A Compact Representation for Trips over Networks built on self-indexes[☆]

Nieves R. Brisaboa[a], Antonio Fariña[a,*], Daniil Galaktionov H.[a], M. Andrea Rodriguez[b]

[a] *University of A Coruña, Database Laboratory, Spain.*
[b] *University of Concepción, Department of Computer Science, Chile.*

Abstract

Representing the movements of objects (trips) over a network in a compact way while retaining the capability of exploiting such data effectively is an important challenge of real applications. We present a new *Compact Trip Representation* (CTR) that handles the spatio-temporal data associated with users' trips over transportation networks. Depending on the network and types of queries, nodes in the network can represent intersections, stops, or even street segments.

CTR represents separately sequences of nodes and the time instants when users traverse these nodes. The spatial component is handled with a data structure based on the well-known Compressed Suffix Array (CSA), which provides both a compact representation and interesting indexing capabilities. The temporal component is self-indexed with either a Hu-Tucker-shaped Wavelet-tree or a Wavelet Matrix that solve range-interval queries efficiently. We show how CTR can solve relevant counting-based spatial, temporal, and spatio-temporal queries over large sets of trips. Experimental results show the space requirements (around 50-70% of the space needed by a compact unindexed baseline) and query efficiency (most queries are solved in the range of 1-1000 microseconds) of CTR.

Keywords: Trips on networks, counting queries, self-index, compression

[☆]Funded in part by European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 690941 (project BIRDS). N. Brisaboa, A. Fariña, and D. Galaktionov are partially funded by Xunta de Galicia/FEDER-UE [CSI: ED431G/01 and GRC: ED431C 2017/58]; by MINECO-AEI/FEDER-UE [Datos 4.0: TIN2016-78011-C4-1-R and ETOME-RDFD3: TIN2015-69951-R]; and by MINECO-CDTI/FEDER-UE [CIEN: LPS-BIGGER IDI-20141259 and INNTERCONECTA: uForest ITC-20161074]. M. A. Rodríguez is partially funded by Fondecyt [1170497] and the Complex Engineering Systems Institute (CONICYT: FBO16). An early partial version of this article appeared in *Proc SPIRE'16* [1].

*Corresponding author

Email addresses: brisaboa@udc.es (Nieves R. Brisaboa), fari@udc.es (Antonio Fariña), d.galaktionov@udc.es (Daniil Galaktionov H.), andrea@udec.cl (M. Andrea Rodriguez)

1. Introduction

Due to the current advances in sensor networks, wireless technologies, and RFID-enabled ubiquitous computing, data about moving-objects (also referred to as trajectories) is an example of massive data relevant in many real applications. Think in the notion of Smart Cities, where the implementation of new technologies in public transportation systems has become more widespread all around the world in the last decades. For instance, nowadays many cities -from London to Santiago- provide the users of the public transportation with smartcards that help in making the payment to access buses or subways an easier task. Even though smartcards may only collect data when users enter to the transportation system, it is possible to derive the users' trip (when they enter and leave the system) using historical data and transportation models [2]. When having this data and considering that for privacy issues the individual trajectory must not be revealed, counting or aggregate queries of trajectories become useful tools for traffic monitoring, road planning, and road navigation systems.

New technologies and devices generate a huge amount of highly detailed, real-time data. Several research exists about moving-object databases (MODs) [3–6] and indexing structures [7–13]. They, however, have addressed typical spatio-temporal queries such as time slice or time interval queries that retrieve trajectories or objects that were in a spatial region at a time instant or during a time interval. They were not specially designed to answer queries that are based on counting, such as the number of distinct trips, which are more meaningful queries for public-transportation or traffic administrators. This problem was recently highlighted in [14], where authors describe an approximate query processing of aggregate queries that count the number of distinct trajectories within a region. In this work, we concentrate on counting-based queries on a network, which includes the number of trips starting or ending at some time instant in specific stops (nodes) or the top-k most used stops of a network during a given time interval.

The work in this paper proposes a new structure named *Compact Trip Representation* (CTR) that answers counting-based queries and uses compact self-indexed data structures to represent the large amount of trips in a compact space. CTR combines two well-known data structures. The first one, initially designed for the representation of strings, is Sadakane's Compressed Suffix Array (CSA) [15]. The second one is the Wavelet Tree [16] (WT). To make the use of the CSA possible, we define a trip or trajectory of a moving object over a network as the temporally-ordered sequence of the nodes the trip traverses. An integer ID is assigned to each node such that a trip is a string with the IDs of the nodes. Notice that this representation avoids the cost of storing coordinates to represent the location users pass through during a trip. It is just enough to identify the stops or nodes and when necessary to map these nodes to geographic locations. Then a CSA, over the concatenation of these strings (trips), is built with some adaptations for this context. In addition, we discretize the time in periods of fixed duration (i.e. timeline split into 5-minute intervals) and each time segment is identified by an integer ID. In this way, it is possible to store the times when trips reach each node by associating the corresponding time ID with each node in each trip. The sequence of times for all the nodes within a trip is self-indexed with a WT to efficiently answer temporal

and spatio-temporal queries.

We experimentally tested our proposal using two sets of data representing trips over two different real public transportation systems. Our results are promising because the representation uses around 50% of its original size and answers most of our spatial, temporal, and spatio-temporal queries within 1−1000 microseconds. No experimental comparisons with classical spatial or spatio-temporal index structures were possible, because none of them were designed to answer the types of queries in this work. Our approach can be considered as a proof of concept that opens new application domains for the use of well-known compact data structures such as the CSA and the WT, creating a new strategy for exploiting trajectories represented in a self-indexed way.

The organization of this paper is as follows. Section 2 reviews previous works on trip representations. It also presents CSA and WT upon which we develop our proposal. We pay special attention to show the internals of those structures and discuss also their properties and functionality. Then, in Section 2.3.2, we also present the *wavelet matrix* (WM) and show how to create a *Hu-Tucker-shaped* WT (WTHT). These are the two variants of WT we use to represent temporal data. In Section 3, we present the main counting-based queries that are of interest for a transportation network. In Section 4, we present CTR and show how to reorganize a dataset of trips to allow a CSA to handle the spatial data and a WT-based structure to manage the temporal data. Section 5 shows how CTR represents the spatial component and how spatial queries are dealt with. In Section 6, we focus on how to represent the temporal component of trips and how to answer temporal queries. We also include a brief comparison of the space/time trade-off of WM and WTHT. In Section 7, we show how spatio-temporal queries are solved by CTR, and Section 8 includes our experimental results. Finally, conclusions and future work are discussed in Section 9.

2. Previous Work

2.1. Models of trajectory and types of queries

There is a large amount of work on data models for moving-object data [3–6, 17–20]. Basically, a moving-object data model represents the continuous change of the location of an object over time, what is called the trajectory of the object.

Moving-object data is an example of big data that differ in the representation of location, contextual or environmental information where the movement takes place, the time dimension that can be continuous or discrete, and the level of abstraction or granularity on which the trajectories are described [21]. A common classification of trajectories distinguishes free from network-based trajectories. *Free trajectories* or Euclidean trajectories are a sequence of GPS points represented by an ad-hoc data type of moving points [17–19]. *Network-based* trajectories are a temporal ordered sequence of locations on networks. This trajectory model includes a data type for representing networks and for representing the relative location of static and moving points on the network [22]. In a recent work, *network-matched trajectories* are defined to avoid the need of a mobile map at the moving-object side [23].

The definition of trajectories at an abstract level must be materialized in an internal representation with access methods for query processing. An early and

broad classification of spatial-temporal queries for *historical positions* of moving objects [8] identifies coordinate- and trajectory-based queries. Coordinate-based queries include the well-known *time-slice*, *time-interval* and *nearest-neighbor queries*. Examples are *find objects or trajectories in a region at a particular time instant or during some time interval*. Another important example of range-based queries is *find the k-closest objects with respect to a given point at a given time instant*. Trajectory-based queries involve topology of trajectories (e.g., overlap and disjoint) and information (e.g., speed, area, and heading) that can be derived from the combination of time and space. An example of such queries would be *find objects or trajectories that satisfy a spatial predicate (eg., leave or enter a region) at a particular time instant or time interval*. There also exist combined queries addressing information of particular objects: *Where was object X at a particular time instant or time interval*. In all previous queries, the results are individual trajectories that satisfy the query constraints.

When dealing with large datasets of trajectories, and due to privacy issues, the individual trajectory cannot be revealed, then anonymized and aggregated trajectories are of concern. In this context, we can further distinguish range- from trajectory-based queries. Range queries impose constraints in terms of a spatial location and temporal interval. Examples of these queries are to retrieve the number of distinct trajectories that intersect a spatial region or spatial location (stop) at a given time instant or time interval, retrieve the number of distinct trajectories that start at a particular location (stop) or in a region and/or end in another particular location of region, retrieve the number of trajectories that follow a path, and retrieve the top-k locations (stops) or regions with the larger number of trajectories that intersect at a given time instant or time interval. Trajectory-based queries require not only to use the spatio-temporal points of trajectories but also the sequence of these points. Examples of such queries are to find the number of trajectories that are heading (not necessarily ending at) to a spatial location during a time interval, find the destination of trajectories that are passing through a region during a time interval, find the number of starting locations of trajectories that go or pass through a region during a time interval.

2.2. Trajectory indexing

Many data structures have been proposed to support efficient query capabilities on collections of trajectories. We refer to [7, Chapter 4] for a comprehensive and up-to-date survey on data management techniques for trajectories of moving objects. We can broadly classify these data structures into two groups: those that index trajectories in free space and those that index trajectories that are constrained to a network.

In free space, it is common to see spatial indexes that extend the *R-Tree* index [24] beyond a simple *3D R-Tree* where the time is one of the dimensions. Two examples of such indexes can be found in [8] where the authors present two fundamental variations of the *R-Tree*: the *STR-Tree* and the *TB-Tree*. Both indexes modify the classical construction algorithm for the *R-Tree*, where the nodes are not only grouped by the spatial distance among the indexed objects, but also by the trajectories they belong to. In the *MV3R-tree* [9], the construction takes into account temporal information of the moving objects, adapting ideas from the Historical *R-Tree* [25]. Another interesting approach is described in [26], where the authors split trajectories of moving objects across

partitions of space, indexing each partition separately. This improves query efficiency, as only the partitions that intersect a query region are accessed.

R-Tree adaptations can also be useful when the trajectories are constrained to a network. They exploit the constraints imposed by the topology of the network to optimize the data structure. This is the case of the *FNR-tree* [10], which consists of a *2D R-Tree* to index the segments of the trajectories over the network, and a forest of *1D R-Trees* used to index the time interval when each trajectory is moving through each segment of the network. The *MON-Tree* [11] can be seen as an improvement over the *FNR-Tree*, saving considerable space by indexing MBRs of larger network elements (edge segments or entire roads) and reducing the number of disk accesses at query time. Both indexes are outperformed by the *TMN-Tree* [27] in query time, which indexes whole trajectories of moving objects with a *2D R∗-Tree* and indexing the temporal component with a B^+-Tree, which proves to be more efficient for that application than the *R-Tree*.

PARINET is another interesting alternative to represent trajectories constrained to a network [12]. It partitions trajectories into segments from an underlying road network using a complex cost model to minimize the number of disk accesses at query time. It takes into account the spatial relations among the indexed network elements, as well as some statistics of the data to index. Then it adds a temporal B^+-tree to index the trajectory segments from each road. Those indexes permit *PARINET* to filter out candidate trajectory segments matching time constraints at query time. The same ideas were used in *TRIFL* [28], where the cost model is adapted for flash storage.

All previous data structures were designed to answer spatio-temporal queries, where the space, in particular geographic coordinates, and time are the main filtering criteria. Examples of such queries are: *retrieve trajectories that crossed a region within a time interval*, *retrieve trajectories that intersect*, or *retrieve the k-best connected trajectories* (i.e., the most similar trajectories in terms of a distance function). Yet, they could not easily support queries such as *the number of trips starting in X and ending at Y*. A recent work in [14] proposes a method to compute the approximate number of distinct trajectories that cross a region. Note that computing aggregate queries of trajectories in the hierarchical structure of classical spatio-temporal indices is usually done by aggregating the information maintained in index nodes at the higher levels to avoid accessing the raw spatio-temporal data. However, for a trajectory aggregate query, maintaining the statistical trajectory information on index nodes does not work because what matters for these queries is to determine the number of *distinct* trajectories in a spatio-temporal query region.

The application of data compression techniques has been explored in the context of massive data about trajectories. The work by Meratnia and de By [29] adapts a classical simplification algorithm by Douglas and Peucker to reduce the number of points in a curve and, in consequence, the space used to represent trajectories. Potamias *et al.* [30] use concepts, such as speed and orientation, to improve compression. It is also possible [31] to compress a trajectory in a way that the maximum error at query time is deterministic, although the method greatly depends on the distance function to be used.

In [32–34], they focus mainly on how to represent trajectories constrained to networks, and in how to gather the location of one or more given moving objects

from those trajectories. Yet, these works are out of our scope as they would poorly support queries oriented to exploit the data about the network usage such as those that compute the number of trips with a specific spatio-temporal pattern (e.g. *Count the trips starting at stop X and ending at stop Y in working days between 7:00 and 9:00*).

A recent work [35] proposed an indexing structure called *NETTRA* to answer *strict* and *approximate path queries* that can be implemented in standard SQL using B^+-trees and self-JOIN operations. For each trajectory, *NETTRA* represents the sequence of adjacent network edges touched by the trajectory as entries in a table with four columns: id, entering and leaving time, and a hash value of the entire path up to and including the edge itself. Using the hash value for the first and last edge on a query path, *NETTRA* determines whether the trajectory followed a specific path between these edges. Also for *strict path queries*, Koide et al. [36] proposed a spatio-temporal index structure called *SNT-Index* that is based on the integration of a FM-index [37] to store spatial information with a forest of B+trees that stores temporal information. To the best of our knowledge, this makes up the first technique using compact data structures to handle spatial data in this scenario. Yet, in our opinion, *strict path queries* have little interest in the context of exploiting data to analyze the use of a transportation network.

Unlike previous works, we designed an in-memory representation, that targets at solving counting-based queries, and is completely based on the use of compact data structures (discussed in the next section) to make it successful not only in time but also in space needs. Since CTR keeps data in a compressed way, it will permit to handle larger sets of trajectories entirely in memory and consequently to avoid costly disk accesses.

