

Compacting Massive Public Transport Data [★]

Benjamín Letelier²[0009–0005–5997–0225], Nieves R.
Brisaboa¹[0000–0001–8025–3048], Pablo Gutiérrez-Asorey¹[0000–0002–6878–6041],
José R. Paramá¹[0000–0002–8727–0980], and Tirso V.
Rodeiro¹[0000–0003–2373–0746]

¹ Universidade da Coruña, CITIC, Campus Elviña, 15071. A Coruña, Spain

² Instituto de Informática, Universidad Austral de Chile, Valdivia, Chile

Abstract. In this work, we present a compact method for storing and indexing users’ trips across transport networks. This research is part of a larger project focused on providing transportation managers with the tools to analyze the need for improvements in public transportation networks. Specifically, we focus on addressing the problem of grouping the massive amount of data from the records of traveller cards as coherent trips that describe the trajectory of users from one origin stop to a destination using the transport network, and the efficient storage and querying of those trips. We propose two alternative methods capable of achieving a space reduction between 60 to 80% with respect to storing the raw trip data. In addition, our proposed methods are auto-indexed, allowing fast querying of the trip data to answer relevant questions for public transport administrators, such as how many trips have been made from an origin to a destination or how many trips made a transfer in a certain station.

Keywords: Compression · Public Transport · Trip analysis.

1 Introduction

The widespread use of public transport cards has opened the door to a wide range of possibilities for analysing people’s movements through cities. Each time a passenger boards any means of public transport, his card number, time, and transport mean get registered. This enables not only the analysis of individuals (where, when, etc.) but also aggregated analysis to study the movement patterns in cities (rush hours, flows between zones, etc.). However, the storage and

[★] This work was partially supported by the CITIC research center funded by Xunta de Galicia, FEDER Galicia 2014-2020 80%, SXU 20% [CSI: ED431G 2019/01]; MCIN/ AEI/10.13039/501100011033 ([EXTRA-Compact: PID2020-114635RB-I00]; “NextGenerationEU”/PRTR [SIGTRANS: PDC2021-120917-C21], [PLAGEMIS: TED2021-129245B-C21]; EU/ERDF A way of making Europe [OASSIS-UDC: PID2021-122554OB-C3]); by GAIN/Xunta de Galicia [GRC: ED431C 2021/53]; by UE FEDER [CO3: IN852D 2021/3]; by Xunta de Galicia [ED481A/2021-183], and by the Fondecyt grant #11221029 of Universidad Austral de Chile.

management of these data are not trivial due to their spatio-temporal nature and their size.

Usually, static information about transport networks and their topology is publicly shared to be the basis of geographic information systems used for public transport webs and applications. There are several queries of interest in this context that could be solved using directly the network information (e.g. *What route should be followed to travel from point A to point B using the transport network?*). This is not the target of this work, our goal is the representation of user behaviour, i.e. movement patterns, preferred routes, etc.

Our current line of research involves the creation of a tool for efficiently storing and analyzing the vast amount of data related to the use of public transport networks. While this research faces three main challenges pertaining to the grouping of traveller cards transaction records into trips, the efficient storage of trips on a novel Compact Data Structure (CDS) [9], and the design and implementation of user interfaces for querying the stored data based on Geographical Information System (GIS) technologies, this work will focus primarily on introducing our compact method for the storage of the trips.

2 Background

Compact Data Structures have gained a lot of notoriety as an efficient strategy for Big Data scenarios, and in particular, we have already done some work at our research group regarding trajectories and mobility data on transport networks [6]. Our proposal follows those footsteps, using a variety of well-known data structures as the building blocks for our approach.

First, it should be noted that, given that CDS operate at bit level, bitvectors (sequences of bits) are one fundamental element in most of these data structures. Bitvectors can be processed in a highly efficient manner due to two well-known basic operations: *rank* and *select* [8]. For this work, we used *rank* and *select* implementations included in [7], both with a temporal cost of $O(1)$, where the *rank* operation uses $0.25n$ additional bits (with n being the size of the bitvector), and *select* at most $0.2n$ bits of additional space.

