), that extends to public transit networks the algorithm proposed by Yen to find the top-k shortest simple paths in a graph. Moreover, we propose a more refined algorithm, called Postponed Yen - Public Transit (PY-PT), enabling a considerable speed up in practice.

Our experiments on several public transit networks show that, in practice, PY-PT is faster than Y-PT by an order of magnitude.

Keywords: Journey planning, shortest path, routing, timetables.

1 Introduction

In the context of multimodal transportation, journeys planning in (schedule-based) public transit networks and accelerating queries for efficient journey planning is a long-standing problem [7]. In the last decade, many algorithms have been developed not only to answer efficiently basic queries like a quickest or an earliest arrival journey, but also to optimize additional criteria like the number of transfers, the cost of the trip, *etc.* or even to offer Pareto optimal solutions combining several criteria [7, 9, 10].

A *transit network* is a set of stops (such as bus stops or trains stations), a set of routes (such as bus, tramway, ferries, metro or train lines), and a set of trips. Trips correspond to individual vehicles that visit the stops along a certain route at a specific time of the day. Trips can be further subdivided into sequences of elementary connections, each given as a pair

*This work has been supported by the French government, through the UCA$^{\text{JEDI}}$ Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-15-IDEX-01, the ANR project MULTIMOD with the reference number ANR-17-CE22-0016, the ANR project Digraphs with the reference number ANR-19-CE48-0013, and by Région Sud PACA.

of (origin/destination) stops and (departure/arrival) times between which the vehicle travels without stopping. In addition, footpaths model walking transfers between nearby stops. A *journey* is a sequence of trips one can take within a transit network (also referred to as a transportation network or a *timetable*).

The k shortest simple paths problem A directed graph (*digraph* for short) is a set of vertices connected by arcs. A *path* from a source to a destination in a digraph is a sequence of vertices starting from the source and ending at the destination, such that consecutive vertices are connected by an arc. A path is simple if it has no repeated vertices. The length (or weight or cost) of a path is the sum of the lengths (or weights or costs) of its arcs. In this context, the k shortest simple paths (kSSP) problem asks to find a set S of k pairwise different simple paths from a source to a destination such that no path outside S has a length strictly less than a path in S. This problem can be solved in time $O(kn(m + n \log n))$ using the algorithm proposed by Yen [17], where n is the number of vertices and m is the number of arcs. Since this is the best known time complexity for this problem, a significant research effort has been put on the design of algorithms for efficiently solving the kSSP problem in practice [1, 2, 13]. Note that, if the paths of S are not required to be simple, the problem can be solved by Eppstein's algorithm in time $O(k + m + n \log n)$ [11].

In fact, a road network can be modelled using a weighted directed graph where crossroads are represented by vertices and routes by arcs with length corresponding to the distances or the travel time between crossroads. So, finding k "best" (shortest, fastest or cheapest) paths from a given origin to a given destination in a road network is straightforward using any kSSP algorithm. Unfortunately, this problem becomes harder in public transit networks. First, because public transit networks are time dependent, i.e., certain segments of the network can only be traversed at specific times. Second, several additional optimization criteria are considered in public transit network such as the arrival time, the departure time, the number of transfers, *etc.*

Journey planning queries in public transit networks Plethora of algorithms were proposed to efficiently answer queries of optimal journeys from a given origin o to a given destination d after a departure time t_0 in a public transit network. For instance, the *Connection Scan Algorithm (CSA)* [10] is the fastest algorithm, without any preprocessing routine, enabling to find an earliest arrival journey from o to d departing after t_0. With the help of a heavy preprocessing routine, the Transfer Patterns algorithm [6] can achieve a tremendous speed up with respect to the CSA. Besides, *Round Based Public Transit Routing (RAPTOR)* [9] is the fastest algorithm (also without any preprocessing routine) enabling to compute a Pareto optimal set of journeys optimizing the arrival time and the number of transfers of a journey. Bast et al.Recently, [7] presented an extensive survey on the topic of journey planning in road and public transit networks.

Related work Vo et al. [16] proposed a time dependent graph modelizing a bus network. Then, they adapt Yen's algorithm to find alternative journeys in this network model. Precisely, they select a set of alternative journeys (journeys sharing only a limited part of their common edges) among those given by Yen's adaptation.

As shown below, Yen's algorithm uses Dijkstra's algorithm as a basic brick to compute shortest detours of a given path. Analogously, Vo et al. [16] used a standard time-dependent shortest path (TDSP) algorithm [15] to compute earliest detours of a journey in a bus network. They evaluated their method on a single network of around 4 000 stops and 8 000 connections, resulting in an average running time of around 1 second to find 5 journeys.

On the other hand, Scano et al. [14] proposed a labelled directed graph modelizing a transportation network where a label is an object composed of the transportation mode (foot, car, bus, *etc.*) and a travel time. This model merges a road and a public transport network together. Then, it is shown how the k shortest path algorithms can be adapted for this model. Specifically, they adapted Yen's and Eppstein's algorithm to work on their model. In both algorithms, a Dijkstra-like algorithm called Dijkstra Regular Language Constraint (DRegLC) [5] is used to answer earliest arrival journeys queries. Moreover, an Iterative Enumeration Algorithm (IEA) is proposed to extract only simple journeys using Eppstein's algorithm. i.e, using Eppstein's k shortest paths algorithm as an iterator and then selecting the simple corresponding journeys (a journey is *simple* if it does not visit a stop more than once).

Experimentally, Scano et al. showed that their IEA is faster than Yen's straightforward adaptation on the transportation network of Toulouse (75 000 nodes, 500 000 road edges and 43 000 public transport edges). On this network, the average running time of Yen's adaptation to find 100 journeys is 250 seconds while it is 0.6 seconds using their refined IEA. However, IEA is not a polynomial-time algorithm, and its memory consumption is too high [14]. In addition, using the labelled directed graph model described in [14] may cause a duplication of the public transit part in practice, i.e, a large number of journeys given by the algorithms proposed in [14] may only differ on the foot-path part while sharing the exact same public transit part. This is undesirable in applications requesting diverse public transit journeys.

Our contribution In this paper, we aim at answering the k earliest arrival journeys queries from a given origin to a given destination in a public transit network. For this purpose, we use the timetable model of public transit networks, i.e, the well-known common model used in [7,9,10]. First, we propose a performant adaptation of Yen's k shortest simple paths algorithm to public transit networks (Yen - Public Transit, Y-PT algorithm). In contrast with [14, 16], we use the Connection Scan Algorithm (CSA) to answer earliest arrival journey queries in our algorithm.

Our main contribution is a novel algorithm, called Postponed Yen's algorithm for Public Transit networks (PY-PT). With the help of a lower bound on the arrival time of a detour journey (a journey that may be one of the k earliest arrival journeys), PY-PT postpones the effective computation of such detour (and so the corresponding earliest arrival journey queries using CSA) with the aim of skipping it. Our experimental results on several train and public transit networks show that the running time of our adaptation of Yen's algorithm is acceptable in practice. Moreover, on the same dataset, the PY-PT algorithm performs 10 to 30 times faster than the Y-PT algorithm on average.

2 Preliminaries

In this section we formalize the inputs and algorithms used in this work. We use almost the same formalization used in [10] for the CSA and as in [1] for Yen's algorithm.