2.3. Underlying Compact Structures of CTR

CTR relies on two components: one to handle the spatial information and another to represent temporal information. The spatial component is based on the well-known a Compressed Suffix Array (CSA) [15]. The temporal component can be implemented with either a Wavelet Tree (WT) [16] or a Wavelet Matrix (WM) [38]. The latter is a variant of WT that performs better when representing sequences built on a large alphabet as we see below.

2.3.1. Sadakane's Compressed Suffix Array (CSA)

Given a sequence $S[1, n]$ built over an alphabet Σ of length σ, the *suffix array* $A[1, n]$ built on S [39] is a permutation of $[1, n]$ of all the suffixes $S[i, n]$ so that $S[A[i], n] \prec S[A[i + 1], n]$ for all $1 \leq i < n$, being $\prec$ the lexicographic ordering. Because A contains all the suffixes of S in lexicographic order, this structure permits to search for any pattern $P[1, m]$ in time $O(m \log n)$ by simply binary searching the range $A[l, r]$ that contains pointers to all the positions in S where P occurs.

To reduce the space needs of A, Sadakane's CSA [15] uses another permutation $\Psi[1, n]$ defined in [40]. For each position j in S pointed from $A[i] = j$, $\Psi[i]$ gives the position z such that $A[z]$ points to $j + 1$. There is a special case when $A[i] = n$, in this case $\Psi[i]$ gives the position z such that $A[z] = 1$. In addition, we could set up a vocabulary array $V[1, \sigma'], (\sigma' \leq \sigma)$ with all the different symbols from Σ that appear in S, and a bitmap $D[1, n]$ aligned to

Ψ so that $D[i] \leftarrow 1$ if $i = 1$ or if $S[A[i]] \neq S[A[i-1]$ ($D[i] \leftarrow 0$; otherwise). Basically, a 1 in D marks the beginning of a range of suffixes pointed from A such that the first symbol of those suffixes coincides. That is, if the i^{th} and $(i+1)^{th}$ one in D occur in $D[l]$ and $D[r]$ respectively, we will have $V[rank_1(D,l)] = V[rank_1(D,x)]\ \forall x \in [l, r-1]$. Note that $rank_1(D,i)$ returns the number of 1s in $D[1,i]$ and can be computed in constant time using $o(n)$ extra bits [41, 42].

By using Ψ, D, and V, it is possible to simulate a binary search for the interval $A[l,r]$ where a given pattern P occurs ($[l,r] \leftarrow bsearch(P)$) without keeping A nor S. Note that, the symbol $S[A[i]]$ pointed by $A[i]$ can be obtained as $V[rank_1(D,i)]$, and we can obtain the following symbol from the source sequence $S[A[i]+1]$ as $V[rank_1(D,\Psi[i])]$, $S[A[i]+2]$ can be obtained as $V[rank_1(D,\Psi[\Psi[i]])]$, and so on. Therefore, CSA replaces S, and it does not need A anymore to perform searches.

However, in principle, Ψ would have the same space requirements as A. Fortunately, Ψ is highly compressible. It was shown to be formed by σ subsequences of increasing values [40] so that it can be compressed to around the zero-order entropy of S [15] and, by using δ-codes to represent the differential values, its space needs are $nH_0 + O(n \log\log \sigma)$ bits. In [43], they showed that Ψ can be split into $nH_k + \sigma^k$ (for any k) *runs* of consecutive values so that the differences within those runs are always 1. This permitted to combine δ-coding of gaps with run-length encoding (of 1-*runs*) yielding higher-order compression of Ψ. In addition, to maintain fast random access to Ψ, absolute samples at regular intervals (every t_Ψ entries) are kept. Parameter t_Ψ implies a space/time trade-off. Larger values lead to better compression of Ψ but slow down access time to non-sampled $\Psi[i]$ values.

In [44], authors adapted Sadakane's CSA to deal with large (integer-based) alphabets and created the *integer-based CSA* (iCSA). They also showed that, in their scenario (natural language text indexing), the best compression of Ψ was obtained by combining differential encoding of runs with Huffman and run-length encoding.

2.3.2. The Wavelet Tree (WT)

Given a sequence $S[1,n]$ built on an alphabet Σ with σ symbols that are encoded with a fixed-length binary code $[0,\sigma)$, a WT [16] built over S is a balanced binary tree where leaves are labeled with the different symbols in S, and each internal node v contains a bitvector B_v. The bitvector in the root node contains the first bit from the codes of all the n symbols in S. Then symbols whose code starts with a 0 are assigned to the left child, and those with codes starting with a 1 are assigned to the right child. In the second level, the bitvectors contain the second bits of the codes of their assigned symbols. This applies recursively for every node, until a leaf node is reached. Leaf nodes can only contain one kind of symbol. The height of the tree is $\log \sigma$, and since the bitvectors of each level contain n bits, the overall size of all the bitvectors is $n \log \sigma$ bits. To calculate the total size of the WT we also need to take into account the space needed to store pointers from each symbol to its corresponding tree node which is $O(\sigma \log n)$ bits. In addition, as we see below the WT reduces the general problem of solving $access(i)$, $rank_c(i)$, and $select_c(i)$ operations to the problem of computing *access*, *rank* and *select* on the bitvectors. Therefore, additional structures to efficiently support those operations add up to $o(n \log \sigma)$

space. The overall size of the WT is $n \log \sigma(1+o(1)) + O(\sigma \log n)$. Figure 1.(left) shows a WT built on the sequence $S = \langle 3\,2\,7\,7\,0\,1\,4\,3\,7\,6\,3\,2\,5\,5\,3 \rangle$ assuming we use a 3-bit binary encoding for the symbols in $\Sigma = [0..7]$. Shaded areas are not included in the WT but help us to see the subsequences handled by the children of a given node.

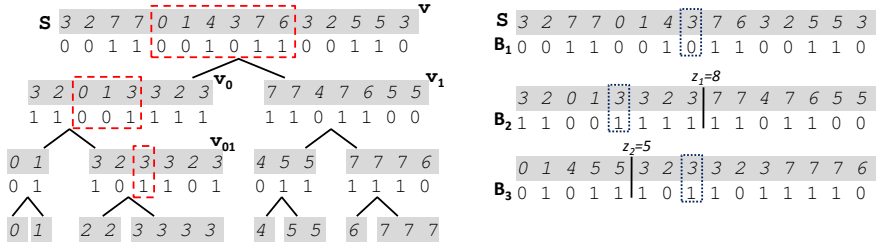

Figure 1: WT (left) and WM (right) for sequence $S = \langle 3\,2\,7\,7\,0\,1\,4\,3\,7\,6\,3\,2\,5\,5\,3 \rangle$. Shaded areas are neither included in the WT nor in the WM.

Among others, the WT permits to answer the following queries in $O(\log \sigma)$ time:

- $access(i)$ returns $S[i]$.
- $rank_c(i)$ returns the number of occurrences of symbol c in $S[1, i]$.
- $select_c(i)$ returns the position of the i-th occurrence of symbol c in S.
- $count(i, j, \alpha, \beta)$ described in [45], returns the number of occurences in $S[i, j]$ of the symbols between α and β.

To solve $access(i)$ and $rank_c(i)$ operations we traverse the WT from the root until we reach a leaf. In the case of $rank_c(i)$ we descend the tree taking into account the encoding of $c = c_1 c_2 ...$ in each level. Being B the bitmap in the root node, if $c_1 = 0$ we move to the left child and set $i \leftarrow rank_0(i)$; otherwise we move to the right child and set $i \leftarrow rank_1(i)$. We proceed recursively until we reach a leaf where we return i. $access(i)$ is solved similarly, but at each level, we either move left or right depending on if $B[i] = 0$ or $B[i] = 1$ respectively. The leaf where we arrive corresponds to the symbol $c = S[i]$ which is returned. To solve $select_c(i)$ we traverse the tree from the leaf corresponding to symbol c until the root. At level j, we look at the value of the j-th bit of the encoding of c (c_j). If $c_j = 0$ we set $i \leftarrow select_0(B_{j-1})$ (where B_{j-1} is the bitmap of the parent of the current node), otherwise we set $i \leftarrow select_1(B_{j-1})$. Then we move to level $j - 1$ and proceed recursively until the root, where the final value of i is returned.

In this work we are also interested in operation $count(i, j, \alpha, \beta)$ that allow us to count the number of occurrences of all the symbols $c \in [\alpha, \beta]$ within $S[i, j]$. Assuming the encodings of the symbols $c \in [\alpha, \beta]$ form also a contiguous range this can be solved in $O(\log \sigma)$ [38, 45]. The idea is to traverse the tree from the root and descend through the nodes that cover the leaves in $[\alpha, \beta]$. At each node v (whose bitmap is B_v and range $B_v[i, j]$ is considered), that covers symbols in range $[a, b]$ we check whether $[a, b] \subseteq [\alpha, \beta]$. In that case we sum

$j - i + 1$ occurrences. If both ranges are disjoint we found a node not covering the range $[\alpha, \beta]$ and stop the traversal. Similarly, if range $[i, j]$ becomes empty traversal stops on that branch. Otherwise, we recursively descend from node v to their children v_l and v_r where we map the interval $B_v[i, j]$ into $B_{v_l}[i_l, j_l]$ and $B_{v_r}[i_r, j_r]$ with $rank_0$ operation. In practice, $i_l \leftarrow rank_0(i-1)+1$, $j_l \leftarrow rank_0(j)$ and $i_r \leftarrow i - i_l + 1$, $j_r \leftarrow j - j_l$. For more details and pseudocodes see [38, 45, 46]. In Figure 1.(left) we can see the nodes ($\mathbf{v}$, $\mathbf{v_0}$, and $\mathbf{v_{01}}$) that must be traversed, and the ranges within the bitmaps in those nodes, to solve $count(5, 10, 3, 7)$. Therefore, we want to compute the number of occurrences of symbols between 3 and 7 that occur within $S[5, 10]$. We start in the root node $\mathbf{v}$, where $B_v[5, 10]$ contains 3 zeroes and 3 ones. We compute $i_l = rank_0(4) + 1 = 3, j_l = rank_0(10) = 5$, and $i_r = 5 - 3 + 1 = 3, j_r = 10 - 5 = 5$. At this point we could move to $\mathbf{v_1}$, but we can see that all the encodings of the symbols $4, 5, 6, 7$ start by 1 and they are covered by $\mathbf{v_1}$. Therefore, we report $i_r - i_l + 1 = 3$ occurrences of symbols in range $[4, 7]$ and no further processing is done in the subtree whose root is $\mathbf{v_1}$. However, we descent to $\mathbf{v_0}$ since $0s$ in $B_v[5, 10]$ could belong to any symbol in range $[0, 3]$, and we have to track only occurrences of symbol 3. We check $B_{v_0}[3, 5]$ and compute $i_l = rank_0(3) + 1 = 1, j_l = rank_0(5) = 2$, and $i_r = 3 - 1 + 1 = 3, j_r = 5 - 2 = 3$. Since the second bit of the encoding of symbol 3 is a 1 (as for symbol 2), we can discard descending on the left child of $\mathbf{v_0}$ and move only to its right child $\mathbf{v_{01}}$ where we are interested in the range $B_{v_{01}}[3, 3]$. Since $\mathbf{v_{01}}$ covers both symbols 2 and 3, and the third bit of the encoding of 2 is a zero whereas it is a one for 3, we do only need to count the number of ones in $B_{v_{01}}[3, 3]$. After computing $i_l = rank_0(2) + 1 = 2, j_l = rank_0(3) = 1$, and $i_r = 3 - 2 + 1 = 2, j_r = 3 - 1 = 2$, we report $j_r - i_r + 1 = 1$ occurrence of symbol 3. Therefore, we conclude $count(5, 10, 3, 7) = 3 + 1 = 4$.

One way of reducing the space needs of a WT consists in compressing its bitvectors [47]. Among others (i.e. Golynski et al. [48] which is better theoretically), Raman el al. technique [49] (RRR) is, in practice, one of the best choices. The overall size of the WT becomes $nH_0(S) + o(n \log \sigma) + O(\sigma \log n)$, whereas operations still require $O(\log \sigma)$ time.

Another way of compressing a WT is to use a prefix-free variable-length encoding for the symbols. For example, Huffman [50] code can be used to build a Huffman-Shaped WT [51], where the tree is not balanced anymore. The size reduces to $n(H_0(S) + 1) + o(n(H_0(S) + 1)) + O(\sigma \log n)$,[1] and average time becomes $O(H_0(S))$ for *rank*, *access*, and *select* (worst-case time is still $O(\log \sigma)$ [52]). By using compressed bitvectors [38] space can be reduced even further to $nH_0(S) + o(n(H_0(S) + 1)) + O(\sigma \log n)$. Unfortunately, the Huffman codes given to adjacent symbols are no longer contiguous, and it is not possible to give a $O(\log \sigma)$ bound for $count(i, j, \alpha, \beta)$ anymore, even if the code is canonical. Hu-Tucker codes [53] can be used instead.[2] Compression degrades slightly with respect to using Huffman coding,[3] but the codes for adjacent symbols are lexi-

[1] $O(\sigma \log n)$ term includes both the tree pointers and the size of the Huffman model.

[2] Hu-Tucker [53] is an optimal prefix code that preserves the order of the input vocabulary. This means that the lexicographic order of the output variable-length binary codes is the same as the order of the input symbols.

[3] Being L_h and L_{ht} the average codeword length of Huffman coding and Hu-Tucker codes respectively, it holds: $H_0 \leq L_h \leq H_0 + 1$ and $H_0 \leq L_{ht} \leq H_0 + 2$ (see [54] (pages 122-123),

cographically contiguous. This permits to solve *count* efficiently. The size of a Hu-Tucker-shaped WT (WTHT) can be bounded to $n(H_0(S)+2)+o(n(H_0(S)+1))+O(\sigma \log n)$ and can be reduced to $nH_0(S)+o(n(H_0(S)+1))+O(\sigma \log n)$ by using compressed bitvectors as well.

The Wavelet Matrix (WM). For large alphabets, the size of the WT is affected by the term $O(\sigma \log n)$. A pointerless WT [57] permits to remove[4] that term by concatenating all the bitmaps level-wise and computing the values of the pointers during the WT traversals. The operations on a pointerless WT have the same time complexity but become slower in practice.

By reorganizing the nodes in each level of a pointerless WT, the *Wavelet Matrix* (WM) [38] obtains the same space requirements ($n \log \sigma(1+o(1))$ bits), yet its performance is very close to that of the regular WT with pointers. Figure 1.(right) shows an example.