We also used an indexing structure known as K^n tree. The K^2 structure was first conceived as a compact web graph representation [5]. However, it turns out to be a more efficient version of a region quadtree [10], and thus, it can be used used as a spatial index [2–4].

It represents a binary matrix of size $n \times n$ using a conceptual K^2 -ary tree for a given K . The root node represents the entire matrix. Then, the matrix is divided into K^2 submatrices of size $n/K \times n/K$. The order for those submatrices is established as left-to-right and top-to-bottom, adding a child node to the root of the tree for each submatrix. Each node is labelled with a bit. If a submatrix contains at least a 1-bit, the corresponding node stores a 1-bit, otherwise, the node stores a 0-bit. For the submatrices with at least one 1-bit, the procedure continues recursively until reaching a submatrix with only 0-bits or the individual cells of the original matrix. The K^2 -tree divides that bitmap in two: L , which

is formed by the bits corresponding to the last level of the tree, and T , which contains the rest.

The K^2 -tree can efficiently answer several queries, like retrieving the value of a single cell, row/column queries, or window queries. Queries are solved by navigating T and L with *rank* and *select* operations.

Note that the K^2 structure can be generalized to any n . In this work, we used a variant for indexing three-dimensional binary matrices called the K^3 -tree [2].

One last concept that is necessary in order to fully understand the following sections is the *prefix sums*. Given an array $S[1..n]$ of non-negative numbers, the prefix sum of S is defined as a new array $S_{PS}[1..n+1]$ that stores the sum of prefixes of the input sequence in the form of $S_{PS}[1] = 0$ and $S_{PS}[j] = \sum_{i=1}^{j-1} S[i]$, with $2 \leq j \leq n+1$. This representation allows for the recovery of the sum in any range on S by using $\sum_{i'=i}^j S[i'] = (S_{PS}[j+1] - S_{PS}[i])$, with $1 \leq i \leq j \leq n$, in $O(1)$. Note that, in the case of $i = j$, the value returned is $S[i]$. One apparent problem with this representation is that S_{PS} requires much more space than S . Therefore, this work used the sampling technique described by Navarro [9] to space optimize prefix sums.

3 Previous concepts on transport networks

This section addresses the basic vocabulary related to transport networks needed to understand the following sections and shows some of the difficulties associated with constructing *trips* from traveller card records.

The fundamental element in any public transport network is the *stop*. A stop is where a traveller can board (or alight) a particular means of transport following a given route (consecutive sequence of stops). Using this basic element, public networks build an entire topology combining stops (routes, lines, schedules, etc.).

We define a *trip* as the act of a user travelling across a transport network from one stop (origin) to a different stop (destination). A trip always consists of, at least, one boarding at an origin and an alighting at a destination. However, a trip may not be a direct trip, that is, it could have intermediate stops where the user alights a vehicle and boards a different one to continue towards its final destination. Each individual pair of boarding-alighting will be denoted as *trip stage* from now on.

The prime difficulty in grouping the travellers' card records into trips resides in the fact that, in many cities around the world, the traveller cards are normally only validated when boarding a means of transport. Thus, there is hardly any alighting information. To tackle this problem, we designed a deduction algorithm for the alighting stops following the general strategy presented in previous works [1]. This strategy exploits the expected space-time continuity of consecutive stages of the same trip, as well as the movement symmetry of travellers (each day's movements start where they left off the previous day) to estimate alighting stops based on the boarding data and the topology of the transport network.

As a final addendum to our definition of trips, note that in practice, we are only interested in storing the alighting of the last trip stage, i.e. the final destination of the whole trip. Applying the same continuity principle that allows us to estimate alightings by using boarding data, we can consider the intermediate alighting stops implicit on a consecutive sequence of boardings. Therefore, we can represent a complete trip as a sequence of stops $T = B_1, T_1, \dots, T_n, A_1$, with the first stop always corresponding to the boarding at the origin stop, the last stop with the alighting at the destination, and any intermediate stop being a boarding in a *transfer stop*.

4 Our proposal

We need to store all the ways users travelled between each possible origin-destination pair of a transport network in a compact and indexed way.