2.1 Graph - definitions and notations

Let $D = (V, A)$ be a digraph with $n = |V|$ vertices and $m = |A|$ arcs, let $N^+(v) = \{w \in V \mid vw \in A\}$ be the set of out-neighbors of vertex $v \in V$, and let $\ell_D : A \to \mathbb{R}^+$ be a length function over the arcs. Sometimes D is pruned.

For every $s, t \in V$, a path from s to t in D is a sequence $P = (s = v_0, v_1, \cdots, v_l = t)$ of vertices with $v_i v_{i+1} \in A$ for all $0 \leq i < l$. An arc uv belongs to a path P ($uv \in P$) if and only if u and v are two consecutive vertices of P, i.e, there is $0 \leq i < l$ such that $u_i = u$ and $u_{i+1} = v$. A

path is *simple* if all of its vertices are distinct, i.e, $v_i \neq v_j$ for all $0 \leq i < j \leq l$. The length of the path P is the sum of the length of its arcs, $\ell_D(P) = \sum_{0 \leq i < l} \ell_D(v_i, v_{i+1})$ The distance $d_D(s,t)$ between two vertices $s, t \in V$ is the length of a shortest s-t path, i.e, a path with the smallest length among all the s-t paths. Given two paths $P = (v_0, \cdots, v_r)$ and $Q = (w_0, \cdots, w_p)$, and an arc $v_r w_0 \in A$, we denote by $P.Q$ the v_0-w_p path resulting from the concatenation of P and Q. That is, $P.Q = (v_0, \cdots, v_r, w_0, \cdots, w_p) = (v_0, \cdots, v_r, Q) = (P, w_0, \cdots, w_p)$.

Given $s, t \in V$, a *set of top-k shortest simple s-t paths* is any set S of s-t simple paths such that $|S| = k$ and $\ell(P) \leq \ell(P')$ for every s-t path $P \in S$ and s-t path $P' \notin S$. The k shortest simple paths problem takes as input a digraph $D = (V, A)$, a length function over the arcs $\ell_D : A \to \mathbb{R}^+$ and a pair of vertices $(s, t) \in V^2$ and asks to find a set of top-k shortest simple s-t paths.

Dijkstra's algorithm finds an s-t shortest path in D with worst-case time complexity in $O(m + n \log n)$.

Let $P = (v_0, v_1, \cdots, v_l)$ be any path in D. Let $0 \leq i < l$, any path $P' = (v_0, \cdots, v_i, v', v'_1, \cdots, v'_r = v_l)$ s.t. $v' \neq v_{i+1}$ is called a *detour* of P at v_i. Note that neither P nor P' are required to be simple. However, if P' is simple, it will be called a *simple detour* of P at v_i. In addition, P' is called a shortest (simple) detour at v_i if and only if P' is a detour with minimum length among all (simple) detours of P at v_i. Finally, the subpath $\pi_i = (v_0, \cdots, v_{i-1})$ of P starting from s and ending at v_{i-1} for $0 \leq i \leq l$ is called i-prefix path of P (the $0-$prefix of any path is an empty path)

2.2 Yen's algorithm

We start by describing Yen's algorithm for finding a set of top-k shortest simple s-t paths in D. For the sake of simplicity, we suppose that D has at least k s-t simple paths.

Yen's algorithm starts by computing a shortest s-t path $P_0 = (s = v_0, v_1, \cdots, v_l = t)$ by applying Dijkstra's algorithm. Note that P_0 is simple since the weights of D are non-negative. Clearly, a second shortest simple s-t path is a shortest simple detour of P_0 at one of its vertices. Yen's algorithm computes, for every vertex v_i in P_0, a shortest simple detour of P_0 at v_i. For this purpose, for $0 \leq i < r$, Yen's algorithm removes the vertices of the i-prefix path $\pi_i = (v_0, \cdots, v_{i-1})$ of P_0 and the arc $v_i v_{i+1}$, then it computes, using Dijkstra's algorithm, a shortest path Q_i from v_i to t. Let $C_i = \pi_i.Q_i$ be the concatenation of π_i and Q_i. First, C_i is simple as Q_i is computed after removing π_i. Second, $v_i v_{i+1} \notin C_i$ as the arc $v_i v_{i+1}$ of P_0 is removed before computing Q_i and constructing C_i. Therefore, C_i is a shortest simple detour of P_0 at i. Note that the index i (called below *deviation-index*) where the path $(v_0, \cdots, v_{i-1}, Q_i)$ deviates from P_0 is kept explicit, i.e, the path is stored with its deviation index. Finally, C_i is added to a set $Candidate$ (initially empty) for every $0 \leq i < l$. Once C_i has been added to $Candidate$ for all $0 \leq i < l$, by remark above, a path with minimum length in $Candidate$ is a second shortest simple s-t path.

Now, let us assume that a set S of top-k' (with $0 < k' < k$) shortest simple s-t paths has been computed and the set $Candidate$ contains a set of simple s-t paths such that there exists a shortest path $Q \in Candidate$ with $S \cup \{Q\}$ a top-$(k'+1)$ set of shortest s-t simple paths.

Let $R = (v_0 = s, \cdots v_j, \cdots, v_r = t)$ be a path in $Candidate$ with minimum length and let j be its deviation index. Similarly to the procedure of finding a second shortest path, Yen's algorithm iterates over the vertices v_i ($j \leq i < r$) of R. At each vertex v_i, a shortest simple detour of R at v_i is added to $Candidate$ (since one of these detours may be a $k' + 1^{th}$ shortest simple s-t path). Let, again, $\pi_i = (v_0, \cdots, v_{i-1})$ be the i-prefix of R. Yen's algorithm removes π_i from D. Then, it removes each arc $v_i w$ such that S contains a path with $(v_0, \cdots, v_i, w)$ as a $i+1$-prefix. Finally, a shortest v_i-t path Q_i is computed, using Dijkstra's algorithm, and the

path $\pi_i.Q_i$ is added to *Candidate* with i as deviation index. This process is repeated until k paths have been found, i.e, when $k' = k$.

Therefore, for each path R that is extracted from *Candidate*, $O(|V(R)|)$ calls of Dijkstra's algorithm are done. This results in a worst-case time-complexity in $O(kn(m + n \log n))$.

2.3 Timetable - definitions and notations

In this section, we describe the data structures used by the CSA, with the same formalization as in [10]. Then we will describe briefly the Connection Scan Algorithm and one of its variant called the profile CSA (PCSA).

Timetable A timetable represents for one specific day the vehicles that exist (train, bus, tram, ferry, ...), when they travel, where they travel and how a passenger can go from one vehicle to another. Formally, a timetable is a quadruple $\mathcal{T} = (S, T, C, F)$ of stops S, trips T, connections C and footpaths F:

- A *stop* is a position outside a vehicle where a passenger can wait. At a stop (and only at a stop) a vehicle can halt and passengers can leave or get on.
- A *trip* is defined by a vehicle going through stops at fixed times. Precisely, a trip is a scheduled vehicle, i.e, a journey done by a unique vehicle from a starting stop to a last stop at a fixed time.
- A *connection* is a vehicle going from one stop to another with no intermediate stops. Formally, a connection c is a quintuple $(c_{dep\_stop}, c_{arr\_stop}, c_{dep\_time}, c_{arr\_time}, c_{trip})$ whose attributes are the departure stop, the arrival stop, the departure time, the arrival time and the trip of c, respectively. A connection must respect two conditions: (1) it cannot be a self loop, i.e, $c_{dep\_stop} \neq c_{arr\_stop}$ and (2) it has a non-zero travel time, i.e, $c_{dep\_time} < c_{arr\_time}$.
- A *footpath* is used to model a transfer, i.e, how to get from one vehicle to another. Formally, a footpath f is a triple $(f_{dep\_stop}, f_{arr\_stop}, f_{dur})$ whose attributes are the departure stop, the arrival stop and the duration of the footpath, respectively. Note that, footpaths are neither trips, nor connections.