As in the WT, the i-th level stores the i-th bits of the encoded symbols. A single bitvector B_i is kept for each level. In the first level, B_1 stores the 1-st bit of the encoding of the symbols in the order of the original sequence S. From there on, at level i, symbols are reordered according to the $(i-1)$-th bit of their encoding; that is, according to the bit they had in the previous level. Those symbols whose encoding had a zero at position $i-1$ must be arranged before those that had a one. After that, the relative order from the previous level is maintained. That is, if a symbol α occurred before other symbol β, and the $(i-1)$-th bit of their encoding coincides, then α will precede β at level i.

If we simply keep the number of zeros at each level ($z_l \leftarrow rank_0(B_l, n)$), we can easily see that the k-th zero at level $i-1$ is mapped at position k within B_i, whereas the j-th one at level $i-1$ is mapped at position $z_l + j$ within B_i. This avoids the need for pointers and permits to retain the same time complexity of the WT operations. For implementation details see [38, 58]. For example to solve $access(S, 8)$, we see that $B_1[8] = \mathbf{0}$ and $rank_0(B_1, 8) = 5$. We move to the next level where we check position 5; we see that $B_2[5] = \mathbf{1}$ and $rank_1(B_2, 5) = 3$. We move to next level and check position $3 + z_2 = 3 + 5 = 8$, where we finally see $B_3[8] = \mathbf{1}$. Therefore, we have decoded the bits **011** that correspond to symbol $3 = access(S, 8)$.

To reduce the space needs of WM we could use compressed bitvectors as for WTs. Space needs become $nH_0(S) + o(n \log \sigma)$ bits. Yet, compressing the WM by giving either a Huffman or Hu-Tucker shape is not possible as the reordering of the WM could lead to the existence of holes in the structure that would ruin the process of tracking symbols during traversals. To overcome this issue an optimal Huffman-based coding was specifically developed for wavelet matrices [38, 59]. This allows to obtain space similar to that of a pointerless Huffman-shaped WT but faster *rank*, *select*, and *access* operations. Unfortunately, since the encodings of consecutive symbols do not form a contiguous range, $count(i, j, \alpha, \beta)$ is no longer supported in $O(\log \sigma)$ time and computing $rank_c(S, j) - rank_c(S, i) + 1$ is required for each c in $[\alpha, \beta]$.

As indicated before, since in CTR we need efficient support for *count* operation, we will try (see Section 6) the Hu-Tucker-shaped WT as well as the

or [55, 56]).

[4] In a pointerless Huffman-shaped WT a term $O(\sigma \log \log n)$ still remains due to the need of storing the canonical Huffman model.

uncompressed WM. In addition, we will couple them with both uncompressed and RRR compressed bitvectors.

3. Counting-based queries

In transportation systems, new technologies such as automatic fare collection (e.g., smartcards) and automatic passenger counting have made possible to generate a huge amount of highly detailed, real-time data useful to define measures that characterize a transportation network. This data is particularly useful because it actually consists of real trips, combining implicitly the service offered by a public transportation system with the demand for the system. When having this data, it is not the data about individual trajectories but measures of the use of the network what matters for traffic monitoring and road planning tasks. Examples of useful measures are accessibility and centrality indicators, referred to how easy is to reach locations or how important certain stops are within a network [60–62]. All these measures are based on some kind of counting queries that determine the number of *distinct* trips that occur within a spatial and/or temporal window.

Among other types of queries, in this work we focus on the following counting queries, which to the best of our knowledge have not been addressed by previous proposals. In general terms, we define two general queries, number-of-trips queries and top-k queries, upon which we apply spatial, temporal or spatio-temporal constraint when useful.

(a) *Number-of-trips queries.* This is a general type of queries that counts the number of distinct trips. When applying spatial, temporal or spatio-temporal constraints, it can specialized in the following queries:

1. Pure spatial queries:
   - *Number of trips starting at node X (`starts-with-x`).*
   - *Number of trips ending at node X (`ends-with-x`).*
   - *Number of trips starting at X and ending at Y (`from-x-to-y`).*
   - *Number of trips using or passing through node X (`uses-x`)*
2. Spatio-temporal queries:
   - *Number of trips starting at node X during time interval $[t_1, t_2]$ (`starts-with-x`).*
   - *Number of trips ending at node X during the time interval $[t_1, t_2]$ (`ends-with-x`).*
   - *Number of trips starting at X and ending at Y occurring during time interval $[t_1, t_2]$ (`from-x-to-y`).* This type of queries is further classified into: (i) `from-x-to-y` with strong semantics (`from-x-to-y-strong`), which considers trips that completely occur within interval $[t_1, t_2]$. (ii) `from-x-to-y` with weak semantics (`from-x-to-y-weak`), which considers trips whose life time overlap $[t_1, t_2]$.
   - *Number of trips using node X during the time interval $[t_1, t_2]$ (`uses-x`).*
3. Pure temporal queries:
   - *Number of trips starting during the time interval $[t_1, t_2]$ (`starts-t`).*
   - *Total usage of network stops during the time interval $[t_1, t_2]$ (`uses-t`).*
   - *Number of trips performed during the time interval $[t_1, t_2]$ (`trips-t`).*

(b) *Top-k queries.* In this type of queries we want to retrieve the k nodes with the highest number of trips. In this case, depending on having a temporal constraint or not we include the following queries:

1. Pure spatial *Top-k* queries:
   - *Top-k most used nodes (`top-k`)* that returns the nodes with the largest number of trips passing through.
   - *Top-k most used nodes to start a trip (`top-k-starts`)* that returns the nodes with the largest number of trips that start at that node.

2. Spatio-temporal *Top-k* queries:
   - *Top-k most used nodes during time interval* $[t_1, t_2]$*(`top-k`)* that returns the nodes with the largest number of trips passing through within time interval $[t_1, t_2]$.
   - *Top-k most used nodes to start a trip during time interval* $[t_1, t_2]$*(`top-k-starts`)* that returns the nodes with the largest number of trips starting there within time interval $[t_1, t_2]$ at that node.

4. Compact Trip Representation (CTR)

If we consider a network $\mathcal{N}$ with σ_s nodes, we can see a dataset of trips $\mathcal{T}$ over $\mathcal{N}$ as a set of z trips, where for each trip $\mathcal{T}_i$, we represent a list with the l_i temporary-ordered nodes it traverses and the corresponding timestamps: $\mathcal{T} = \{\langle(s_1^i, s_2^i, \ldots, s_{l_i}^i), (t_1^i, t_2^i, \ldots, t_{l_i}^i)\rangle\}$, $i \in [1, z]$, $s_j^i \in [1, \sigma_s]$, and $t_x^i \leq t_y^i, \forall x < y$. Note that every node in the network can be identified with an integer ID (s_1^i) and that, if we are interested in analyzing the usage patterns of the network, we will probably be interested in discretizing time into time intervals (i.e. 5-min, 30-min intervals). Therefore, we will have σ_t different time intervals that can also be identified with an integer ID ($t_j^i \in [0, \sigma_t)$).

The size of the time interval is a parameter for the time-discretizing process that can be adjusted to fit the required precision in each domain. For example, in a public transportation network where we could have data including five years of trips, one possibility would be to divide that five-years period into 10-minute intervals hence obtaining a vocabulary of roughly $\sigma_t = 5 \times 365 \times 24 \times 60/10 = 262,800$ different intervals. Other possibility would be to use cyclically annual 10-minute periods resulting in $\sigma_t = 262,800/5 = 52,560$. However, in public transportation networks, queries such as *"Number of trips using the stop X on May 10 between 9:15 and 10:00"* may be not as useful as queries such as *"Number of trips using stop X on Sundays between 9:15 and 10:00"*. For this reason, CTR can adapt how the time component is encoded depending on the queries that the system must answer.

Example 4.1. Figure 2 shows a network that contains $\sigma_s = 10$ nodes numbered from 1 to 10. Over that network we have six trips ($z = 6$), and, for each of them, we indicate the sequence of nodes it traverses and the time when the trip goes through those nodes. If we discretize time into 5-minute intervals, starting at 08:00h, and ending at 9:20h, we will have have $\sigma_t = 16$ different time intervals. Any timestamp within interval $[08:00, 08:05)$ will be assigned time-code 0, those within $[08:05, 08:10)$ code 1, and so on until times

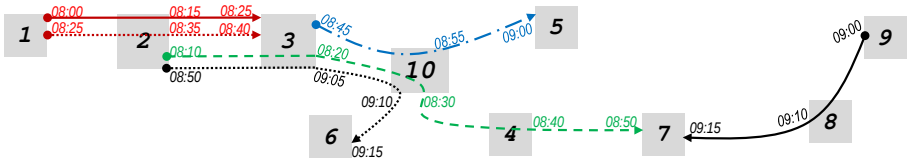

Figure 2: A set of trips over a network with 10 nodes.

within $[09:15, 09:20)$ that are given time-code 15. Therefore, our dataset of trips will be: $\mathcal{T}$: $\{\langle(\mathbf{1},\mathbf{2},\mathbf{3}), (5,7,8)\rangle, \langle(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{6}), (10,13,14,15)\rangle, \langle(\mathbf{1},\mathbf{2},\mathbf{3}), (0,3,5)\rangle, \langle(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7}), (2,4,6,8,10)\rangle, \langle(\mathbf{3},\mathbf{10},\mathbf{5}), (9,11,12)\rangle, \langle(\mathbf{9},\mathbf{8},\mathbf{7}), (12,14,15)\rangle\}$, where bold numbers indicate node IDs and slanted ones indicate times. □

In CTR we represent both the spatial and the temporal component of the trips using well-known self-indexing structures in order to provide both a compact representation and the ability to perform fast indexed searches at query time. In Section 5 we focus on the spatial component and discuss how we adapted CSA to deal with trips. We also show how we support spatial queries. Then, in Section 6 we show that the times, which are kept aligned with the spatial component of the trips, can be handled with a WT-based representation. Actually we study two alternatives (a WTHT and a WM) and show how temporal and spatio-temporal (Section 7) queries are supported by CTR.

5. Spatial component of CTR

We use a CSA to represent the spatial component of our dataset of trips within CTR. Yet, we perform some preprocessing on $\mathcal{T}$ before building a CSA on it. Initially, we sort the trips by their first node (s_1^i), then by the last node ($s_{l_i}^i$), then by the starting time (t_1^i), and finally, by its second node (s_2^i), third node (s_3^i), and successive nodes ($s_j^i, 3 < j \leq l_i$). Note that the start time (t_1^i) of the trip does not belong to the spatial component, but it is nevertheless used for the sorting.[5]

Following with Example 4.1, after sorting the trips in $\mathcal{T}$ with the criteria above, our sorted dataset $\mathcal{T}^s$ would look like: $\mathcal{T}^s$: $\{\langle(\mathbf{1},\mathbf{2},\mathbf{3}), (0,3,5)\rangle, \langle(\mathbf{1},\mathbf{2},\mathbf{3}), (5,7,8)\rangle, \langle(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{6}), (10,13,14,15)\rangle, \langle(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7}), (2,4,6,8,10)\rangle, \langle(\mathbf{3},\mathbf{10},\mathbf{5}), (9,11,12)\rangle, \langle(\mathbf{9},\mathbf{8},\mathbf{7}), (12,14,15)\rangle\}$. Note that $(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{6})$ appears before $(\mathbf{2},\mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7})$ because during the sorting process we compare $(\mathbf{2},\mathbf{6},2,\mathbf{3},\mathbf{10},\mathbf{6})$ with $(\mathbf{2},\mathbf{7},10,\mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7})$; that is, we compare the starting nodes ($\mathbf{2}$ and $\mathbf{2}$) and then the ending nodes ($\mathbf{6}$ and $\mathbf{7}$). If needed (not in this example) we would have also compared the slanted values (2 and 10) that are the starting times of the trips, and finally the rest of nodes ($\mathbf{3},\mathbf{10},\mathbf{6}$ and $\mathbf{3},\mathbf{10},\mathbf{4},\mathbf{7}$). Similarly, the two trips containing nodes $(\mathbf{1},\mathbf{2},\mathbf{3})$ are sorted by the starting times (0 and 5).

In a second step, we enlarge all the trips $\mathcal{T}_i^s, 1 \leq i < z$ with a fictitious terminator-node $\$_i$ whose timestamp is set to that of the initial node of the

[5] This initial sorting of the trips will allow us to answer some useful queries very efficiently (i.e., count trips starting at X and ending at Y).

trip. We choose terminators such that $\$_i \prec \$_j, \forall i < j$; that is the lexicographic value of $\$_i$ is smaller for smaller i values. In addition, the lexicographic value of any terminator must be lower than the ID of any node in a trip. Therefore, an enlarged trip $\mathcal{T}_i^s$ would become $\mathcal{T}_i' = \langle (s_1^i, s_2^i, \ldots, s_{l_i}^i, \$_\mathbf{i}), (t_1^i, t_2^i, \ldots, t_{l_i}^i, \mathbf{t_1^i}) \rangle$.

The next step involves concatenating the spatial component of all the enlarged trips and to add an extra trailing terminator $\$_0$ to create a sequence $S[1, n]$. $\$_0$ must be lexicographically smaller than any other entry in S (then it also holds $\$_0 \prec \$_i \ \forall i \in [1, z]$). In the top part of Figure 3, we can see array S for the running example, as well as the corresponding time-IDs that are regarded in sequence $Icode$ (I shows the original times).

Finally, we build a CSA on top of S to obtain a self-indexed representation of the spatial component in CTR. Figure 3 depicts the structures Ψ and D used by CTR built over S. There is also a vocabulary V containing a \$ symbol and the different node IDs in lexicographic order.

Note that the use of different values $\$_i$ as terminators ensures that our sorting criteria are kept even if we follow the standard suffix-sort procedure[6] required to build suffix array A during the creation of CSA. Yet, when we finish that process, we can replace all those $\$_i$ terminators by a unique \$. This is the reason why there is only one \$ symbol in V.

Although they are not needed in CTR, we show also suffix array A and Ψ' for clarity reasons in Figure 3. Ψ' contains the first entries of Ψ from a regular CSA, whereas we introduced a small variation in CTR for entries $\Psi[1, z+1]$. For example, $A[8] = 1$ points to the first node of the first trip $S[1]$. $\Psi[8] = 10$ and $A[10] = 2$ point to the second node. $\Psi[10] = 14$ and $A[14] = 3$ point to the third node. $\Psi[14] = 2$ and $A[2] = 4$ point to the ending $\$_1$ of the first trip. Therefore, in the standard CSA, $\Psi'[2] = 9$ and $A[9] = 5$ point to the first node of the second trip. However, in CTR, $\Psi[2] = 8$ and $A[8] = 1$ point to the first node of the first trip. With this small change, subsequent applications of Ψ will allow us to cyclically traverse the nodes of the trip instead of accessing the following entries of S.