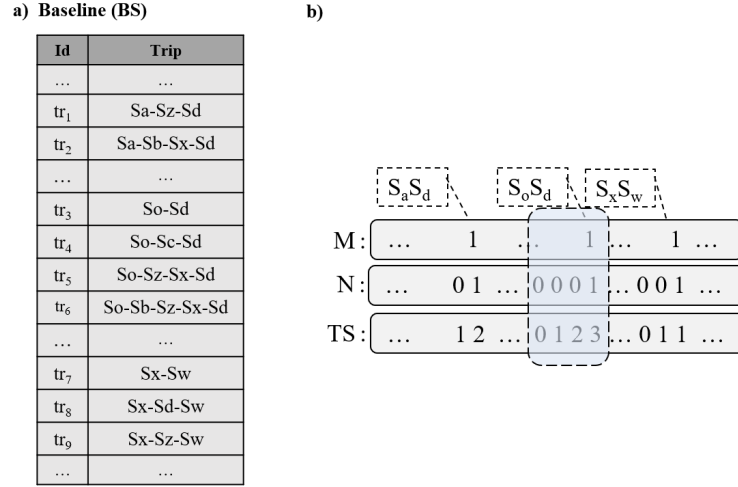


Fig. 1. A readable trip matrix describing each trip and the number of passengers that made it on the left; and its proposed compact representation on the right.

Let $S = [S_1, S_2, \dots, S_\sigma]$ be the ordered set of stops in a public transport network, and let $BS[1..v]$ be the ordered list of different possible trips that travellers actually follow across the considered network. Each of these trips is an ordered list of stops, we denote each of them as *type of trip*. In Figure 1.a, we can see a list of types of trips. For example, the type of trip tr_1 starts at S_a , includes a transfer at stop S_z , and ends at stop S_d . It is possible to compactly store BS by using three vectors:

1. The bitvector $M[1..\sigma^2]$ specifies for which *origin-destination* pairs there are types of trips in BS . Bitvector M simulates a conceptual origin-destination

matrix of size $\sigma \times \sigma$, where the cell at coordinates (i, j) stores a 1-bit if there is at least one trip between S_i and S_j in BS . M is simply the result of storing the contents of that matrix row by row in a vector. If at least one type of trip starts at S_i and ends at S_j , then $M[((i-1) * \sigma) + j]$ stores a 1-bit and a 0-bit otherwise.

2. The bitvector $N[1..v]$ indicates for each pair of origin-destination stops how many different types of trips exist in BS connecting them. More precisely, for each 1-bit in M , N stores in unary the number of different trips between two particular stops. For example, the N bitvector in Figure 1.b shows that there are four ways (0001) to get to S_d from S_o .
3. The integer vector $TS[1..v]$ contains the number of transfer stops for every type of trip represented in N . In the example of Figure 1.b, the four ways to get to S_d from S_o require 0, 1, 2, and 3 transfers.

Any extra information related to a trip can be represented as an array aligned with N or TS . This can be any type of information that may be of interest to transport managers, such as the example we have chosen for this paper: total number of passengers Q that made that trip last month. Additionally, in our implementation, we store Q as its prefix sum representation Q_{PS} in order to allow aggregated and individual queries on it.

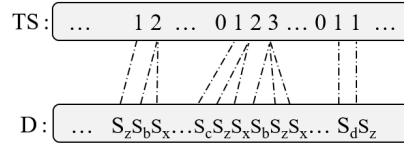


Fig. 2. Plain Transfer Representation (PTR). An additional vector D and its alignment with TS in order to recover transfer stops.

This simple representation is enough for storing and querying direct trips. However, it does not allow recovering information related to transfer stops of a type of trip. The straightforward approach to achieve this, henceforth *Plain Transfer Representation* (PTR), is by just adding a new vector D with all those intermediate stop identifiers (see Figure 2). For the implementation of this method, we replace TS with its prefix sum representation TS_{PS} .