Note that, all the connections of a trip form a sequence $c^1, c^2 \dots\ c^\phi$, such that $c^i_{arr\_stop} = c^{i+1}_{dep\_stop}$ and $c^i_{arr\_time} < c^{i+1}_{dep\_time}$ for all $0 \leq i \leq \phi$.

Going from a connection c to a connection c' with $c_{trip} \neq c'_{trip}$ is possible if and only if there is a footpath f^t from c_{arr_stop} to c'_{dep_stop} such that c' is reachable via f^t, i.e, $f^t_{dur} \leq c'_{dep\_time} - c_{arr\_time}$. A loop is introduced on each stop to allow a passenger to get off at a stop and take another trip going through this stop.

Journeys A journey describes how a passenger can travel through a public transit network. It is made of *legs* that are sequences of connections of the same trip. Formally, a journey is a sequence of alternating footpaths and legs $J = (f^0, l^0, f^1, l^1 \dots f^{r-1}, l^r, f^r)$, where $l^i = (c^i_0, \cdots, c^i_{\delta_i})$. That is, a passenger takes the footpath f^0 from $f^0_{dep\_stop}$ to $f^0_{arr\_stop}$, then takes the connection c^1_0, c^1_1, $\cdots$, $c^1_{\delta_1}$, proceeds to take the footpath f^1 from $f^1_{dep\_stop}$ to $f^1_{arr\_stop}$ *etc.* until he reaches $f^r_{arr\_stop}$. A journey must start and end with a footpath, which can be a self loop. In this paper, we sometimes denote a journey as a sequence of footpaths and connection, i.e, $J = (f^0, c^0, c^1, \cdots, c^\alpha, f^1, c^{\alpha+1}, \cdots, f^{r-1}, c^{\gamma+1}, \cdots c^\phi, f^r)$ where $c^0 = c^0_0$, $c^1 = c^0_1, \cdots, c^\phi = c^r_{\delta_r}$.

Given two stops o and d in $\mathcal{S}$, an o-d journey J is a journey $(f^0, c^0, \cdots, c^\phi, f^r)$ such that f^0 starts from o and f^r ends at d. We define the departure time of a journey $dep_t(J)$ as the departure time of its first footpath, formally, $dep_t(J) = c^0_{dep\_time} - f^0_{dur}$. Similarly, the arrival time of a journey $arr_t(J)$ is the arrival time of its last footpath, i.e, $arr_t(J) = c^\phi_{arr\_time} + f^r_{dur}$.

A journey is called simple if it does not visit twice the same stop (except for self loop footpaths). Formally, let $J = (f^0, l^0 = (c^0_0, \cdots, c^0_{\delta_0}), \cdots, f^i, l^i = (c^i_0, \cdots, c^i_{\delta_i}), \cdots, f^j, l^j = (c^j_0, \cdots, c^j_{\delta_j}), \cdots, l^r = (c^r_0, \cdots, c^r_{\delta_r}), f^r)$ be a journey. For all $0 \leq i < j \leq r$, let c_{dep_stop} be the departure stop of c^i_α for $0 \leq \alpha \leq \delta_i$. Similarly, for $0 \leq \beta \leq \delta_j$, let c'_{dep_stop} be the departure stop of c^j_β and c'_{arr_stop} be the arrival stop of c^j_β. We have $c_{dep\_stop} \neq c'_{dep\_stop}$ and $c_{dep\_stop} \neq c'_{arr\_stop}$. [1]

The concatenation of two journeys $J = (f^0, l^0, \cdots, l^r, f^r)$ and $J' = (f'^0 = f^r, l'^0, \cdots, l'^\ell, f'^\ell)$ such that $f^r = f'^0$ and $arr_t(J) \leq dep_t(J')$ is the journey starting by f^0, follows J until f^r, then it follows J' until f'^ℓ. Formally, $J" = (f^0, l^0, \cdots, f^r = f'^0, \cdots, = l'^\ell, f'^\ell)$ (we denote $J" = J.J'$).

Given a journey $J = (f^0, c^0, \cdots c^i, \cdots, c^\phi, f^r)$, a journey $Q = (f'^0, c'^0, \cdots, c'^i, \cdots, c'^w, f'^\ell)$ is called a *detour* of J at i if $f'^0 = f^0, c'^0 = c^0, \cdots, c'^{i-1} = c^{i-1}$ but $c'^i \neq c^i$ and $f'^\ell_{arr\_stop} = f^r_{arr\_stop}$. Moreover, if Q is simple, it is called a *simple detour* of J at i. Similarly, Q is called an earliest arrival (simple) detour of J at i, if $arr_t(Q) \leq arr_t(Q')$ for each (simple) detour Q' of J at i.

Two journeys are equal if and only if all of their attributes are the same.

We denote by $\mathcal{J}^{t_0,t_{max}}_{o,d}$ the set of o-d simple journeys starting from o after t_0 and reaching d before t_{max}, i.e, $\mathcal{J}^{t_0,t_{max}}_{o,d} = \{J$ s.t. J is a simple o-d journey with $dep_t(J) \geq t_0$ and $arr_t(J) \leq t_{max}\}$.

2.4 Connection Scan Algorithm

The CSA answers earliest arrival time journey queries from a given origin o to a given destination d. That is, departing after a given time t_0, how to get from o to d as soon as possible.

Similarly to Dijkstra's algorithm, the CSA will store an earliest arrival time for each stop in an array. A connection is considered *reachable* if a passenger can sit in the public transit vehicle of the connection. However, the main difference between Dijkstra's algorithm and the CSA is the fact that the CSA does not use a priority queue. Instead, the CSA iterates over all the connections sorted by their departure time (the same ordering is used for all queries). The CSA checks whether a connection is reachable or not. If so, it improves the arrival time at the arrival stop of the connection. Once all the connections have been scanned, the earliest arrival time to a stop is the current arrival time stored for the stop. The main advantage of avoiding the use of a priority queue is that, while more connections are scanned, the amount of work per connections is significantly reduced. Therefore, the CSA is significantly faster than Dijkstra's algorithm [10].

2.5 Profile Connection Scan Algorithm

The result of the Profile Connection Scan Algorithm (PCSA) is a mapping between a departure time from a departure stop onto the earliest arrival time at the arrival stop. In other words, the profile problem solves simultaneously the earliest arrival problem for all departure times.

Compared with the CSA, the PCSA iterates on the connections sorted decreasingly by departure time, which leads to the fact that it solves the all-to-one problem. The PCSA constructs journeys from late to early and exploits the fact that an early journey can only have later jour-

[1] We suppose that a leg cannot have a loop, as a user may get off and wait outside the corresponding vehicle.

neys as subjourneys. It has been reported in [10] that the PCSA is one order of magnitude slower than the CSA, which is acceptable considering the fact that it solves the all-to-one problem.