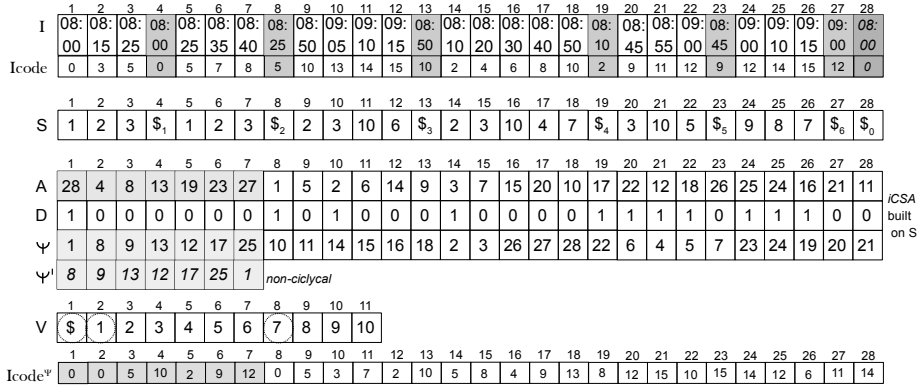

Figure 3: Structures involved in the creation of a CTR.

Another interesting property arises from the use of a cyclical Ψ on trips, and from using trip terminators. Since the first entries in $\Psi[2, z+1]$ correspond

[6]Suffix $S[i, n]$ is compared with suffix $S[j, n]$.

the \$ symbols that mark the end of each trip in S (remember that $\Psi[1]$ corresponds the $\$_0$), we can see that the j^{th} node of the i^{th} trip can be obtained as $V[rank_1(D, \Psi^j[i+1])]$, (where $\Psi^3[x] = \Psi[\Psi[\Psi[x]]]$). This property makes it very simple to find starting nodes for any trip. For example, if we focus on the shaded area $\Psi[2, 7]$, we can find the ending terminator $\$_4$ of the fourth trip at the 5^{th} position (because the first $\$_0$ corresponds to the final \$ at $S[28]$). Therefore, its starting node can be found as $V[rank_1(D, \Psi[4+1])]$. Since $\Psi[5] = 12$ and $rank_1(D, 12) = 3$, the starting node is $V[3] = \mathbf{2}$. For illustration purposes note that it would correspond to $S[A[12]]$. By applying Ψ again, the next node of that trip would be obtained by computing $\Psi[12] = 16$, $rank_1(D, 16) = 4$, and accessing $V[4] = \mathbf{3}$ (that is, we have obtained $V[rank_1(D, \Psi[\Psi[4+1]])] = \mathbf{3}$, and so on.

Regarding the space requirements of the CSA in CTR, we can expect to obtain a good compressibility due to the structure of the network, and the fact that trips that start in a given node or simply those going through that node will probably share the same sequence of "next" nodes. This will lead us to obtaining many *runs* in Ψ [43], and consequently, good compression.

5.1. Dealing with Spatial Queries

With the structure described for representing the spatial component of the trips, the following queries can be solved.

- *Number of trips starting at node X (`starts-with-x`).* Because Ψ was cyclically built in such a way that every \$ symbol is followed by the first node of its trip, this query is solved by $[l, r] \leftarrow bsearch(\$X)$ over the CSA, which results on a binary search for the pattern $\$X$ over the section $\Psi[2, z+1]$ corresponding to \$ symbols. Then $r - l + 1$ gives the number of trips starting at X.

- *Number of trips ending at node X (`ends-with-x`).* In a similar way to the previous query, this one can be answered with $bsearch(X\$)$.

- *Number of trips starting at X and ending at Y (`from-x-to-y`).* Combining both ideas from above, and thanks to the cyclical construction of Ψ, this query is solved using $bsearch(Y\$X)$.

- *Number of trips using node X (`uses-x`).* Even though we could solve this query with $bsearch(X)$, it is more efficient to solve it by directly operating on D. Assuming that X is at position p in the vocabulary V of CTR ($V[p] = X$), its total frequency is obtained by $occs_X \leftarrow select_1(D, p+1) - select_1(D, p)$. If p is the last entry in V, we set $occs_X \leftarrow n + 1 - select_1(D, p)$.

- *Top-k most used nodes (`top-k`).* We provide two possible solutions for this query named: sequential and binary-partition approaches.

  - To return the k most used nodes using *sequential approach (`top-k-seq`).* The idea is to apply $select_1$ operation sequentially for every node from 2 to $|V|$ to compute the frequency of each node and to return the k nodes with highest frequency. We use a min-heap that is initialized with the first k nodes, and for every node s from $k+1$ to $|V|$, we compare its frequency with that of the minimum node (the root) from the heap. In

case the frequency of s is higher, the root of the heap is replaced by s and then moved down to comply with the heap ordering. At the end of the process, the heap will contain the top-k most used nodes $\langle p_1, p_2, \ldots, p_k \rangle$, which can be sorted with the heapsort algorithm if needed. Finally, we return $\langle V[p_1], V[p_2], \ldots, V[p_k] \rangle$. Note that this approach always performs $|V|$ $select_1$ operations on D.

- The *binary-partition (`top-k-bin`)* approach takes advantage of a skewed distribution of frequency of the nodes that trips traverse. Working over D and V, we recursively split D into two segments after each iteration. If possible, we leave the same number of different nodes in each side of the partition. Initially, we start considering the range in $D[l, r] \leftarrow D[select_1(D, 2), n] = D[z + 2, n]$ which corresponds to the nodes that appear in V from positions $i = 2$ to $j = |V|$.[7] We use a priority queue that is initialized as $Q \leftarrow (\langle i, j \rangle, \langle l, r \rangle)$. Then, assuming $m = i + \frac{j-i+1}{2}$ and $q = select_1(D, m)$, we create two partitions $D[l, q-1]$ and $D[q, r]$, which correspond respectively to the nodes in $V[i, m-1]$ and $V[m, j]$. These segments created after the partitioning step are pushed into Q. The pseudocode can be found in Figure 4.

  The priority of each segment in Q is directly the size of its range in D $(r - l + 1)$. When a segment extracted from Q represents the instance of only one node $((\langle i, j \rangle, \langle l, r \rangle)$, with $i = j)$, that node is returned as a result of the top-k algorithm (we return $V[i]$). The algorithm stops when the first k nodes are found.

  For example, when searching for the top-1 most used nodes in the example from Figure 3, Q is initialized with the segment $[8, 28]$, corresponding to nodes from 1 to 10 (positions from 2 to 11 in V). Note that the entries of D from 1 to 7 and $V[1]$ represent the \$ symbol. Since it is not an actual node, it must be skipped. Then $[8, 28]$ is split producing the segments $[8, 20]$ for nodes 1 to 5 ($V[2, 6]$) and $[21, 28]$ for nodes 6 to 10 ($V[7, 11]$). After three more iterations, we extract $(\langle 3, 3 \rangle, \langle 14, 18 \rangle)$, hence obtaining the segment $[14, 18]$ for the single node 3 (position 4 in V), concluding that the *Top-1 most used node* is $\mathbf{3} = V[4]$ with a frequency equal to $5 = 18 - 14$.

- *Top-k most used nodes to start a trip (`top-k-starts`).* Both `top-k` approaches above can be adapted for answering `top-k-starts`. However, unlike its simpler variant, it requires performing $bsearch(\$X)$ over Ψ (rather than a *select* on D) at each iteration, hence increasing the temporal complexity of the operation.

  The implementation of the linear approach is straightforward. The binary-partition approach differs slightly from the algorithm in Figure 4: in (l.2) we insert $(\langle 2, |V| \rangle, \langle 2, z+1 \rangle)$ into Q, and we replace (l.11) with $[x, y] \leftarrow bsearch(\$V[m])$; $q \leftarrow x$.

[7] We skip the \$ at the first entry of V and its corresponding entries in D; that is, $D[1, select_1(D, 2) - 1]$.

```
GetTopK_most_used_nodes (k):
(l.1 ) Q ← new PriorityQueue();
(l.2 ) Q.push(⟨2, |V|⟩, ⟨select_1(D, 2), n⟩);
(l.3 ) current_k ← 0;
(l.4 ) while current_k < k:
(l.5 )     (⟨i, j⟩, ⟨l, r⟩) ← Q.pop();
(l.6 )     if i = j:
(l.7 )         topK[current_k] ← V[i];
(l.8 )         current_k ← current_k + 1;
(l.9 )     else:
(l.10)         m ← i + (j−i+1)/2;
(l.11)         q ← select_1(D, m + 1);
(l.12)         Q.push(⟨i, m − 1⟩, ⟨l, q − 1⟩);
(l.13)         Q.push(⟨m, j⟩, ⟨q, r⟩);
(l.14) return topK;
```

Figure 4: Algorithm *Top-k most used nodes* using binary-partition approach.

5.2. Implementation details

In our implementation of CSA, we used the iCSA[8] from [44] briefly discussed in Section 2.3.1. Yet, we introduced some small modifications:

- The construction of the Suffix Array A is done with *SA-IS* algorithm [63].[9] In comparison with the *qsufsort* algorithm[10] [64] used in the original iCSA, it achieves a linear time construction and a lower extra working space.
- In iCSA, they used a plain representation for bitvector D and additional structures to support $rank_1$ in constant time using $(0.375 \times n$ bits). With that structure, they could solve *select* in $O(\log n)$ time (yet they did not actually needed solving *select* in iCSA). In our CSA, we have used the *SDArray* from [65] to represent D. It provides a very good compression for sparse bitvectors, as well as constant-time *select* operation.
- In [44], *bsearch* operation was implemented with a simple binary search over Ψ rather than using the backward-search optimization proposed in the original CSA [15]. In our experiments, we used backward search since it led to a much lower performance degradation at query time when a sparse sampling of Ψ was used.

6. Temporal component of CTR

In this section we focus on the temporal component associated with each node of the enlarged trips $\mathcal{T}'_i$ in our dataset. Recall that in Figure 3, sequence I contains the time associated with each node in a trip, and $Icode$ a possible encoding of times. In CTR we focus on the values in $Icode$, yet, since S is not kept anymore in CTR, we reorganize the values in $Icode$ to keep them

[8] http://vios.dc.fi.udc.es/indexing
[9] https://sites.google.com/site/yuta256/sais
[10] http://www.larsson.dogma.net/research.html

aligned with Ψ rather than with S. Those values are represented within array $Icode^{\Psi}$ in Figure 3. For example, we can see that $Icode^{\Psi}[4]$ corresponds with $Icode[A[4]] = 10$, $Icode^{\Psi}[15]$ corresponds with $Icode[A[15]] = 8$, and so on.

Aiming at having a compact representation of $Icode^{\Psi}$ while permitting fast access and resolution of range-based queries (that we could use to search for trips within a given time interval), we have considered two WT-based alternatives from the ones presented in Section 2.3.2:

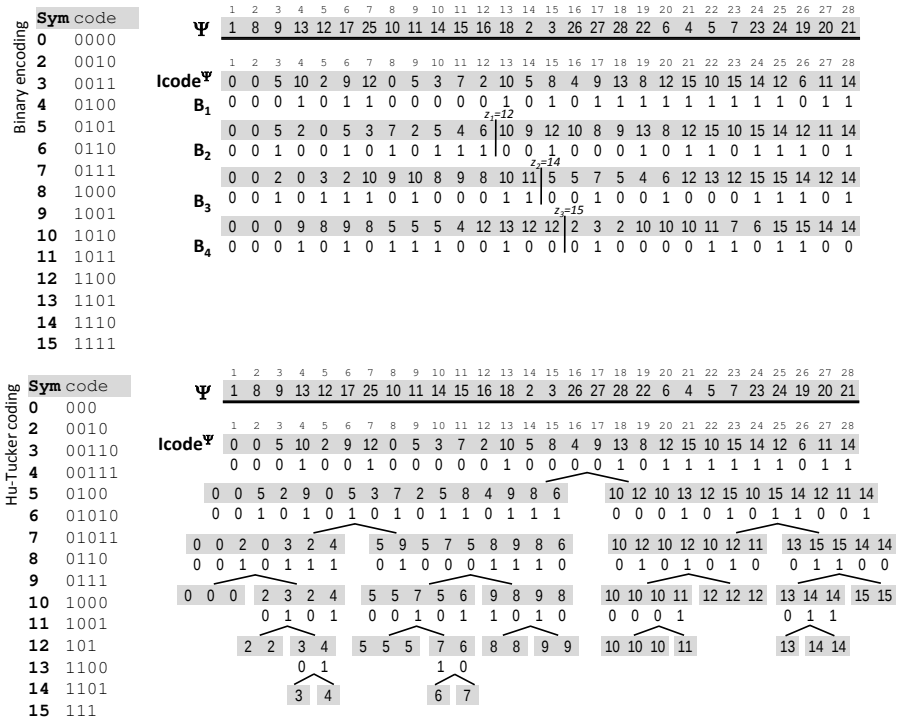

Figure 5: Balanced WM (top) and Hu-Tucker-shaped WT (bottom) for the times within $Icode^{\Psi}$ in Figure 3.

- A Wavelet Tree [16] using variable-length Hu-Tucker codes [53] (WTHT). Recall this is the WT variant that permits to compress the original symbols with variable-length codes and still supports *count* operation in $O(log(\sigma_t))$ time. Since Hu-Tucker coding assigns shorter codes to the most frequent symbols, the compression of our WTHT is highly dependent of the distribution of frequencies for the $Icode^{\Psi}$. Yet, if our trips represent movements of single users in a transportation network, we could expect to observe two or more periods corresponding to rush hours within a single day. This would lead to obtaining a skewed distribution of the frequencies for the symbols in $Icode^{\Psi}$, and consequently, we could expect to have better compression than if we used a balanced WT. The expected number of bits of our WTHT is $nH_0(Icode^{\Psi})$.

- A balanced Wavelet Matrix (WM) [38]. As we showed in Section 2.3.2 the WM is typically the most compact uncompressed variant of WT and it is faster than a pointerless WT. This is the reason why we chose a balanced

WM instead of a balanced WT as this second alternative. Recall that, $Icode^{\Psi}$ contains n symbols, and each symbol can be encoded with $\log \sigma_t$ bits, hence the balanced WM will be a matrix of $n \log \sigma_t$ bits.