5 Improving our proposal

The previous approach PTR requires sequential searches on D in case of querying data about transfer stops. To improve this, our second proposal, *Tree Transfer Representation* (TTR), indexes the transfer stops identifiers of the existing trips using K^3 -trees, by considering a transfer at stop S_t of a trip that begins in S_o and ends in S_d as a 3D point $\langle S_o, S_d, S_t \rangle$ over a 3D-grid. TTR uses two K^3 -trees, one for trips with exactly one transfer stop and the other one for

every trip with two or more transfer stops, as illustrated in Figure 3. Note that direct trips can already be retrieved by using only M , N and TS (see Figure 1).

From now on, we will refer to the K^3 - tree indexing the trips with only 1 transfer stop as $T1$ - tree (Figure 3.a), and the K^3 - tree indexing trips with 2 or more transfer stops as $T2^+$ - tree (Figure 3.c).

$T1$ - tree: representing a single transfer stop. Any type of trip with a single transfer stop that starts at S_o , ends at S_d , and passes through S_t can be represented as the 3-tuple $\langle S_o, S_d, S_t \rangle$ in a 3D-grid, which can be represented and indexed with a K^3 - tree. Observe that each 1-bit in the last level of the tree (L_1) corresponds to one triplet.

For example, in Figure 3.b, the first (from left to right) shadowed 1-bit in L_1 indicates that there is a type of trip that starts at S_x , ends at S_w , and has only one transfer stop at S_d .

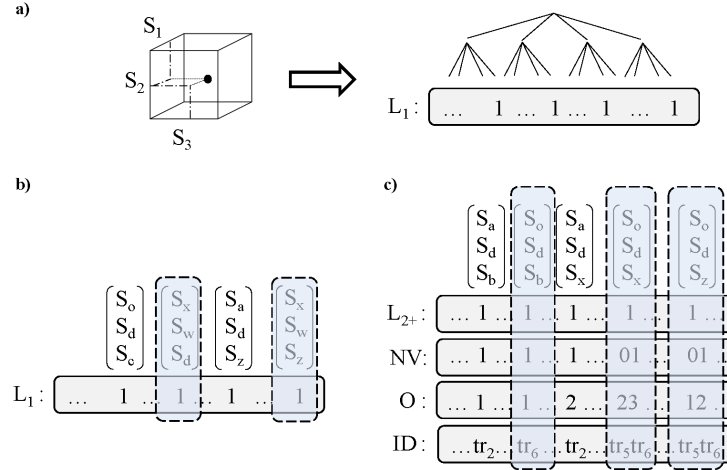


Fig. 3. Tree Transfer Representation (TTR). Additional K^3 - trees ($T1$ and $T2^+$) to efficiently index transfer stops.

$T2^+$ - tree: representing two or more transfer stops. This tree is built following the same idea of $T1$ - tree, but considering the type of trips with two or more transfer stops. However, observe that now, for a given triplet $\langle S_o, S_d, S_t \rangle$, we may have several types of trips in BS that start at S_o , end at S_d , and make a transfer in S_t .

This particularity requires three additional data structures:

1. A bitvector NV for storing the number of types of trips (in unary) that start at S_o , end at S_d , and make a transfer in S_t .

2. For each of those types of trips, the array O indicates the position of the transfer stop at S_t within the stops of the corresponding type of trip. Observe in Figure 3.c that these values are aligned with the values on NV .
3. For each position of NV , the array ID stores the identifier of the corresponding type of trip, which is the position of the trip within BS .

Figure 3.c. shows an example, we only display the last level of the $K^3 - tree$ (L_{2+}). Each 1-bit in L_{2+} corresponds to the triplet of stops displayed above it (shown only for ease of understanding). Focusing on the triplet $\langle S_o, S_d, S_x \rangle$, its 1-bit in L_{2+} means that there is at least one type of trip with origin in S_o , a transfer in S_x , and destination S_d . However, there can be more than one type of trip having that property, and each of them may have other transfers at different stops. In our example, there are two types of trips connecting S_o and S_d through S_x , signalled with the bits 01 in NV . The array O indicates that in the first type of trip, S_x is the second transfer stop of the type of trip whereas, in the second type of trip, S_x is the third transfer stop. The array ID stores the position in BS of the corresponding type of trip, observe that in our example, the two types of trips are tr_5 and tr_6 (see Figure 1).