Note that, the PCSA offers, from each stop s to the arrival stop d, a single earliest arrival s-d journey departing after t_0 and reaching d before t_{max}.

Let M be the output of the PCSA, we denote by $M_{o,d}^{t_0,t_{max}}$ the earliest arrival journey, given by M, starting from o and reaching d, departing after t_0 and arriving before t_{max}.

3 Problem definition

In this section, we formalize the k Earliest Arrival Time problem definition.

k Earliest Arrival Time ($kEAT$) problem In this paper, we aim at finding k earliest arrival time ($kEAT$) simple journeys from a given origin to a given destination. Formally, the problem takes as input a timetable $\mathcal{T} = (S, T, C, F)$, origin and destination stops o, d in S, a departure time t_0, a maximum arrival time t_{max} (often $t_{max} = t_0 + 24h$ or $t_{max} = t_0 + 48h$) and an integer k. It asks to find a set $\mathcal{J}^* = \{J_1, J_2, \cdots, J_k\}$ of top-k earliest arrival o-d simple journeys i.e, $J_i \neq J_j$ for $0 \leq i < j \leq k$, and for every J in $\mathcal{J}^*$, $J' \in \mathcal{J}_{o,d}^{t_0,t_{max}}$, $arr_t(J) \leq arr_t(J')$.

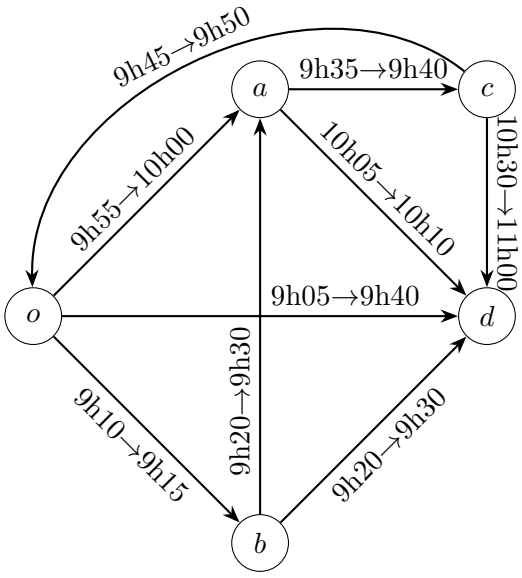


Figure 1: Toy network for k earliest arrival time journeys

Example In the example of Figure 1, we look for the four earliest arrival time journeys from o to d departing after 9h00:

The earliest arrival journey arrives at d at 9h30, starts with o and reaches d via b, the passenger arrives at b at 9h15 and waits 5 minutes before boarding the connection going from b to d, $J_0 = (o, d, b)$. The second journey arrives at 9h40 and goes directly from o to d, $J_1 = (o, d)$. The third journey arrives at 10h10 and goes from o to b then a then d, the passenger arrives at b at 9h15, waits 10 minutes then boards the connection going from b to a, arrives at 9h30 and waits 35 minutes before boarding the connection going from a to d, $J_2 = (o, b, a, d)$. The fourth journey arrives at 10h10 and goes from o to a then d, $J_3 = (o, a, d)$.

Note that the journey $J_{ns} = (o, b, a, c, o, a, d)$ arriving at 10h10 is not a part of the solution as it is not simple (it visits the station o twice).

Indeed, there are other o-d journeys in this example, however they all have an arrival time greater than 10h10, that is $\{J_0, J_1, J_2, J_3\}$ are the four earliest arrival simple o-d journeys.

Each edge in the graph belongs to a specific trip, meaning that between each step in the examples there is a self loop footpath.

4 Public Transit Yen's algorithm (Y-PT)

In this section, we describe our adaption of Yen's algorithm on public transit networks, called Y-PT algorithm. As described before, Y-PT algorithm solves the $kEAT$ problem. So, it takes as input a timetable $\mathcal{T} = (S, T, C, F)$, origin and destination stops o, d in S, a departure time t_0, a maximum arrival time t_{max} $(= t_0 + 48h)$ and an integer k, and returns a set $Output = \{J_1, J_2, \cdots, J_k\}$ of top-k earliest arrival o-d simple journeys in $\mathcal{T}$.

Roughly, Y-PT algorithm starts by computing a first earliest arrival journey, iterates over its connections in order to compute its earliest arrival simple detours and adds their minimum (the detour with minimum arrival time) to the output. Then, Y-PT algorithm repeats this process until k journeys are added to the output.

Now, let us give a precise and formal description of Y-PT algorithm whose pseudocode is presented in Algorithm 1. Analogously to Yen's algorithm, Y-PT starts by computing an earliest arrival journey J_0 and adding it (with 0 as deviation index) to a set of candidate journeys called $Candidates$. The journeys of the set $Candidates$ are non-decreasingly sorted by their arrival time. Also, the algorithm initializes the output set $Output$ as an empty set. After this initialization phase, the algorithm extracts a minimum element from the set $Candidates$, i.e, a journey $J = (f^0, c^0, \cdots, c^\phi, f^r)$ with minimum arrival time among those in $Candidates$ and adds it to $Output$. Let $C_J = (c^0, c^1, \cdots, c^\phi)$ be the sequence of connections of J. The algorithm iterates over the connections in C_J starting from the deviation index of J. Precisely, let j be the deviation index of J, for each connection $c^i = (c^i_{dep\_stop}, c^i_{arr\_stop}, c^i_{dep\_time}, c^i_{arr\_time}, c^i_{trip})$ for $j \leq i \leq \phi$, the algorithm removes the prefix stations, i.e, each station visited by one of the connections $c^0, \cdots, c^{i-1}$, (equivalent to the prefix path of Yen's) from $\mathcal{T}$. This is done to ensure that the candidate journey is simple.

Moreover, in order to avoid duplications of journeys, for each journey J in $Output$ starting with the connections $c^0, c^1, \cdots, c^{i-1}, c'$, the connection c' is removed from $\mathcal{T}$. Then, using the CSA, the Y-PT algorithm computes an earliest arrival journey $Q = (f^0_Q, c^0_Q, \cdots, c^\omega_Q, f^\ell_Q)$ from $c^{i-1}_{arr\_stop}$ to d with $c^{i-1}_{arr\_time}$ as departure time [2]. Let J_{new} be the concatenation of the prefix of J and Q, i.e, $J_{new} = (f^0, c^0, \cdots, c^{i-1}, f^0_Q, c^0_Q, \cdots, c^\omega_Q, f^\ell_Q)$. The journey J_{new} is added to $Candidates$ with i as deviation index.

Y-PT algorithm repeats this process until k journeys are added to $Output$.

5 Public Transit Postponed Yen's algorithm (PY-PT)

Here we explain Postponed Yen algorithm for public transit (PY-PT algorithm) whose pseudocode is presented in Algorithm 2. This algorithm is inspired from the Postponed Node Classification algorithm (PNC) for the $kSSP$ described in [2].

[2] If the element right before c^i is a footpath, i.e, $J = (f^0, \cdots, f^\lambda, c^i, \cdots, f^r)$. It is possible to have journeys with two consecutive footpaths. In order to avoid such scenario, the CSA call is forced to compute a journey starting with a self loop footpath.