In Figure 5, we show both the WM and the WTHT built on top of $Icode^{\Psi}$ from Figure 3. The binary code-assignment to the source symbols in $[1, \sigma_t]$ and that obtained after applying Hu-Tucker encoding algorithm [53] are also included in the figure.

6.1. Dealing with Temporal queries

With either one of the described alternatives (WTHT or WM) to represent time intervals we can answer the following pure temporal queries:

- *Number of trips starting during the time interval* $[t_1, t_2]$ *(`starts-t`).* Since we keep the starting time of each trip within $Icode^{\Psi}[1, z]$, we can efficiently solve this query by simply computing $count(1, z, t_1, t_2)$.
- *Total usage of network stops during the time interval* $[t_1, t_2]$ *(`uses-t`).* This query can be seem as the sum of the number of trips that traversed each network node during $[t_1, t_2]$. We can solve this query by computing $count(z + 1, n, t_1, t_2)$.
- *Number of trips performed during the time interval* $[t_1, t_2]$ *(`trips-t`).* This is also an interesting query that permits to know the actual network usage during a time interval. To solve this query we could compute `trips-t` by subtracting the number of trips that started after t_2 (`starts-t`$(t_2 + 1, \sigma^t - 1)$) and the number of trips that ended before t_1 (`ends-t`$(0, t_1 - 1)$) from the total number of trips (z). However, recall that $Icode^{\Psi}[1, z]$ has the starting time of each trip, but we do not keep their ending time. We could solve `ends-t`$(0, t_1 - 1)$ by taking the first node (X) of each trip starting before t_1, then applying Ψ until reaching the ending node (Y), and finally getting the ending time of that trip associated to node Y. However, this would be rather inefficient. A possible solution to efficiently solve `ends-t`$(0, t_1 - 1)$, would require to increment our temporal component, in parallel with $Icode^{\Psi}[1, z]$, with another WT-based representation of the ending times for our z trips. This would permit to report the number of trips ending before t_1 as $count'(1, z, 0, t_1 - 1)$, but would increase the overall size of CTR. Yet, note that even without keeping ending-times, we could provide rather accurate estimations of `trips-t` for a system administrator. For example, using `uses-t` to compute the number of times each trip went through any node during the time interval $[t_1, t_2]$, and dividing that value by the average nodes per trip. Another good estimation can also be obtained with `starts-t`(t_1, t_2).

6.2. Implementation details

We include here details regarding how we tune our WTHT and WM. As we discussed in Section 2.3.2, both WTHT and WM are built over bitvectors that require support for *rank* and *select* operations. In our implementations we included two alternative bitvector representations avaliable at *libcds* library:[11]

[11] https://github.com/fclaude/libcds

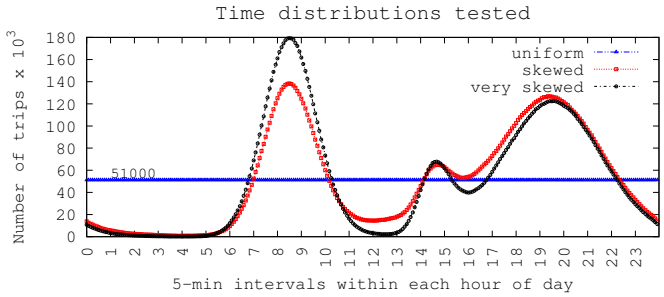

Figure 6: Time distributions used. The y-axis indicates the number of passengers per each 5-min interval.

- A plain bitvector based on [42] named *RG* with additional structures to support *rank* in constant time (*select* in logaritmic time). *RG* includes a sampling parameter (*factor*) that we set to value 32. In this case, our bitvector *RG* uses $n(1 + 1/32)$ bits. That is, we tune *RG* to use a sparse sampling.

- A compressed RRR bitvector [49]. The *RRR* implementation includes a sampling parameter that we tune to values 32, 64, and 128. Higher sampling values achieve better compression.

In advance, when presenting results for WTHT and WM we will consider the four bitvector configurations above. Regarding our implementations of WM and WTHT, note that we reused the same implementation of WM from [38], and we created our custom WTHT implementation, paying special focus at solving *count* efficiently.

6.3. Comparing the space/time trade-off of WM *and* WTHT

In order to compare the efficiency of our WTHT (that uses variable-length codes and supports *count* efficiently) with a balanced WM alternative under different time distributions (recall that this WM is time distribution invariant), we run some experiments that evaluate the average time to execute *count* operation on both representations.

We used a dataset of generated trips (Refer to Section 8.1 for the details about Madrid dataset) and we generated three kinds of time distributions for our evaluation. We refer to them as: uniform, skewed and very skewed. They are shown in Figure 6. According to the total number of passengers in a day, in the uniform distribution 51,000 passengers use the network for each 5-minute interval. We also generated a skewed distribution for the time interval frequencies in an effort to model the usage of a public transportation network in a regular working day, where the starting time of a trip is generated according to the following rules:

- With 30% of probability, a trip occurs during a morning rush hour.

- With 45% of probability, a trip occurs in an evening rush hour.

- With 5% of probability, a trip occurs during lunch rush hour.

- The remaining 20% of probability is associated to unclassified trips, starting at a random hour of the day, which may also fall into one of the three previous periods discussed.

In the very skewed distribution we increase the rush-hour probabilities with 40% for the morning rush hour, 50% for the evening rush hour, 8% for lunch period and only 2% of random movements.

Then we built the WTHT and the WM considering two different granularities for the discretization of times: five-minute and thirty-minute intervals. Then, we generated 10,000 random intervals of times $[t_1, t_2]$ over the whole time sequence of the dataset considering interval widths of five minutes, one hour, and six hours. Finally, we run 10,000 $count(z + 2, n, t_1, t_2)$ queries (we show average times) from each query set over the six configurations of WTHT and WM (2 different granularities for the time discretization and 3 datasets).

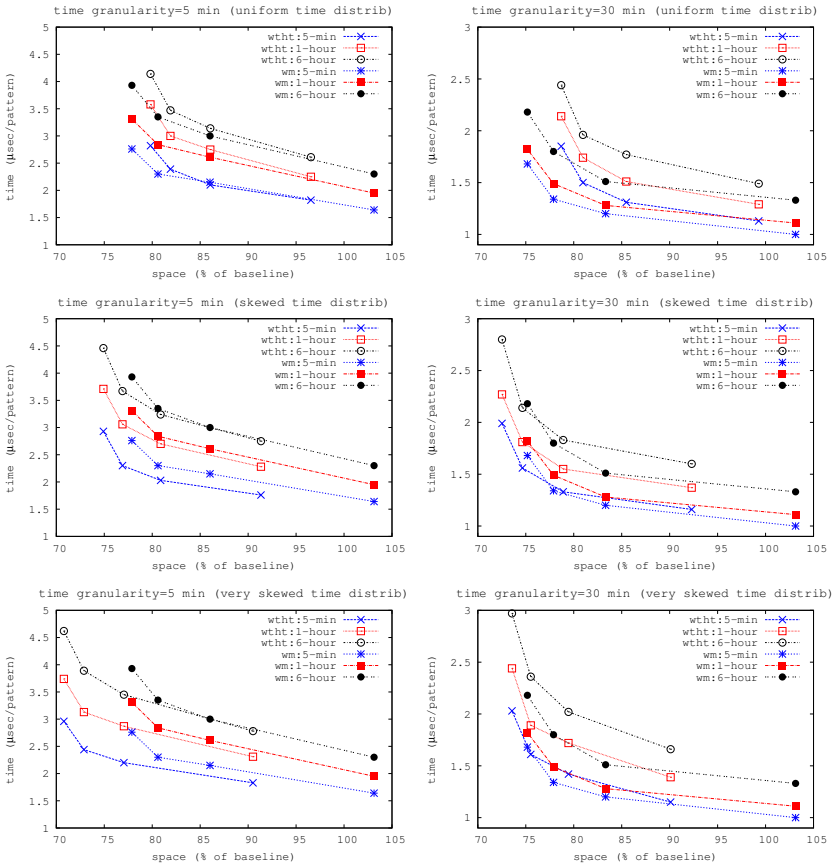

Figure 7: Space/time trade-offs for *count* queries depending on the time distribution: uniform (top), skewed (middle), and very skewed (bottom). Time granularity for the time index is 5 minutes (left) or 30 minutes (right).

In Figure 7, we show the results of our experiments. In the upper part of the figure, we include the results for WTHT and WM built over the times assuming uniform frequency distribution. In the middle part we assume times follow a

the skewed distribution, and in the bottom of the figure we show results when considering a very skewed distribution. Moreover, figures in the left column show results for our structures considering that a 5-minute granularity is chosen for the discretization of times, whereas figures on the right column assume time granularity is 30 minutes. For each scenario we include plots `wtht:5-min`, `wtht:1-hour`, and `wtht:6-hour` for WTHT (range width for *count* is respectively 5-minutes, 1-hour, and 6-hours). We also present those plots for WM (`wm:5-min`, `wm:1-hour`, and `wm:6-hour`).

The baseline used for the space usage (x-axis) is the size of an array of fixed-length time-interval IDs represented with the least number of bits needed (12 bits and 9 bits respectively for 5-minute and 30-minute granularity, see Section 8.1).

When times are uniformly distributed, our WTHT can only exploit the redundancy introduced by the \$ symbols. This fact permits WTHT to obtain only a minimal compression (around $96-98\%$ of the baseline) when using a RG (plain) bitvector, whereas WM uses more space than the baseline (around 104%). Recall that for each plot we present four points corresponding (left-to-right) to RRR_{128}, RRR_{64}, RRR_{32}, and RG bitvectors. When using compressed bitvectors (RRR), WM becomes the best choice. It is both more compact (bitvectors in WM are more compressible) and faster than WTHT. In any case, using RRR clearly slows down queries.

A skewed distribution favors the compression for a statistical coder like Hu-Tucker, which explains the higher compression obtained. However, it also slightly increases the query times, especially in the wider one-hour and six-hours query sets. This happens because the probability of having a query that forces to descend completely up to the leaves of the WTHT increases.

For a very skewed distribution, the gap in compression between WTHT and WM increases clearly (around 5 percentage points), whereas query times remain similar to those in the previous scenario.

As a conclusion of the experiments discussed in this section, we have shown that the distribution of the sequence of times can be exploited by our WTHT to achieve a better compression and even improved query times than the balanced WM counterpart.

7. Dealing with Spatio-temporal queries

Apart from the pure spatial and temporal queries discussed in the previous sections, we can combine both the self-indexed spatial and temporal components from CTR to answer spatio-temporal queries. The idea is to restrict spatial queries to a time interval $[t_1, t_2]$. An example of this type of query is to return the *number of trips starting at node X that occurred between t_1 and t_2*, which we can solve by first finding the range in the CSA of the trips starting in X and then relying on the *count* operation in the WTHT (or WM). The following spatio-temporal queries can be solved by CTR:

- *Number of trips starting at node X during time interval $[t_1, t_2]$ (`starts-with-x`).* Recall that in the time sequence we also included timestamps associated with the area of \$-symbols in Ψ. Particularly, for each \$, we keep the time of the first node of its trip. Therefore, we can perform $[l, r] \leftarrow bsearch(\$X)$ as in

a regular spatial query to find the range $\Psi[l, r]$ ($[l, r] \subseteq [2, z+1]$) that corresponds to \$ symbols that end a trip which started at node X. Then, since the time sequence $Icode^{\Psi}$ (represented with either a WTHT or WM) is aligned with Ψ, we can filter out those trips that started within $[t_1, t_2]$ performing operation $count(l, r, t_1, t_2)$. In Figure 8 (steps ① and ②) we can see the steps involved.

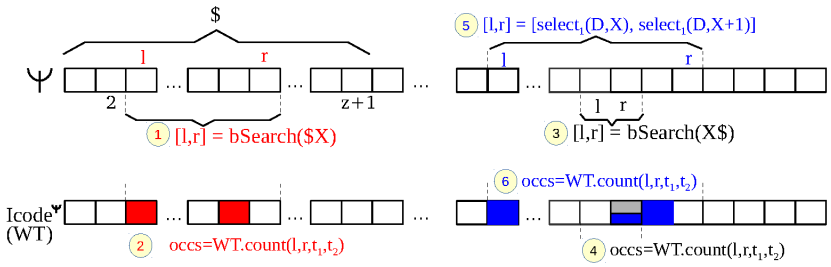

Figure 8: Trips staring at, ending at, or using node X during time interval $[t_1, t_2]$.

- *Number of trips ending at node X during the time interval $[t_1, t_2]$ (`ends-with-x`).* As above, we initially perform the spatial query $[l, r] \leftarrow bsearch(X\$)$ to obtain the range in Ψ that corresponds to the pattern $X\$$ (trips ending at node X). Then, we use $count(l, r, t_1, t_2)$ operation to count how many of those trips match the temporal constraint. See steps ③ and ④ in Figure 8.

- *Number of trips using node X during the time interval $[t_1, t_2]$ (`uses-x`).* As in the corresponding spatial query, the range $[l, r]$ in Ψ is obtained with two $select_1$ operations on D. Finally, $count(l, r, t_1, t_2)$ finds the occurrences within the time interval $[t_1, t_2]$. See steps ⑤ and ⑥ in Figure 8.

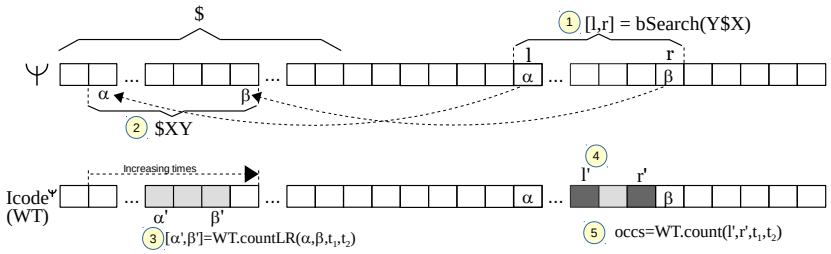

Figure 9: Trips staring at X and ending at Y during time interval $[t_1, t_2]$.

- *Number of trips starting at X and ending at Y occurring during time interval $[t_1, t_2]$ (`from-x-to-y`).* We consider two different semantics. A query with *strong semantics* will obtain trips that start and end within $[t_1, t_2]$. Whereas, a query with *weak semantics* will obtain trips whose time intervals overlap $[t_1, t_2]$ and, therefore, they could actually start before t_1 or end after t_2.