6 Supported queries

6.1 Obtaining the types of trips to travel from the origin stop S_o to the destination stop S_d (*getTrips*)

Given an origin stop S_o and a destination stop S_d , this query returns all the existing types of trips between S_o and S_d . The first step in this query is to determine the range in N corresponding to the trips between the target stops. This can be solved using M and N , for both PTR and TTR. As an illustrative example, suppose we want to recover the types of trips between origin S_o and destination S_d , as shown in Figure 1.

First, we need to determine if at least one trip between S_o and S_d does exist. This is true if and only if the value in the position $pos = ((o - 1)\sigma) + d$ of M is equal to 1, where o and d are the indexes of the origin and destination stops respectively, and σ the total number of stops in the network. Then, the number of existing 1-bits in $M[1..pos]$ is computed by using $q = rank_1(M, pos)$. After that, the positions of the q -th and the $(q + 1)$ -th 1-bits in N are calculated by using $\langle i, j \rangle = \langle select_1(N, q) + 1, select_1(N, q + 1) \rangle$, returning the range $\langle i, j \rangle$, the number of positions of this range is the number of different types of trips. As it can be seen highlighted in Figure 1.b., there are four different types of trips between the pair (S_o, S_d) .

The next step is to recover the transfer stops of those trips. This process varies depending on the version:

PTR: Conceptually, we retrieve the range $[i..j]$ of TS to obtain the number of transfer stops on each trip, and then search D for the corresponding stop

identifiers. Given that PTR (unlike in the figure that uses a plain representation) uses the prefix sum representation TS_{PS} , for each position k within $[i..j]$, we recover the range $\langle l, r \rangle = \langle TS_{PS}[k], TS_{PS}[k+1] \rangle$, which corresponds to the range in D that stores the transfer stops of the type of trip represented by position k . Then, for each k , the type of trip $\langle S_o, D[l..r], S_d \rangle$ is included in the response. Note that a direct trip occurs whenever $l = r$, in which case the trip $\langle S_o, S_d \rangle$ is included in the response.

TTR: First, if there is a 0 within $TS[i..j]$ (direct trip), $\langle S_o, S_d \rangle$ is included in the response. Second, if there is a 1 within $TS[i..j]$ (meaning trips with only 1 transfer), then the region $\langle S_o, S_d, S_1 \rangle \times \langle S_o, S_d, S_\sigma \rangle$ is recovered from the $T1 - tree$. We add all the points within this region to the response. In the example of Figure 3.b, we can see that there are two types of trips between S_x and S_w , one uses S_d as a transfer stop while the other uses S_z .

Finally, if there is a 2 or a larger number within $TS[i..j]$ (trips with 2 or more transfer stops), then the same region is recovered from the $T2^+ - tree$ (Figure 3.c), obtaining the p points $\langle S_o, S_d, S_t \rangle$ that exist in the region and their $rank_1$ value over L_{2+} . For each point p and its $rank_1$ value l_r , a $select_1(NV, l_r) + 1$ and a $select_1(NV, l_r + 1)$ would be needed to obtain the range $O[o_l..o_r]$ (highlighted in Figure 3.c) assuming NV was stored as a bitvector, but in practice, given that we store NV as its prefix sum representation NV_{PS} , only one subtraction $NV_{PS}[l_r + 1] - NV_{PS}[l_r]$ is needed to obtain the same range.

Observe in Figure 3.c that there are three 1-bits in L_{2+} corresponding to types of trips with origin S_o and destination S_d . The first one (from left to right) indicates that S_b is the first transfer stop of the type of trip tr_6 , the second 1-bit, indicates that the third transfer stop of tr_6 is S_x and that the second stop of tr_5 is also S_x . Finally, the third 1-bit, signals that the second stop of tr_6 is S_z and that the first stop of tr_5 is also S_z .

6.2 Obtaining the total number of people who made a trip starting at S_o and ending at S_d (*getPeople*)

The algorithm for this query is the same for both PTR and TTR. The first step is again to find the range in N $\langle i, j \rangle$ that corresponds with the trips between the target stops. Using the structures depicted in Figure 1.b, the results would be the sum of all the values within that range in the vector Q (for the example of origin S_o and destination S_d , see the highlighted range in Figure 1.b), given that Q uses a prefix sum representation, the result is computed simply as $Q_{PS}[j+1] - Q_{PS}[i]$.