Algorithm 1 Public Transit - Yen's algorithm (PT-Y)

Input A timetable $\mathcal{T} = (S, T, C, F)$, an origin and a destination stops (o and d), departure and maximum arrival time t_{dep}, t_{max} and an integer k
Output a set of top-k earliest arrival journeys from o to d departing after t_{dep}
$J_0 \leftarrow CSA(\mathcal{T}, o, d, t_{dep}, t_{max})$
$Candidates \leftarrow \{(J_0, 0)\}$
$Output \leftarrow \emptyset$
while $\mid Output \mid < k$ and $Candidate \neq \emptyset$ **do**
    $\varepsilon = (J, j) \leftarrow extractmin(Candidates)$
    let $J = (f^0, c^0, \cdots, c^\phi, f^r), j)$
    add J to $Output$
    for each connection c^i with $j \leq i \leq \phi$ in J **do**
        $c_{arr_stop} \leftarrow$the arrival stop of c^{i-1}
        $c_{arr_time} \leftarrow$the arrival time of c^{i-1}
        $\pi = (f^0, c^0, \cdots, c^{i-1})$
        $S_\pi \leftarrow$ the set of stations visited by one of the connections $(c^0, \cdots, c^{i-1})$
        $C_{dev} \leftarrow \{c'$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, c')\}$
        $\mathcal{T}' = (S \setminus S_\pi, T, C \setminus C_{dev}, F)$
        $Q \leftarrow CSA(\mathcal{T}', c_{arr\_stop}, d, c_{arr\_time}, t_{max})$
        $J_{new} \leftarrow \pi.Q$
        add (J_{new}, i) to $Candidates$
Return $Output$

PY-PT algorithm has the same input as Y-PT algorithm, and it also returns a set of top-k earliest arrival simple journeys from the origin to the destination in a timetable. However, the journeys given by Y-PT are not necessarily the same as those given by PY-PT, i.e, the order of extraction of journeys is not necessarily the same. This may occur in scenarios where several journeys from the origin to the destination have the same arrival time.

The main drawback of Y-PT algorithm is its excessive number of calls of the CSA. Here, with the help of lower bounds on the arrival time of simple detours, we propose to postpone these calls in order to avoid some of them. We show that this can be done while preserving the correctness of the algorithm. In contrast with Y-PT algorithm where all journeys in the set $Candidates$ are simple, the PY-PT algorithm may add non-simple journeys to the set $Candidates$. As shown below, this corresponds to detours whose effective computation (and so their corresponding CSA calls) are postponed.

Let us now describe PY-PT algorithm in details.

For a query from the origin o to the destination d starting at time t_0, the PY-PT algorithm first uses the Profile CSA (PCSA). Let M be the mapping output by PCSA. The mapping M associates to each station $s \in S$ and each departure time $t \geq t_0$ the earliest arrival s-d journey, providing it is possible to reach d from s before t_{max} when starting at t (we let $t_{max} = t_0 + 48h$ in our experiments).

Similarly to Y-PT algorithm, PY-PT algorithm starts by adding an earliest arrival time journey J_0 to a set of candidate journeys called $Candidates$. An element ε in $Candidates$ has three attributes, the journey J, its deviation index i and a boolean flag ζ indicating whether J is simple or not. So, the element $\varepsilon_0 = (J_0, 0, 1)$ is added to $Candidates$. In contrast with Y-PT algorithm where a CSA call is consumed to compute J_0, PY-PT algorithm extract J_0 from the already computed mapping M. Precisely, $J_0 = M_{o,d}^{t_0, t_{max}}$. Then, also like Y-PT algorithm, the $Output$ set is initialized with an empty set. After these initializations steps, the algorithms

starts by extracting an earliest arrival journey (J, j, ζ) among those in *Candidates*. Suppose $J = (f^0, c^0, \cdots, c^\alpha, f^1, c^{\alpha+1}, \cdots, c^\beta, f^2, \cdots, c^{\gamma+1}, \cdots, c^\phi, f^r)$. Two cases are distinguished:

- **if $\zeta = 1$ (J is simple)**: J is added to the *Output*, then all the earliest arrival detours of J are added to *Candidates*. This is done as follows, let $C_J = (c^0, c^1, \cdots, c^\phi)$ be the sequence of connections of J, at each connection c^i (for $j \leq i < \phi$) in C_J, an earliest arrival detour J_{new} of J at i is extracted. This is done with the help of M as described below.

  The journey J_{new} may not be simple (also described below). However, J_{new} will be added to the set *Candidate* with i as deviation index and $\zeta = 1$ if Q is simple (and $\zeta = 0$ otherwise).

- **if $\zeta = 0$ (J is not simple)**: Then J is "repaired", i.e., it is replaced (if possible) by its corresponding earliest arrival simple journey. For this purpose, the algorithm applies almost the same routine as Y-PT algorithm. Precisely, let $c^j = (c^j_{dep\_stop}, c^j_{arr\_stop}, c^j_{dep\_time}, c^j_{arr\_time}, c^j_{trip})$ be the connection at the deviation index, the algorithm removes the prefix stations, i.e, each station visited by one of the connections $c^0, \cdots, c^{j-1}$, from $\mathcal{T}$. Also, for each journey J' in *Output* starting with the connections $c^0, c^1, \cdots, c^{j-1}, c'$, the connection c' is removed from $\mathcal{T}$. Then, using the CSA, PY-PT algorithm computes an earliest arrival journey $Q = (f^0_Q, c^0_Q, \cdots, f^{\ell-1}_Q, l^\phi_Q, f^\ell_Q)$ from $c^{j-1}_{arr\_stop}$ to d with $c^{j-1}_{arr\_time}$ as departure time. Let J_{new} be the concatenation of the prefix of J and Q, i.e, $J_{new} = (f^0, c^0, \cdots, c^{j-1}, f^0_Q, c^0_Q, \cdots, f^\ell_Q)$. The journey J_{new} is added to the *Candidates* with j as deviation index and with $\zeta = 1$ (as J_{new} is simple).

The PY-PT algorithm repeats this process until k journeys are added to *Output*.

Now, let us explain how the journey J_{new} is computed (in the case where $\zeta = 1$). The pseudocode of this procedure is described in Algorithm 3. Let $c^i = (c^i_{dep\_stop}, c^i_{arr\_stop}, c^i_{dep\_time}, c^i_{arr\_time}, c^i_{trip})$ be the i^{th} connection of C_J (for $j \leq i < \phi$), the following procedure is applied:

- First, the algorithm scans the connections starting with $c^i_{arr\_stop}$ after $c^i_{arr\_time}$ leading to new journeys, i.e, different from those in *Output*. Precisely, let $C_{dev} = \{c_{old} \in C$ s.t. there is a journey in *Output* starting with the connections $c^0, \cdots, c^{i-1}, c_{old}\}$, let $C^N = \{c \in C$ s.t. $c_{dep\_stop} = c^i_{dep\_stop}$, $c_{dep\_time} \geq c^i_{dep\_time}$ and $c \notin C_{dev}\}$ be the set of new deviating connections. The algorithm scans the connections of C^N. Let c^{LB} be a connection of C^N leading to an earliest arrival journey from $c^i_{dep\_stop}$ to d using M. Formally, for each c in C^N, let J_c be the journey via c following M, i.e, let $J_c = c.M^{c_{arr\_time}, t_{max}}_{c_{arr\_stop}, d}$, then c^{LB} is a connection in C^N s.t. $arr_t(J_{c^{LB}}) \leq arr_t(J_c)$ for each c in C^N[3].