  In Figure 9, we show the step-by-step process to solve this type of queries. As in a spatial query, we start by searching for the range $[l, r] \leftarrow bsearch(Y\$X)$ in Ψ corresponding to trips starting at Y and ending at X (*step-*①). Next, due to our sorting of trips, the range for $Y\$X$ in $\Psi[l, r]$ can be mapped to a

continuous range $[\alpha, \beta]$ of the same size in the $\$XY$ region of Ψ. We compute $\alpha \leftarrow \Psi[l], \beta \leftarrow \alpha + r - l$ (*step-*②). Furthermore, note that the range for $\$XY$ preserves the same order as that for $Y\$X$.

At this point, since $Icode^{\Psi}$ was aligned with Ψ, we could check ending-time constraints within $Icode^{\Psi}[l, r]$ and starting-time constraints within $Icode^{\Psi}[\alpha, \beta]$ (recall we keep starting times associated with the corresponding \$ of each trip). Note also that, due to our sorting (by starting-node, ending-node, starting-time,...) the times in $Icode^{\Psi}[\alpha, \beta]$ are increasing (all of them correspond to trips with the same starting-node X and ending-node Y). Therefore, we can find the continuous subrange $[\alpha', \beta'] \subseteq [\alpha, \beta]$ corresponding to trips that start within $[t_1, t_2]$ (*step-*③). We refer to this operation as *countLR* in Figure 9. Thus, that assuming $Icode^{\Psi}[\alpha, \beta]$ are increasing, $[\alpha', \beta'] \leftarrow countLR(\alpha, \beta, t_1, t_2)$ would report the positions $[\alpha', \beta'] \subseteq [\alpha, \beta]$ such that $\alpha' = argmin_x(Icode^{\Psi}[x] \geq t_1)$ and $\beta' = argmax_x(Icode^{\Psi}[x] \leq t_2)$.

Using a WT, a simple way to implement *countLR* consists in performing two binary searches within $[\alpha, \beta]$ to find $[\alpha', \beta']$, where at each step we would use *access* operation. This would cost $O(\log n \log \sigma)$. Yet, we could also regard on *count* operation to obtain a more efficient and also rather straightforward implementation of *countLR* so that we set $\alpha' \leftarrow count(\alpha, \beta, -1, t_1 - 1)$ and $\beta' \leftarrow \alpha' + count(\alpha, \beta, t_1, t_2)$. It costs $O(\log \sigma)$.

- *Strong semantics (`from-x-to-y-strong`).* Note that the subrange $[\alpha', \beta']$ (containing trips starting within $[t_1, t_2]$) has a matching subrange $[l', r'] \subseteq [l, r]$ (*step-*④), where some of the ending times of these trips will fall inside $[t_1, t_2]$ (this allows us to check the ending time constraint). By performing $count(l', r', t_1, t_2]$ we get the final result (*step-*⑤). To sum up, answering this query requires: one *bsearch* over Ψ (to find $[l, r]$), one *access* to Ψ to obtain α ($\beta \leftarrow \alpha + r - l$), one *countLR* to find $[\alpha', \beta']$, and one *count* (to count the valid ending times in $[l', r']$).
- *Weak semantics (`from-x-to-y-weak`).* The size of $[\alpha', \beta']$ is already a partial answer. To get the final result, we need to add also the occurrences of those trips starting before t_1 that end at t_1 or later. To do so, if $l < l'$, we need to compute $count(l, l' - 1, t_1, \sigma_t)$. This gives us the number of time instants in the range $[l, l')$ of $Icode^{\Psi}$ that fall inside $[t_1, \sigma_t]$. That is, ending times equal or after t_1.

- *Top-k most used nodes during time interval $[t_1, t_2]$ (`top-k`).* Both the sequential and binary-partition approaches discussed in Section 5.1 can easily be extended to support this query. The idea is that, when we add a node either to the min-heap or priority-queue respectively, we compute its frequency within time interval $[t_1, t_2]$ (using *count* operation) rather than using its overall frequency.
  - In the *sequential approach (`top-k-seq`)*, given a node whose corresponding range in Ψ is $D[l, r]$, we compute its frequency using $count(l, r, t_1, t_2)$ instead of simply using $r - l + 1$. The rest of the process is exactly as discussed for the pure spatial `top-k-seq` query.
  - In the *binary-partition approach (`top-k-bin`)*, we have to consider the priority of a given segment as the number of trips covered by that segment that occurred during $[t_1, t_2]$. Again, given a segment $[l, r]$ in Ψ we

compute that priority as $p_l^r \leftarrow count(l, r, t_1, t_2)$ instead of $p_l^r \leftarrow r - l + 1$. Apart from that, the only modifications that we must consider over the pure spatial `top-k-bin` algorithm in Figure 4 are: we replace (l.2) by $p_l^r \leftarrow count(select_1(D, 2), n, t_1, t_2)$; $Q.push(\langle 2, |V| \rangle, \langle select_1(D, 2), n \rangle, \underline{p_l^r})$, and we replace (l.12) and (l.13) respectively by $Q.push(\langle i, m-1 \rangle, \langle l, q-1 \rangle, \underline{count(l, q-1, t_1, t_2)})$ and $Q.push(\langle m, j \rangle, \langle q, r \rangle, \underline{count(q, r, t_1, t_2)})$.

- *Top-k most used nodes to start a trip during time interval* $[t_1, t_2]$ *(`top-k-starts`).* Following the same guidelines discussed above for `top-k`, adapting the sequential and binary-partition solutions for the spatial `top-k-starts` to include temporal constraints is straightforward.

8. Experimental evaluation

We have run experiments to evaluate both the space requirement and performance at query time of CTR when dealing with spatial, temporal and spatio-temporal queries over two different datasets (Porto and Madrid) that are described in Section 8.1.

We have used several configurations of CTR by tuning both its spatial and temporal components. In the spatial part, we set the Ψ sampling parameter (t_Ψ) to the values $t_\Psi = \{32, 128, 512\}$. For the temporal component, we have tested both the balanced WM, and the Hu-Tucker-shaped WT (WTHT) using the same bitvector configurations discussed in Section 6.3. That is, using either a plain bitvector RG with a sparse sampling (RG_{32}), or a RRR bitvector with sampling parameter $\in \{32, 64, 128\}$ (RRR_{32}, RRR_{64}, and RRR_{128}).

8.1. Experimental datasets

We used two different datasets of trips in our experiments:

- **Madrid dataset**: Using GTFS[12] data from the public transportation network of Madrid,[13] we generated a dataset of synthetic trips combining the subway network with the Spanish commuter rail system (called *cercanías*). In total, there are 313 different stations/nodes from 23 lines.

  We generated 10 million trips with lengths varying from 2 to 31 nodes traversed. Those lengths follow a binomial distribution. The average length of the trips is 11.81 nodes.

  In the generation of a trip of length l, we randomly choose a starting node from a line, and the starting direction. Then, we follow that line until we reach a switching node. At this node, we decide whether to follow the current line or to switch to a new line. We allow only up to four line switches for a given trip, and use fixed probability values to decide whether to switch line or not. Such probability is 0.5, 0.1, 0.05, and 0.02 respectively for the first, second, third, and fourth line switch in a trip. We also avoid revisiting nodes in the same trip. Generation process ends when l nodes have been added to the trip, or a dead end is reached.

[12] GTFS is a well-known specification for representing an urban transportation network. See https://developers.google.com/transit/gtfs/reference?hl=en

[13] Data from the EMT corporation at https://www.emtmadrid.es/movilidad20/googlet.html

As a baseline, the plain representation of the generated trips using a 9-bit integer ($\lceil \log_2 314 \rceil = 9$) for every node-ID (and \$ separator) would require 137.47 MiB.

We also generated synthetic times for those trips following the same rules used to create the time distribution named *skewed* in Figure 6, so most of the trip timestamps belong to rush hours. Yet, instead of using only regular working days, we distinguished four kinds of days in a week: regular working days; Fridays and holiday eves; Saturdays; and Sundays and holidays. We also assume that there are two kinds of weeks related to high and low season periods. Therefore, a time interval may belong to eight types of day. When discretized at five-minute intervals we obtain 2,304 distinct time intervals, while when we use thirty-minute intervals we obtain 384. In the former case, our baseline for the generated times using 12 bits per time-ID would occupy 183.30 MiB. In the latter one, each time-ID requires 9 bits and the temporal baseline requires 137.47 MiB.

- **Porto dataset**: We downloaded a collection of 1,710,671 trajectories from the city of Porto corresponding to taxi trips during a full year (from July 1, 2013 to June 30, 2014), provided by [66].[14] Among other fields those data include, for each taxi ride, a list of GPS coordinates and times gathered every 15 seconds of the trip. We adapted such data to our needs by using a map matching algorithm provided by the Graphhopper library,[15] and OpenStreetMap cartography.[16] This permitted us to figure out the streets that trips were passing through. Finally, trips were encoded as a sequence of identifiers corresponding to adjacent stretches of street (that is, basic street segments with no intersections) the trip traversed, each one of them tagged with a timestamp.

  After filtering incomplete matches, 1,617,774 trips, built over 59,618 distinct street segments, were used for the dataset. Due to the nature of the network and the trips, the average number of street segments per trip is 64.74; that is, the length of the trips is longer than in Madrid dataset. Since we needed $16 = \lceil \log_2 59{,}618 \rceil$ bits to represent each segment in a trip, the total size of our plain spatial baseline is 202.85 MiB.

  For the temporal part, we considered only one kind of day. Therefore, when we sample those 24 hours into five-minute intervals, we obtain 288 distinct time intervals that are given a 9-bit time-ID. Consequently the overall size of the temporal baseline becomes 114.10 MiB. However, if we split those 24 hours into thirty-minute intervals, only 48 time intervals arise. In this case, each time-ID needs only 6 bits and the total size of the temporal baseline is 76.07 MiB. The average number of daily passengers for each time interval is shown in Figure 10.

[14] Description at http://www.geolink.pt/ecmlpkdd2015-challenge/dataset.html. Download at https://archive.ics.uci.edu/ml/machine-learning-databases/00339/train.csv.zip

[15] https://github.com/graphhopper/map-matching

[16] http://www.openstreetmap.org/

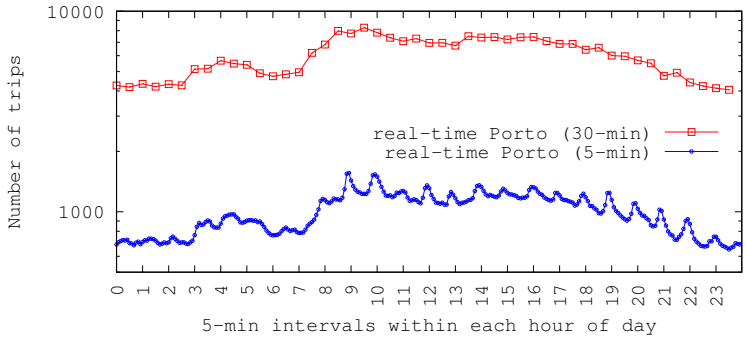

Figure 10: Real time distribution from Porto. The y-axis (in log-scale) indicates the average number of passengers per day for each time interval considering both 5-min and 30-min time intervals.

8.2. Space Requirements of CTR

We show the compression obtained by CTR when built on our two test datasets. Compression is shown as the percentage of the size of the plain baselines discussed above. Using different configurations of CTR, we will show the compression of the spatial component (CSA), that of the temporal component (WTHT and WM), and finally the overall compression of CTR.

	t_Ψ		
	32	128	512
Madrid	41.32%	26.80%	23.06%
Porto	23.66%	15.49%	13.37%

Table 1: Compression of CSA with respect to the spatial baseline.

Results regarding the compression obtained by CSA are given in Table 1. The compression ratio is calculated over a plain spatial-only (stop-IDs or street-segment-IDs in each case) representation. In a rather dense configuration of CSA with $t_\Psi = 32$ we obtain compression ratios around 41% and 23% for Madrid and Porto datasets respectively. Those results are interesting from the simple point that the baseline representations were only using respectively 9-bits per node (Madrid) and 16-bits per segment (Porto). As expected, compression improves as we increase the Ψ sampling parameter t_Ψ. We show that by tuning CSA in a more sparse setup we can almost halve the space needs of using $t_\Psi = 32$. Yet, the resulting CSA would become much slower as we will see in the next section. In general, we can see that CSA obtains better compression in Porto than in Madrid. This is probably due to the longer and more predictable trips. Note that is not common to arrive at an intersection having more than two valid street links where to navigate to.

In Table 2, we focus on the space needed by the temporal component of CTR. In this case we show the compression ratios obtained by WTHT and WM considering that time is either discretized into 5-min or 30-min intervals. Recall that the size of the plain baseline representations differs depending on the discretization period. Both WTHT and WM were tuned by using bitvector representations RG_{32}, RRR_{32}, RRR_{64}, and RRR_{128}.

It is interesting to see that in the synthetic dataset from Madrid RRR bitvectors always lead to a better compression than the plain RG, while in the real

	Type of bitvector in WM/WTHT			
	RG_{32}	RRR_{32}	RRR_{64}	RRR_{128}
Madrid (WTHT, 5-min)	91.33%	80.89%	76.90%	74.90%
Madrid (WM, 5-min)	103.13%	86.03%	80.61%	77.88%
Madrid (WTHT, 30-min)	92.30%	78.90%	74.66%	72.52%
Madrid (WM, 30-min)	103.14%	83.32%	77.90%	75.18%
Porto (WTHT, 5-min)	93.52%	102.61%	98.27%	96.11%
Porto (WM, 5-min)	103.13%	106.88%	101.41%	98.66%
Porto (WTHT, 30-min)	96.00%	103.78%	99.08%	96.74%
Porto (WM, 30-min)	103.12%	107.00%	101.50%	98.75%

Table 2: Compression of WM and WTHT with respect to the temporal baseline.

dataset from Porto that is never the case. In some cases the *RRR* does not compress the times at all. Consequently, for Porto dataset, the faster plain *RG* bitvectors are probably the best choice. In Madrid dataset, we can see an actual space/time trade-off: *RRR* obtains better compression but will be slower (as we will see in the next section).