6.3 Obtaining all the origin and destination stops that uses S_t as transfer stop (*getOriginDestinations*)

PTR: First, the range containing all the trips starting at S_t must be discarded, as we are only interested in trips where S_t was boarded as a transfer stop.

The range $\langle i, j \rangle$ in M , representing the trips starting at S_t , can be calculated as $i = ((t-1)\sigma) + 1$ and $j = 1 + i + \sigma$. Then by using *rank* and *select* operations on M and N , in a similar way to the first step of the *getTrips* query, we obtain the corresponding range in N .

Then, we search sequentially D for S_t appearances and, for each of those positions, its corresponding positions in N (excluding those falling in the range where S_t is the starting stop). For each position k of N obtained in the previous step, we compute $r_1 = \text{rank}_1(N, k) + 1$ and then, $\text{pos} = \text{select}_1(M, r_1) - 1$. This recovers the position in M that corresponds to the k -th trip in N . Finally, the origin-destination pair is calculated by using $\langle S_o, S_d \rangle = \langle \lfloor \text{pos}/\sigma \rfloor + 1, (\text{pos} \% \sigma) + 1 \rangle$. If the obtained destination is different from S_t , the pair is added to the result with a count of 1 unless it was already present, in that case, then a 1 is added to its count.

TTR: First, the region $\langle S_1, S_1, S_t \rangle \times \langle S_\sigma, S_\sigma, S_t \rangle$ is searched on each tree. In the case of the $T1 - \text{tree}$, just the returned p points $\langle S_o, S_d, S_t \rangle$ are needed. All the pairs $\langle S_o, S_d \rangle$ are added to the result with a count of 1.

In the case of $T2^+ - \text{tree}$, for each position p with a 1-bit in L_{2+} returned by the search within $\langle S_1, S_1, S_t \rangle \times \langle S_\sigma, S_\sigma, S_t \rangle$, we compute l as the $\text{rank}_1(L_{2+}, p)$. Then, for each l , the value $NV_{PS}[l+1] - NV_{PS}[l]$ is added to the count corresponding to the origin-destination designed by the considered position p of L_{2+} .

7 Experimental evaluation

The experiments were performed on a server with an Intel(R) Xeon(R) E5-2470 @ 2.30GHz CPU and a main memory of 4x16 GB DDR3-1067 MHz. All algorithms were programmed using C++11 language, with several data structures of the SDSL Library [7], and compiled using g++ version 8.3.0 with `-O3 -DNDEBUG -march=native` options. All given runtimes are real (wallclock) times.

7.1 Input data

As explained, the computation of real trips from only the boarding information is a complex task, which is being carried out in parallel in the same project. Since those trips are not ready yet, we opted to test our developed methods using trips simulated over the A Coruña (Spain) bus transport network instead, containing 24 different bus lines and a total number of stops $\sigma = 1,101$.

Partial representations of the transport network were used to create four datasets of different sizes (detailed in Table 1). Each dataset contains the trip of 150 million users that were simulated through a random-walk algorithm over the corresponding transport network.

Table 1. Description of the datasets.

Dataset	# bus lines	# stops (σ)	# trips (v)
coruna-25	6	259	7,898,453
coruna-50	12	531	15,591,174
coruna-75	18	845	21,412,796
coruna-100	24	1,101	22,898,895

7.2 Used methods

This work includes the **baseline** method proposed in Figure 1.a, containing all the registered trips ordered by origin stop, then by destination stop and then by trip length. This order allows answering *getTrips* and *getPeople* by using binary searches, while *getOriginDestinations* is solved using a sequential search.