- Second, the algorithm scans the footpaths starting with $c^i_{dep\_stop}$ leading to new journeys, i.e, different from those in *Output*. Again, let $F_{dev} = \{f_{old}$ s.t. there is a journey in *Output* starting with the connections $c^0, \cdots, c^{i-1}$ followed by $f_{old}\}$, let $F^N = \{f \in F$ s.t. $f_{dep\_stop} = c^i_{dep\_stop}$ and $f \notin F_{dev}\}$ be the set of the new deviating footpaths and let f^{LB} be a footpath of F^N leading to an earliest arrival journey from $c^i_{dep\_stop}$ to d using M. Precisely, for each f in F^N, let J_f be the journey via f following M, i.e, $J_f = f.M^{c^i_{dep\_time} + f_{dur}, t_{max}}_{farr\_stop, d}$, then f^{LB} is a footpath in F^N s.t. $arr_t(J_{f^{LB}}) \leq arr_t(J_f)$ for each f in F^N.

[3]If the element right before c^i is a footpath, i.e, $J = (f^0, \cdots, f^\lambda, c^i, \cdots, f^r)$. It is possible to have journeys with two consecutive footpaths. In order to avoid such scenario, the footpaths starting with $c^i_{arr\_stop}$ will not be scanned.

Now let Q_{min} be the journey with minimum arrival time among $J_{c^{LB}}$ and $J_{f^{LB}}$ and let J_{min} be the journey formed by the concatenation of the prefix journey of J and Q_{min}, i.e, $J_{min} = (f^0, c^0, \cdots, c^{i-1}, Q_{min})$. Note that, J_{min} may not be simple as the sub-journey extracted from M may revisit a station of one of the prefix connections. For instance, a station that is visited by c^0 or $c^1, \cdots$, or c^{i-1} may be visited again by $J_{c^{LB}}$ (or by J_f^{LB})[4].

To conclude, in contrast with Y-PT algorithm where an earliest arrival simple detour is computed at each index of an extracted journey using the CSA, PY-PT algorithm consider an earliest arrival detour (not necessarily simple) given by the already computed PCSA at each index, and two cases are distinguished: If the earliest arrival detour is simple, then a CSA call is saved and a shortest simple detour is added to *Candidates*. If not, i.e, the earliest arrival detour is not simple, PY-PT algorithm inserts this non-simple detour to the set of *Candidates* with a flag indicating that it is not simple. Recall that journeys in *Candidates* are non-decreasingly stored by their arrival time. So, only when this non-simple detour is extracted from *Candidates*, its simple version will be computed using the CSA. In other words, the actual computation of such simple detour is "postponed". Such postponement may end up saving some CSA calls, typically when k earliest arrival journey are added to *Output* while none-simple journeys, whose actual computation is postponed, are still in *Candidates*, i.e, their whole "repair" procedure is skipped.

Note that, despite these postponements, the order of extraction of simple journeys from *Candidates* remains valid. This is because a journey J in *Candidate* is either inserted with its real arrival time (the case where J is simple) or with a lower bound on its arrival time (the case where J is non-simple, by Claim 1).

Claim 1. *Let $J = (f^0, c^0, f^1 \cdots, c^\phi, f^r)$ be an o-d journey with J_{ns} an earliest arrival detour of J at i and J_s with an earliest simple arrival detour of J at i (where $0 \leq i \leq \phi$). Then, $arr_t(J_{ns}) \leq arr_t(J_s)$*

Proof. The proof follows from the fact that an earliest arrival detour of J at i arrives earlier than any detour of J at i. In particular, it arrives earlier than any earliest arrival simple detour of J at i. □

6 Experimental evaluation

In this section we describe our experimental evaluation. First, we start by describing our implementation and settings (Section 6.1), then we discuss our experimental results on train and public transit networks (Section 6.2).

6.1 Experimental settings

Here we describe the details of the implementation and the setting used in our experiments.

We have implemented Y-PT and PY-PT algorithms in Java and our code is publicly available at [4]. Note that in our implementations the parameter k is not part of the input, this enables the use of these methods as iterators, able to return a next earliest arrival itinerary as long

[4]When scanning the connections starting with $c^i_{arr\_stop}$ after $c^i_{arr\_time}$, the journey $M^{c^i_{arr\_time}, t_{max}}_{c^i_{arr\_stop}, d}$ can start either with a self loop footpath or a footpath. On the other hand, when scanning footpaths starting with $c^i_{dep\_stop}$ the journey $M^{c^i_{dep\_time}+f_{dur}, t_{max}}_{farr\_stop, d}$ cannot start with anything other than a self loop footpath, to do so the PCSA stores journeys in two separate data structures, one for journeys starting with a self loop footpath and one for the other journeys.

Algorithm 2 Public Transit - Postponed Yen's algorithm (PY-PT)

Input A timetable $\mathcal{T}$, an origin and a destination stops (o and d), departure and maximum arrival time (t_{dep} and t_{max}), and an integer k
Output a set of top-k earliest arrival simple journeys from o to d departing after t_{dep}
$M \leftarrow PCSA(\mathcal{T}, o, d, t_{dep}, t_{max})$
$J_0 \leftarrow M_{o,d}^{t_{dep},t_{max}}$
$Candidate \leftarrow \{(J_0, 0, \zeta = 1)\}$
$Output \leftarrow \emptyset$
while $Candidate \neq \emptyset$ and $|Output| < k$ **do**
    $\varepsilon = (J, j, \zeta) \leftarrow extractmin(Candidates)$
    Let $J = (f^0, c^0, \cdots, c^\phi, f^r)$
    if $\zeta = 1$ (J is simple) **then**
        add J to $Output$
        for each vertex c^i in $(c_j, \cdots, c^\phi)$ **do**
            $J_{new} \leftarrow$ EarliestArrivalDetour(J, i, M)
            $\zeta' \leftarrow 0$
            if J_{new} is simple **then**
                $\zeta' \leftarrow 1$
            add (J_{new}, i, ζ') to $Candidate$
    else
        $S_\pi \leftarrow$ the set of stations visited by one of the connections $(c^0, \cdots, c^{j-1})$
        $C_{dev} \leftarrow \{c$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, c)\}$
        $\mathcal{T}' = (S \setminus S_\pi, T, C \setminus C_{dev}, F)$
        $Q \leftarrow CSA(\mathcal{T}', c_{arr\_stop}, d, c_{arr\_time}, t_{max})$
        if Q exists **then**
            $J_{new} \leftarrow (f^0, c^0, \cdots, c^j, Q)$
            add $(J_{new}, j, \zeta = 1)$ to $Candidates$
return $Output$

as one exists. Despite the fact that some additional optimizations could be added to the implementation if k is a part of the input.

Networks setting We have evaluated the performances of our algorithms on two train networks (Germany and Switzerland) and three public transit networks (Paris, Berlin and Stockholm). The characteristics of these networks are presented in Table 1. This dataset is publicly available via a GTFS feed (`https://transitfeeds.com/`), we downloaded this dataset in October 2019.