	Type of bitvector in WM/WTHT				Type of bitvector in WM/WTHT			
	RG_{32}	RRR_{32}	RRR_{64}	RRR_{128}	RG_{32}	RRR_{32}	RRR_{64}	RRR_{128}
Madrid (WTHT, 5-min)	69.90%	63.93%	61.65%	60.51%	62.07%	56.10%	53.82%	52.68%
Madrid (WM, 5-min)	76.64%	66.87%	63.77%	62.21%	68.81%	59.04%	55.94%	54.38%
Madrid (WTHT, 30-min)	66.81%	60.11%	57.99%	56.92%	57.68%	50.98%	48.86%	47.79%
Madrid (WM, 30-min)	72.23%	62.32%	59.61%	58.25%	63.10%	53.19%	50.48%	49.12%
Porto (WTHT, 5-min)	48.81%	52.08%	50.52%	49.74%	42.22%	45.49%	43.93%	43.15%
Porto (WM, 5-min)	52.27%	53.62%	51.65%	50.66%	45.68%	47.03%	45.06%	44.07%
Porto (WTHT, 30-min)	43.39%	45.51%	44.23%	43.59%	35.91%	38.03%	36.75%	36.11%
Porto (WM, 30-min)	45.33%	46.39%	44.89%	44.14%	37.85%	38.91%	37.41%	36.66%
	$t_\Psi = 32$				$t_\Psi = 512$			

Table 3: Overall Compression of CTR including different configurations for both the spatial and temporal components.

Finally, in Table 3, we show the overall compression ratios of CTR. We use the same configurations for WTHT and WM as in Table 2, and both the most dense and sparse tuning of CSA ($t_\Psi = 32$ and $t_\Psi = 512$ respectively). For Madrid dataset, the pair (node,timestamp) is represented with $9 + 9 = 18$ bits in our baseline representation when time is discretized into 30-minute intervals, and with $9 + 12 = 21$ when we use 5-minute intervals. In the case of Porto dataset, when using 30-minute intervals, each pair (node,timestamp) from the baseline requires $16 + 9 = 25$ bits. If discretization considers 5-minute intervals, the baseline requires $16 + 6 = 22$ bits. We can see that the overall compression of CTR in Madrid dataset ranges between 76% and 50%. Also we show that Porto dataset is much more compressible, obtaining compression ratios from around 50% to 35%.

8.3. Performance at query time

Through this section, we evaluate the time performance of CTR when solving spatial, temporal, and spatio-temporal queries. We have randomly generated 10,000 query patterns from our two datasets for each type of query. Each time measurement presented below is the average execution time of 10,000 runs using the corresponding query patterns, except for the `top-k` queries where we perform 100 runs of the *top-k* algorithms with $k = \{10, 100\}$.

Our test machine has an Intel(R) Core(tm) i5-4690@3.50GHz CPU (4 cores/4 siblings) and 8GB of DDR3 RAM. It runs Ubuntu Linux 16.04 (Kernel 4.4.0-21-generic). The compiler used was g++ version 5.4.0 and we set compiler

optimization flags to $-O9$. All our experiments run in a single core and time measures refer to CPU user-time.

During the generation of query patterns, for those queries involving only one node X from the network, we have randomly chosen X 10,000 times from the available network nodes. This is the case of the query patters used both for the spatial queries `starts-with-x`, `ends-with-x`, and `uses-x` or the spatio-temporal `starts-with-x`, `ends-with-x`, and `uses-x`. In the case of the spatial `from-x-to-y` and the spatio-temporal `from-x-to-y-strong`, and `from-x-to-y-weak` the pair of network nodes $\langle X, Y \rangle$ that compose our query patterns were generated by randomly choosing 10,000 trips and then extracting the initial X and ending Y nodes of those trips.

Moreover, we also generated the time intervals $[t_1, t_2]$ required for the spatio-temporal queries. Considering the different available time-IDs, we chose a random starting instant t_1 and then randomly generated the width of that interval from five minutes to two hours. Note that if we discretized time into 5-minute intervals and *interval-width* = 59 minutes, our time interval $[t_1, t_2]$ would contain exactly 12 time IDs ($t_2 \leftarrow t_1 + 11$). However, if time was discretized into 30-minute intervals, $[t_1, t_2]$ would contain only 2 time IDs ($t_2 \leftarrow t_1 + 1$). We followed the same procedure to gather the query patterns used for the pure temporal queries `uses-t` and `starts-t`.

8.3.1. Space/time trade-off when dealing with spatial queries

In Figures 11 and 12, we show the performance of CTR at solving spatial queries for Madrid and Porto datasets respectively. Note that all these queries can be answered using only the CSA component of CTR. Therefore, the size of the temporal component is not considered here and compression values (x-axis) refer only to the size of CSA with respect to the spatial baseline as in Table 1. We show the average query time (in μs) depending on the space used by CSA with three different sampling configurations ($t_\Psi = \{512, 128, 32\}$).

Results show that the queries that involve searching in the \$ region of Ψ, such as `starts-with-x` or `from-x-to-y` are considerably slower than queries `ends-with-x` and `uses-x` due to the large size of that region. Recall there is one \$ for every trip.

In both datasets, we can see that `uses-x` (solved using *select* on D rather than *bsearch* on Ψ) is the fastest query. On average, it takes only around 10ns per query. Except in the most sparse configuration of CSA, queries `ends-with-x`, `starts-with-x`, and `from-x-to-y` require typically less than 10μs. This basically shows the cost of performing *bsearch* on a compressed Ψ. In the most sparse setup, times for `starts-with-x` and `from-x-to-y` are always better in Madrid than in Porto dataset, and `ends-with-x` draws rather identical times. With the densest configuration ($t_\Psi = 32$), `ends-with-x` and `from-x-to-y` are respectively around 10-20% fastest in Madrid dataset (`ends-with-x` takes 4.05μs and 4.51μs respectively, and `from-x-to-y` takes 4.54μs and 5.66μs). However, `starts-with-x` performs around 20% faster in Porto dataset (2.28μs vs 2.90μs).

Focusing on `top-k` queries, we can see huge differences between `top-k-starts` and the rest of the `top-k` queries, as the former needs to perform *bsearch* over the compressed Ψ instead of a *select* on D.

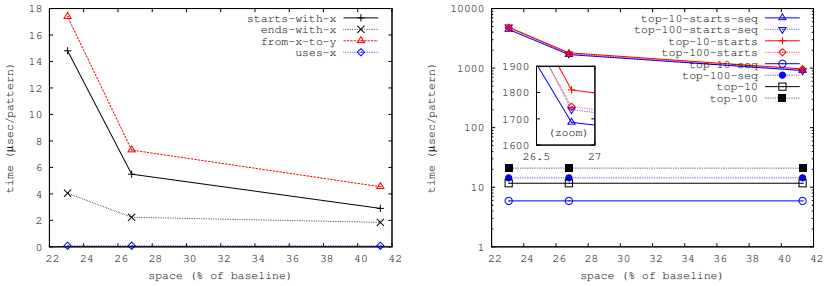

Figure 11: Spatial queries (left) and spatial *top-k* queries (right) for Madrid.

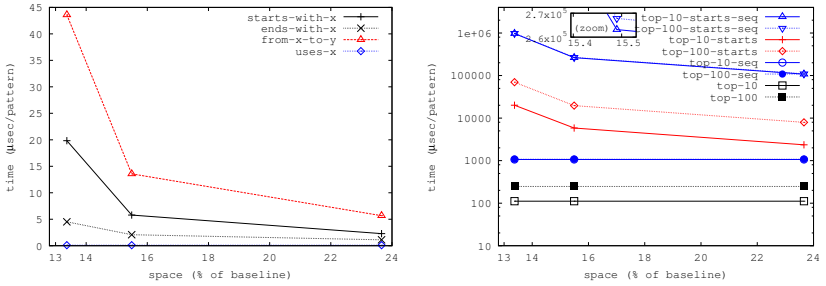

Figure 12: Spatial queries (left) and spatial *top-k* queries (right) for Porto.

We can also see that due to the small number of stops in Madrid dataset, it is always more efficient to use the sequential version of `top-k-starts` and `top-k` algorithms. This is also because a rather uniform frequency among nodes increases the number of insertions in the priority queue (i) of the binary algorithm needed for retrieving the first k nodes ($i \approx |V|$). Moreover, note that for the sequential algorithm i is at most $|V|$, whereas for the binary-partition counterpart it could become up to $2|V| - 1$.

However, in Porto dataset, where nodes follow a biased distribution (some streets are far more used than others by taxis), and whose vocabulary is 190 times larger than that of Madrid's, the binary-partition version of `top-k-starts` and `top-k` algorithms is clearly faster than the sequential counterpart (`top-k-seq` and `top-k-starts-seq`). Note that in Madrid dataset, `top-100` returns 32% of the nodes (hence sequential processing worths it) whereas in Porto dataset less than 0.2% of the nodes are returned.

The gap between `top-10-seq` and `top-100-seq` that we can clearly appreciate in Madrid dataset is due to the cost of the insertion of nodes in the min-heap. However, the gap between the binary `top-10` and `top-100` is mainly related to the number of iterations performed until the binary-partition algorithm gathers the first 10 and 100 nodes returned respectively. The same discussion applies for `top-k-starts` queries.

8.3.2. Space/time trade-off when performing temporal queries

In this section we focus on the performance of the temporal component of CTR. We use the same configurations as in Table 2 for WM and WTHT, and show the space/time trade-offs obtained when solving pure temporal queries. Figures 13 and 14 present the results obtained at `uses-t` and `starts-t` queries

for Madrid and Porto datasets respectively. Note that, in this case, since the CSA is not actually needed to solve temporal queries, we do not include its size within the compression values (x-axis).

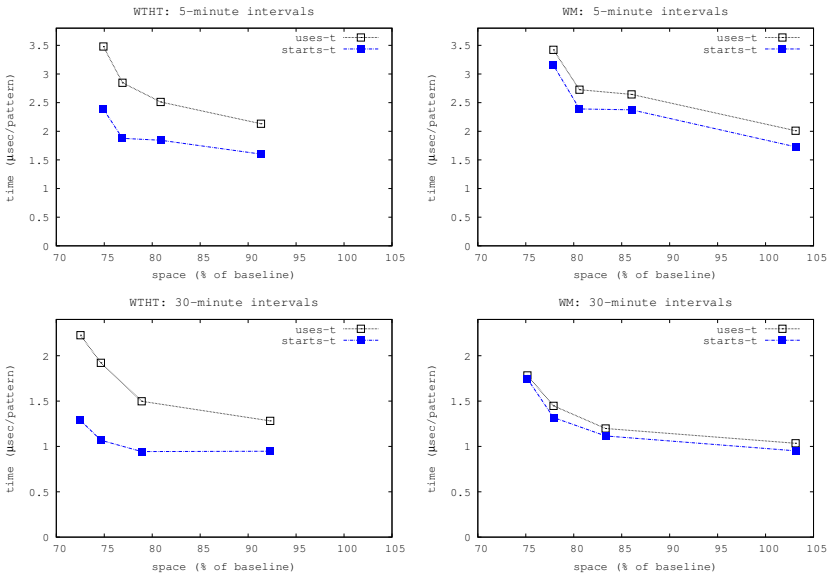

Figure 13: Pure temporal queries for Madrid. CTR uses either a WTHT (left) or a WM (right). Time granularity is 5 minutes (top) or 30 minutes (bottom).

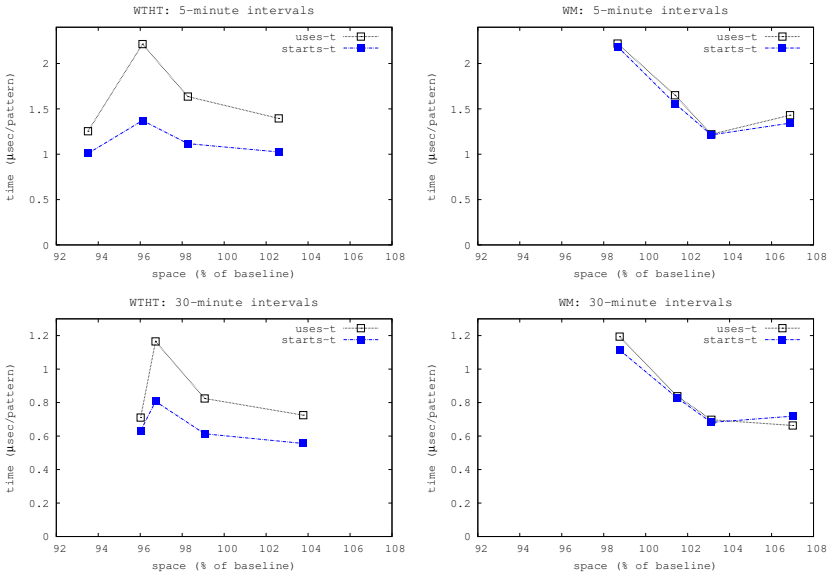

Figure 14: Pure temporal queries for Porto. CTR uses either a WTHT (left) or a WM (right). Time granularity is 5 minutes (top) or 30 minutes (bottom).

We can see that when running `uses-t` queries, both WTHT and WM obtain rather similar times (requiring less than 4μs to perform a *count* operation in all cases) and that those times improve as the height of the structure decreases.

We can see that in the highest WTHT and WM, corresponding to using 5-min intervals in Madrid dataset, `uses-t` requires less than 3.5μs. Then, when using 30-min intervals, the time required to solve `uses-t` is always below 2.3μs (yet WM performs faster than WTHT here), and those times are similar to the ones obtained for Porto dataset when using 5-min intervals. And finally, the best query times (below 1.2μs) are obtained for Porto dataset with 30-min intervals.

Regarding `starts-t`, recall that it also performs a *count* operation, but within a smaller range ($[1, z]$) in comparison with the range $[z + 1, n]$ where *count* is performed for `uses-t`. We can see that, whereas WM obtains similar times to those of `uses-t` query, `starts-t` performs clearly faster than `uses-t` over WTHT.

As a final note, recall that in Madrid dataset, bitvector RG always needs more space than RRR counterparts whereas in Porto dataset (as discussed in Section 8.2) RG obtains the best space values when using 5-min intervals and still requires less space than RRR_{32} when using 30-min intervals. This is the reason why while plots for Madrid dataset are decreasing from left to right, in Porto dataset the first point (RG) in the left figures (5-min intervals), and the third point (RG) in the right figures (30-min intervals) require less space than the others (RRR) and are also typically faster.

8.3.3. Space/time trade-off when performing spatio-temporal queries

In Figures 15 and 16 we show the space/time tradeoff obtained by CTR when dealing with spatio-temporal queries. Recall that this type of queries require both using the CSA, to exploit indexed access to the nodes in the trips, and the temporal component of CTR to handle temporal constraints. In this case, the space values showed in the figures include both the size of CSA and that of either WM or WTHT. Therefore, we also show the overall space needs of CTR. In the case of CSA we have set $t_\Psi = 32$ (a fixed dense sampling), and for WM and WTHT we used again the same configurations as in the previous sections obtained by varying the bitvectors and the temporal discretization.