In addition to our proposed methods, PTR and TTR, we also include space-efficient versions for each one, called PTR-C and TTR-C. In PTR-C, the array D was stored using the minimum amount of bits needed, that is, $\log_2 \sigma$ bits to represent each element, while the prefix sums TS_{PS} and Q_{PS} were stored using the sampling technique. The same technique was used in the arrays TS , O , ID and the prefix sums NV_{PS} and Q_{PS} of TTR-C. In all methods, the data type used in each array is the smallest integer data type of C++ that can represent the biggest value stored in the array.

All the tested methods are constructed once for each dataset and stored on disk. To run an experiment over any method, first, the data structure is loaded into main memory and then the corresponding experiment is executed.

7.3 Results

Table 2 shows the resulting sizes of every approach tested for the four experimental datasets, in MBs, as well as the compression ratios compared to the baseline.

Table 2. Space usage on each dataset (MB).

	baseline	PTR	PTR-C	TTR	TTR-C
coruna-25	159.90	70.00 (43.78%)	28.01 (17.52%)	63.98 (40.01%)	35.51 (22.21%)
coruna-50	313.20	134.77 (43.03%)	56.77 (18.12%)	126.61 (40.42%)	68.52 (21.88%)
coruna-75	426.75	181.40 (42.51%)	75.16 (17.61%)	177.82 (41.67%)	98.44 (23.07%)
coruna-100	453.34	191.18 (42.17%)	83.74 (18.47%)	189.90 (41.89%)	109.54 (24.16%)

Notice that all of our proposals occupy less space than the **baseline** on each dataset. The non-compressed versions use between 40.01% and 43.78% the size of the **baseline**. The compressed version PTR-C uses between 17.52% and 18.47% the space of the baseline, while TTR-C needs between 22.21% and 24.16%.

After building the data structures, our first experiment consisted in testing the speed of the query *getTrips*. Each method had to reconstruct all the kinds of trips that started on a random origin S_i and ended on a random S_j . This

was repeated 1,000,000 times on each method and the result is the average time to reconstruct one kind of trip that exists between an origin and a destination. Each method searched exactly the same values on each query. Table 3 shows the result of this experiment in $\mu s/trip$.

Table 3. Time of *getTrips* to recover one trip ($\mu s/trip$).

	baseline	PTR	PTR-C	TTR	TTR-C
coruna-25	0.077	0.067	0.313	0.322	0.332
coruna-50	0.115	0.076	0.318	0.777	0.777
coruna-75	0.18	0.078	0.317	1.804	1.761
coruna-100	0.252	0.074	0.317	2.612	2.602

The obtained results of this first experiment prove that PTR and PTR-C recover single trips in $O(1)$. The time difference between both versions can only be explained by the access time to the arrays since both use exactly the same algorithm. The **baseline** method is always slower than PTR and always faster than every other method. TTR and TTR-C do not perform well on this query, since both versions need to access both $T1$ and $T2+$ trees in the worst case (if there are trips with one or more transfer stops to be recovered). Note that the access time of the arrays that affected the results in PTR and PTR-C does not affect TTR and TTR-C, achieving similar times.

The next experiment consisted in testing the speed of *getPeople* query. Each method had to recover the total number of people that started their trip on a random origin S_i and ended on a random S_j . This was repeated 1,000,000 times on each method, using exactly the same values in each query. The displayed value is the average time to answer one query. Table 4 shows the result of the experiment in $\mu s/query$.

Table 4. Time of *getPeople* query to recover one query ($\mu s/query$).

	baseline	PTR	PTR-C	TTR	TTR-C
coruna-25	2.688	0.735	0.734	0.719	0.727
coruna-50	3.497	0.619	0.628	0.766	0.770
coruna-75	3.975	0.773	0.770	0.682	0.639
coruna-100	4.22	0.469	0.478	0.729	0.472

Theoretically, our proposals should answer the *getPeople* query in $O(1)$, since the range in N can be obtained in $O(1)$ and, given that Q is stored as a prefix sum, we can also return any range of data in $O(1)$. In the case of the **baseline**, it answers the query in $O(k + \log_2(v))$, with k being the number of trips that start in S_o and end in S_d and $\log_2(v)$ the additional time to return the range using the binary searches. The results of this particular experiment show that PTR, PTR-C, TTR and TTR-C can answer the *getPeople* query at least $3.6\times$ faster than the **baseline**.