The public transit networks are more dense than the train networks, i.e. the connections to stops ratio is smaller on train networks than public transit networks. This can be easily explained because the train networks can only use trains whereas the public transit networks can use buses, trains, ferries and many other means of transportation. Therefore, we will show the performances of our algorithms on those two types of networks.

In our experiments, we have randomly chosen 1000 queries (source-destination pairs of stops) for each public transit network, and we have run each algorithm for each of these pairs for k going from 2 to 100.

We have considered the execution time and the number of CSA calls. Note that the number of CSA calls is an indication of the running time which is independent of the implementation

Algorithm 3 EarliestArrivalDetour(J, i, M)

$c^i \leftarrow$ the i^{th} connection of J
$c_{arr\_stop} \leftarrow$ the arrival stop of c^{i-1}
$c_{arr\_time} \leftarrow$ the arrival time of c^{i-1}
$C_{dev} \leftarrow \{c' \text{ s.t. there is a journey } J' \text{ in } Output \text{ starting with } (c^0, \cdots, c^{i-1}, c')\}$
$C^N = \{c' \in C \text{ s.t. } c'_{dep\_stop} = c_{arr\_stop}, c'_{dep\_time} \geq c_{arr\_time} \text{ and } c' \notin C_{dev}\}$
$c^{LB} \leftarrow$ a connection in C^N leading to a minimum arrival time from c_{arr_stop} to d after c_{arr_time} following M
$F_{dev} \leftarrow \{f \text{ s.t. there is a journey } J' \text{ in } Output \text{ starting with } (c^0, \cdots, c^{i-1}, f)\}$
$F^N = \{f \in F \text{ s.t. } f_{dep\_stop} = c_{arr\_stop} \text{ and } f \notin F_{dev}\}$
$f^{LB} \leftarrow$ a footpath in F^N leading to a minimum arrival time from c_{arr_stop} to d following M
$J_{c^{LB}} \leftarrow c^{LB}.M^{c^{LB}_{arr\_time}, t_{max}}_{c^{LB}_{arr\_stop}, d}$
$J_{f^{LB}} \leftarrow f^{LB}.M^{c_{arr\_time}+f^{LB}_{dur}, t_{max}}_{f^{LB}_{arr\_stop}, d}$
$J_{min} \leftarrow$ the earliest arrival journey among $J_{c^{LB}}$ and $J_{f^{LB}}$
$\pi = (f^0, c^0, \cdots, c^{i-1})$
$J_{new} \leftarrow \pi.J_{min}$
return J_{new}

and the architecture of the machine.

All reported computations have been performed on computers equipped with an Intel(R) Core(TM) i7-1185G7 at 3.00GHz and 32 GB of RAM.

Network	Stops	Connections	Lines	Trips	Footpaths
Germany	74 398	3 601 420	3 599	168 024	599 284
Switzerland	29 844	2 599 675	5 645	248 826	27 202
Paris	44 534	3 209 401	1 864	150 963	502 291
Berlin	28 651	1 379 755	1 296	63 569	62 456
Stockholm	14 258	703 326	664	34 799	22 138

Table 1: Characteristics of the PT networks: number of stops, connections, lines, trips and footpaths.

6.2 Experimental results

In this section, we describe and analyse our experimental results on public transit networks.

We have measured the average and the median of the algorithms' running time in the considered networks. The data (the running time and the number of CSA calls) in Tables 2 and 3 and figure 3 corresponds to the biggest experienced value of k ($k = 100$). While the data in Figure 2 corresponds to their evolution with respect to the values of k.

The average and median running times reported in Table 2 show that the PY-PT algorithm is significantly faster than the Y-PT algorithm for every considered network (the average speed up of the running time is bigger than 10 for $k = 100$). Moreover, a refined comparison on Germany and Paris networks (Figure 3) show that PY-PT is faster than Y-PT for almost all queries. In addition, Figures 2a and 2b shows that this speed up remains considerable even for small values of k (even for $k = 2$) for Stockholm and Switzerland networks. This means that the time consumed for the PCSA computation routine is compensated by the extraction of

		Germany	Switzerland	Paris	Berlin	Stockholm
Y-PT	avg	94.6	42.0	66	22.7	7.2
	med	47.3	30.6	25.1	14	3.5
PY-PT	avg	3.6	1.9	5.4	0.8	0.2
	med	1.7	1.4	3.8	0.5	0.1

Table 2: Running time (s) of the algorithms on PT networks, ($k = 100$)

		Germany	Switzerland	Paris	Berlin	Stockholm
Y-PT	avg	2132	2158	1355	1788	2072
	med	1729	1749	1262	1604	1510
PY-PT	avg	32	77	39	7.6	8.3
	med	12	56	26	7	2

Table 3: Number of CSA calls using each of the algorithms on PT networks, ($k = 100$)

simple detours, even for $k = 2$. In addition, very similar results were obtained on the remaining networks. Based on these remarks, we conclude that, in practice, PY-PT is faster than Y-PT for almost every scenario (the value of k, the query specifications and the network structure).

Furthermore, on Stockholm and Switzerland networks, Table 3 and Figures 2c and 2d show that the number of CSA calls is significantly reduced using PY-PT. This ensures that a similar speed up is guaranteed for any experimental settings [12].

As the obtained results are similar, we only displayed data obtained from experiments on selected networks (Stockolm and Switzerland for Figure 2, Paris and Germany for Figure 3). However, the results/plots corresponding to the remaining networks are very similar.

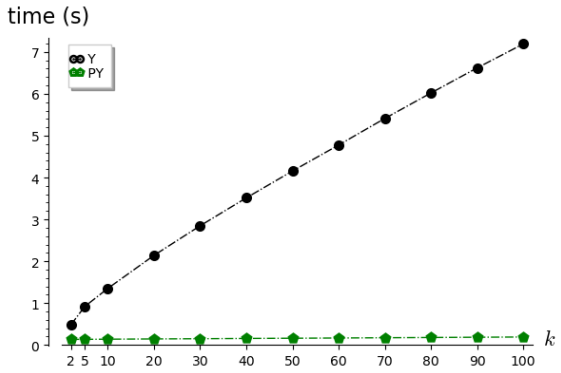
To conclude, on average, PY-PT algorithm is more than 10 times faster than Y-PT algorithm, it is also faster than Y-PT for almost every scenario.

7 Conclusion

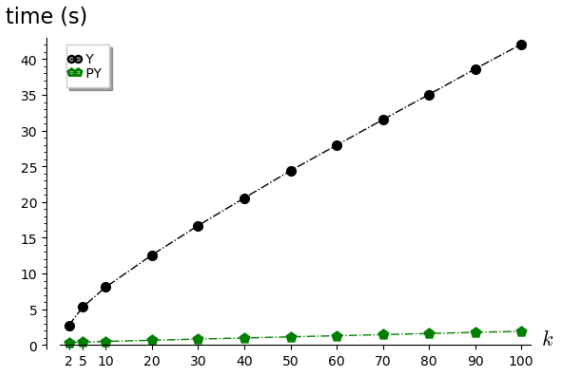
In this paper, we have shed lights on a new style of journey planning in public transit networks, offering a vast set of interesting solutions. This is done by adapting the k shortest simple paths problem to the public transit network context. We proposed a straightforward adaptation of Yen's algorithm and a more refined version answering the proposed problem in a reasonable running time.