The final experiment consisted in testing the speed of the methods in the *getOriginDestinations* query. Each method had to recover a list containing all

the origin-destination pairs that used the stop S_t as a transfer stop. This was repeated 1,000 times on each method, using exactly the same values in each query. The results are shown in Table 5, containing the average time to answer one query in *ms/query*.

Table 5. Time of *getOriginDestinations* query to recover one query (*ms/query*).

	baseline	PTR	PTR-C	TTR	TTR-C
coruna-25	53.079	31.147	1927.08	6.898	6.777
coruna-50	101.962	58.446	3749.51	14.649	14.449
coruna-75	141.806	85.286	5027.86	22.070	22.040
coruna-100	152.269	95.884	5252.03	22.409	22.329

As it can be seen in Table 5, tree-based methods always achieve better performance as the intermediate stops are indexed while the baseline and PTR need extra steps to reach them. Thus, TTR and TTR-C can return all the existing points that contain the searched transfer stop in logarithmic time, while the **baseline**, PTR and PTR-C need to do a sequential search over all the stored trips. Note that PTR is always faster than the **baseline**, while PTR-C access times lead to much worse results.

8 Conclusions and future work

This work introduced two compact data structures that represent user trips over a public transport network, PTR and TTR. We fed these data structures with the simulated data of 150 million users travelling across the entire A Coruña bus transport network. The results showed that our main approaches use less than 44% of the space required by the **baseline** to store these trips, while the space-efficient versions we implemented, PTR-C and TTR-C, uses less than 18.47% and 24.16% of the space used by the **baseline** respectively.

The proposed representations are able to efficiently answer three queries of interest: retrieve trips from an origin stop S_i to a destination stop S_j (*getTrips*), get the total number of people who travelled between any pair origin-destination (*getPeople*), and calculate a list containing all the origin-destination pairs who used a certain stop S_t as transfer stop (*getOriginDestination*). In terms of results, it is advisable to use our compressed tree version TTR-C if transfer stops retrieval is needed and the plain version PTR in all other cases.

As future work, we plan to create a hybrid approach between PTR and TTR-C that could allow faster performance on all the queries, while still using less space compared to our **baseline**. We also intend to expand our query capabilities using feedback from transport administrators and test our methods using real trip data from the city of Madrid.

References

1. Alsger, A., Assemi, B., Mesbah, M., Ferreira, L.: Validating and improving public transport origin–destination estimation algorithm using smart card fare data. *Transportation Research Part C: Emerging Technologies* **68**, 490–506 (2016)
2. de Bernardo, G., Álvarez-García, S., Brisaboa, N.R., Navarro, G., Pedreira, O.: Compact querieable representations of raster data. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) *String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings. Lecture Notes in Computer Science*, vol. 8214, pp. 96–108. Springer (2013)
3. Brisaboa, N.R., Bernardo, G.d., Gutiérrez, G., Luaces, M.R., Paramá, J.R.: Efficiently Querying Vector and Raster Data. *The Computer Journal* **60**(9), 1395–1413 (02 2017)
4. Brisaboa, N.R., Gómez-Brandón, A., Navarro, G., Paramá, J.R.: Gract: A grammar-based compressed index for trajectory data. *Information Sciences* **483**, 106–135 (2019)
5. Brisaboa, N.R., Ladra, S., Navarro, G.: Compact representation of web graphs with extended functionality. *Inf. Syst.* **39**, 152–174 (2014)
6. Brisaboa, N., Fariña, A., Galaktionov, D., V Rodeiro, T., Rodriguez, A.: Improved structures to solve aggregated queries for trips over public transportation networks. *Information Sciences* **584** (11 2021)
7. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: Plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) *Experimental Algorithms*. pp. 326–337. Springer International Publishing, Cham (2014)
8. Jacobson, G.: Space-efficient static trees and graphs. In: *30th annual symposium on foundations of computer science*. pp. 549–554. IEEE Computer Society (1989)
9. Navarro, G.: *Compact Data Structures: A Practical Approach*. Cambridge University Press, USA (2016)
10. Samet, H.: *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann (2006)