Interesting questions are asked about designing algorithms answering k earliest arrival journeys query faster. Whether by improving / proposing faster methods than PY-PT algorithm, or even with the help of a preprocessing routine. For instance, a more specific question is whether one can use journey planning algorithms like Transfer Patterns algorithm [6] to answer k earliest arrival journeys queries ?

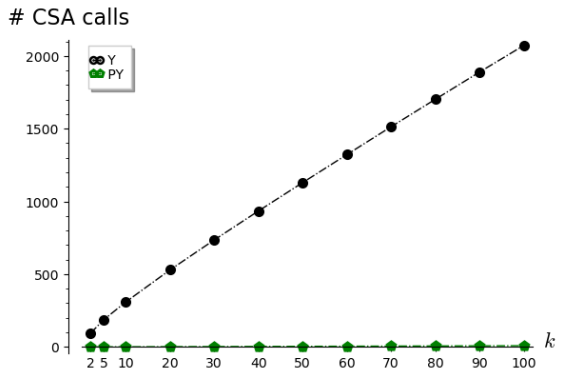
In addition, the approach proposed in this paper does not guarantee any dissimilarity of the proposed journeys, i.e, a large part of output journeys may overlap in some scenarios. So, an interesting question is the study of finding journeys that are "dissimilar" in public transit networks, as studied for shortest dissimilar paths finding in a graph [3, 8].



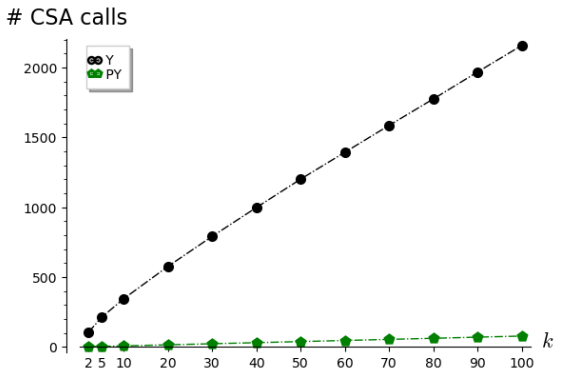
(a) Average running time on Stockholm



(b) Average running time on Switzerland

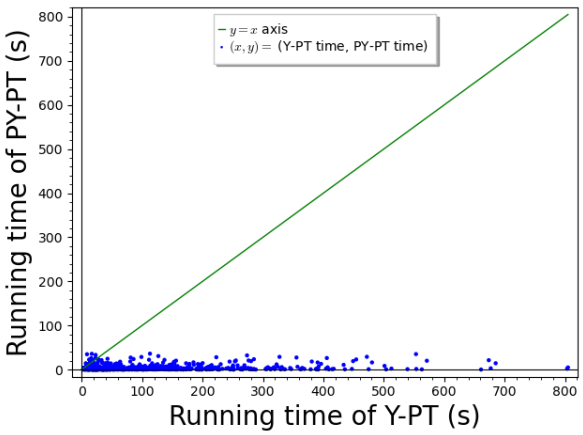


(c) Average number of CSA calls on Stockholm

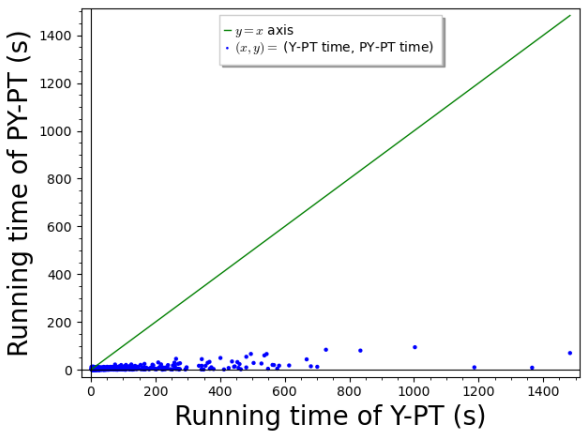


(d) Average number of CSA calls on Switzerland

Figure 2: The running time of the $kEAT$ algorithms on Switzerland train network and Stockholm public transit network with respect to the values of k



(a) Running time of Y-PT and PY-PT on Germany



(b) Running time of Y-PT and PY-PT on Paris

Figure 3: Comparison of the running time of Y-PT and PY-PT on a train network and a public transit network

References

[1] A. Al Zoobi, D. Coudert, and N. Nisse. Space and time trade-off for the k shortest simple paths problem. In 18th International Symposium on Experimental Algorithms (SEA), volume 160, page 13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[2] A. Al Zoobi, D. Coudert, and N. Nisse. Finding the k Shortest Simple Paths: Time and Space trade-offs. Research report, Inria ; I3S, Université Côte d'Azur, Apr. 2021.

[3] A. Al Zoobi, D. Coudert, and N. Nisse. On the complexity of finding k shortest dissimilar paths in a graph. Research report, Inria ; CNRS ; I3S ; UCA, 2021.

[4] A. Al Zoobi and A. Finkelstein. PT-KSSP Github repository, 2021. `https://github.com/fink-arthur/PT-KSSP`.

[5] C. Barrett, K. Bisset, M. Holzer, G. Konjevod, M. Marathe, and D. Wagner. Engineering label-constrained shortest-path algorithms. In International conference on algorithmic applications in management, pages 27–37. Springer, 2008.

[6] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger. Fast routing in very large public transportation networks using transfer patterns. In Algorithms - ESA 2010, 18th Annual European Symposium. Proceedings, Part I, pages 290–301, 2010.

[7] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In Algorithm engineering, pages 19–80. Springer, 2016.

[8] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser. Exact and approximate algorithms for finding k-shortest paths with limited overlap. In Proceedings of the 20th International Conference on Extending Database Technology, (EDBT), pages 414–425, Venice, Italy, Mar. 2017. OpenProceedings.org.

[9] D. Delling, T. Pajor, and R. F. Werneck. Round-based public transit routing. Transportation Science, 49(3):591–604, 2015.

[10] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. Connection scan algorithm. ACM Journal of Experimental Algorithmics (JEA), 23:1–56, 2018.

[11] D. Eppstein. Finding the k shortest paths. SIAM Journal on Computing, 28(2):652–673, 1998.

[12] D. S. Johnson. A theoretician's guide to the experimental analysis of algorithms. Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges, 59:215–250, 2002.

[13] D. Kurz and P. Mutzel. A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph. In S. Hong, editor, 27th International Symposium on Algorithms and Computation, ISAAC 2016, December 12-14, 2016, Sydney, Australia, volume 64 of LIPIcs, pages 49:1–49:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[14] G. Scano, M.-J. Huguet, and S. U. Ngueveu. Adaptations of k-shortest path algorithms for transportation networks. In 2015 International Conference on Industrial Engineering and Systems Management (IESM), pages 663–669. IEEE, 2015.

[15] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. ACM Journal of Experimental Algorithmics (JEA), 5:12–es, 2000.

[16] K. D. Vo, T. V. Pham, H. T. Nguyen, N. Nguyen, and T. Van Hoai. Finding alternative paths in city bus networks. In 2015 International Conference on Computer, Control, Informatics and its Applications (IC3INA), pages 34–39. IEEE, 2015.

[17] J. Y. Yen. Finding the k shortest loopless paths in a network. Management Science, 17(11):712–716, 1971.