For queries `starts-with-x`, `ends-with-x`, and `uses-x` we can see typically small differences between using WM or WTHT. In Madrid dataset, WM overcomes WTHT being 2-30% faster in these types of queries. However, in Porto dataset WTHT is slightly faster (from 1 to 25%) than its WM counterpart.

For queries `from-x-to-y-strong` and `from-x-to-y-weak` we can see a big gap between the times reported by WTHT and WM. This gap arises because in WM we have used exactly the *countLR* operation discussed in Section 7 that is implemented with two calls to the *count* operation from the WM.[17] However, in our implementation of WTHT we have engineered an improved version of *countLR* where, during the execution of *count*, we also report α' and β', hence avoiding two calls to *count*.

Finally, we also include results for `top-k` and `top-k-starts` queries in Figures 17 and 18. As explained in Section 8.3.1, the sequential approach is preferred when the frequency distribution of nodes is rather uniform (Madrid dataset). Otherwise, the binary-partition counterpart outperforms it. The need for applying a temporal constraint simply accentuates this effect in comparison

[17] For WM we used exactly the same implementation in [38] and simply added the new operation *countLR* that calls the underlying *count* from the WM.

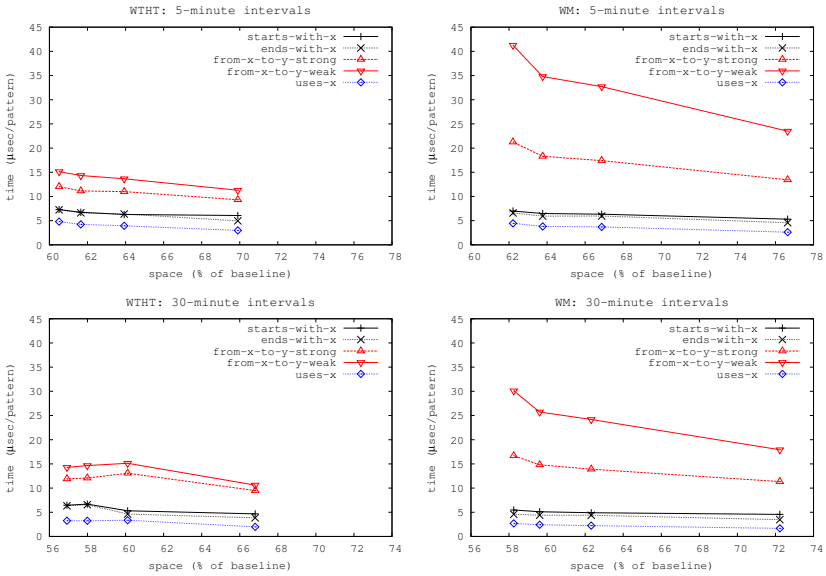

Figure 15: Spatio-temporal queries for Madrid. CTR uses either a WTHT (left) or a WM (right). Time granularity is 5 minutes (top) or 30 minutes (bottom).

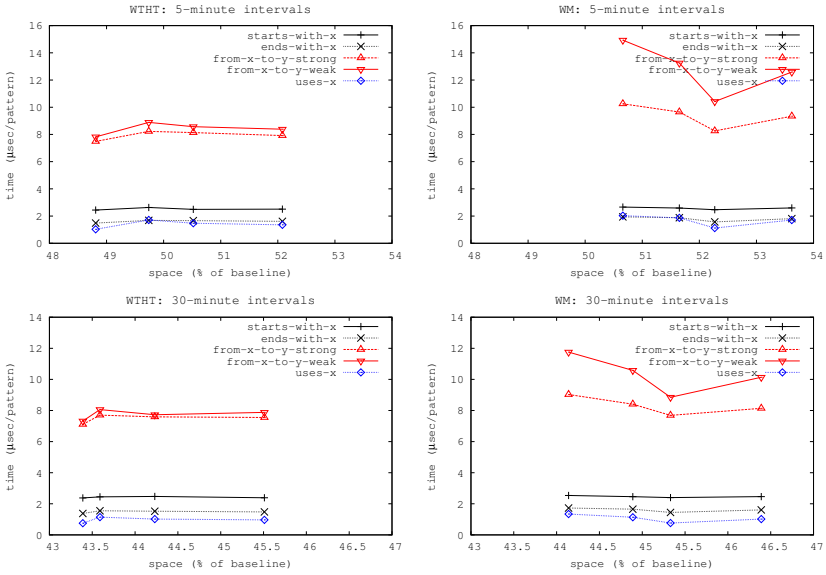

Figure 16: Spatio-temporal queries for Porto. CTR uses a fixed $t_\Psi = 32$ for CSA, and either a WTHT (left) or a WM (right). Time granularity is 5 minutes (top) or 30 minutes (bottom).

with the corresponding pure spatial queries.

9. Conclusions and future work

With the installation of better user-tracking mechanisms in public transportation networks, or the fact that a simple app installed in a mobile phone

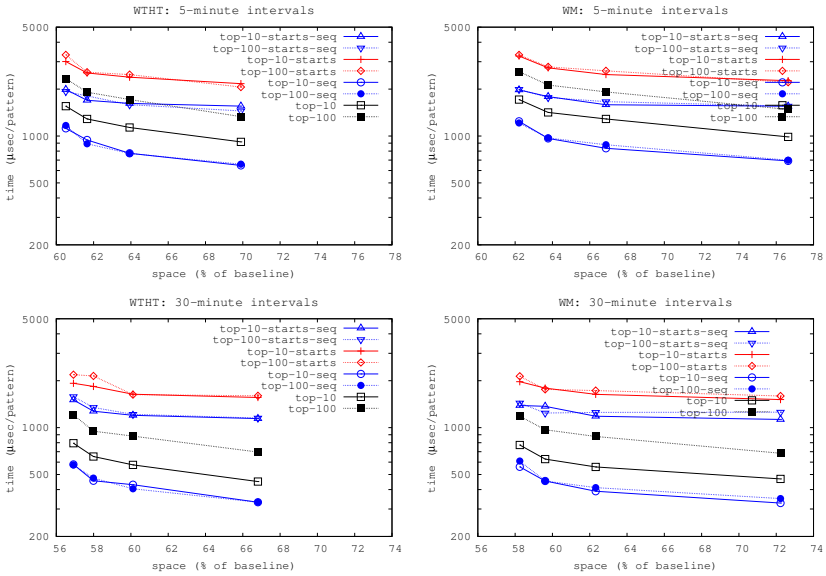

Figure 17: Spatio-temporal `top-k` and `top-k-starts` queries for Madrid. CTR uses either a WTHT (left) or a WM (right). Time granularity is 5 minutes (top) or 30 minutes (bottom).

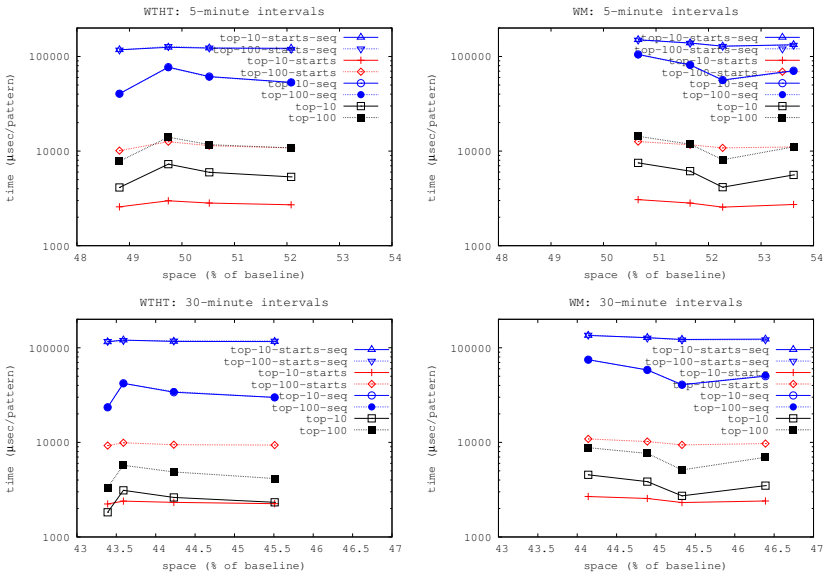

Figure 18: Spatio-temporal `top-k` and `top-k-starts` queries for Porto. CTR uses a fixed $t_\Psi = 32$ for CSA, and either a WTHT (left) or a WM (right). Time granularity is 5 minutes (top) or 30 minutes (bottom).

permits us to track user movements, the problem of storing user trips to finally support network analysis operations has been gaining increasing interest in multiple scenarios. For example, we could consider a network management administration, a taxi company, services like Uber, Cabify, Car2go, or simply end-user applications.

With enough data of vehicle trips from a significant amount of drivers over the network composed of the streets in a city, it would be possible to infer traffic rules by examining turns that nobody takes, their usual driving speed across the network, congestion points at a given time, and other useful information. Also, a taxi company (or similar services) could benefit from knowing the city areas where it is more probable that a user would start a trip, the average time to go from one area to another, etc. This also applies for the administrators of public transportation networks including buses, trains, subway, etc.

We have presented CTR and showed that it is a powerful tool to represent user trips. Actually, we have used CTR to handle user trips from two different scenarios: the network of subway and local trains from Madrid, and taxi trips from Porto. CTR uses compact data structures to store both the nodes traversed (spatial component) by an user during a trip and the corresponding timestamps (temporal component). This permits us not only to reduce the amount of data to store but also to efficiently perform spatial, temporal, and spatio-temporal queries that can help us to analyze the actual usage of the network.

In particular, we used the well-known CSA to represent the spatial component of the trips. For Madrid dataset, the size of CSA is around 20-40% the size of the source data. Porto dataset is still more compressible and CSA requires only around 13-24% the space of the original data. This structure is enough to solve typical spatial queries within microseconds and `top-k` queries in milliseconds. For the temporal component, we used two WT-based structures. We adapted the existing balanced WM and we created a *Hu-Tucker-shaped* WT (WTHT) that permits to exploit a biased distribution of times to gain compression. These structures obtained only a moderate improvement in compression with respect to a plain representation of times (compression ratio from 70 to 105%), but they provided indexed access to the temporal data, and consequently allowed us to support temporal queries very efficiently. Finally, we have also shown that the overall CTR, including both CSA and either WTHT or WM, permits also to efficiently solve spatio-temporal queries (within microseconds); that is, spatial queries constrained to a time period. The overall compression obtained by CTR is around 55-75% in Madrid dataset and around 43-54% in Porto dataset.

We have presented CTR as a proof of concept development, and we have shown how to solve different types of queries. Yet, based on the underlying data-structures, CTR is flexible enough to allow us to increase its functionality. As future work, we are interested in exploiting the underlying network topology to obtain a more compact representation of the nodes in the trips. Also, we want to explore ways to improve the compression of the temporal component. In this line, we consider that an inverted-index based representation can be promising.

References

[1] N. R. Brisaboa, A. Fariña, D. Galaktionov H., M. A. Rodríguez, Compact trip representation over networks, in: Proc. 23th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 9954, 2016, pp. 240–253.

[2] M. A. Munizaga, C. Palma, Estimation of a disaggregate multimodal public transport Origin–Destination matrix from passive smartcard data from Santiago, Chile, Transportation Research Part C: Emerging Technologies 24 (2012) 9–18. doi:10.1016/j.trc.2012.01.007.

[3] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. Lorentzos, E. Nardelli, M. Schneider, J. R. R. Viqueira, Chapter 4: Spatio-temporal models and languages: An approach based on data types, in: Spatio-Temporal Databases: The CHOROCHRONOS Approach, LNCS 2520, 2003, pp. 117–176.

[4] S. Spaccapietra, Editorial: Spatio-Temporal Data Models and Languages, GeoInformatica 5 (1) (2001) 5–9. doi:10.1023/A:1011403703806.

[5] L. Forlizzi, R. H. Güting, E. Nardelli, M. Schneider, A data model and data structures for moving objects databases, in: Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD), 2000, pp. 319–330. doi:10.1145/342009.335426.

[6] M. Erwig, R. H. Güting, M. Schneider, M. Vazirgiannis, Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases, GeoInformatica 3 (3) (1999) 269–296. doi:10.1023/A:1009805532638.

[7] N. Pelekis, Y. Theodoridis, Mobility Data Management and Exploration, Springer, 2014. doi:10.1007/978-1-4939-0392-4.

[8] D. Pfoser, C. S. Jensen, Y. Theodoridis, Novel Approaches in Query Processing for Moving Object Trajectories, in: Proc. 26th International Conference on Very Large Data Bases (VLDB), 2000, pp. 395–406.

[9] Y. Tao, D. Papadias, MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries, in: Proc. 27th International Conference on Very Large Data Bases (VLDB), 2001, pp. 431–440.

[10] E. Frentzos, Indexing Objects Moving on Fixed Networks, in: Proc. 8th International Symposium on Spatial and Temporal Databases (SSTD), 2003, pp. 289–305.

[11] V. T. de Almeida, R. H. Güting, Indexing the Trajectories of Moving Objects in Networks, GeoInformatica 9 (1) (2005) 33–60. doi:10.1007/s10707-004-5621-7.

[12] I. Sandu Popa, K. Zeitouni, V. Oria, D. Barth, S. Vial, Indexing In-network Trajectory Flows, The VLDB Journal 20 (5) (2011) 643–669. doi:10.1007/s00778-011-0236-8.

[13] P. Cudré-Mauroux, E. Wu, S. Madden, TrajStore: An Adaptive Storage System for Very Large Trajectory Data Sets, in: Proc. 26th International Conference on Data Engineering (ICDE), 2010, pp. 109–120.

[14] Y. Li, C. Chow, K. Deng, M. Yuan, J. Zeng, J. Zhang, Q. Yang, Z. Zhang, Sampling big trajectory data, in: Proc. 24th ACM International Conference on Information and Knowledge Management (CIKM), 2015, pp. 941–950.

[15] K. Sadakane, New text indexing functionalities of the compressed suffix arrays, Journal of Algorithms 48 (2) (2003) 294–313.

[16] R. Grossi, A. Gupta, J. S. Vitter, High-order entropy-compressed text, in: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2003, pp. 841–850.

[17] O. Wolfson, B. Xu, S. Chamberlain, L. Jiang, Moving objects databases: Issues and solutions, in: Proc. 10th International Conference on Scientific and Statistical Database Management (SSDBM), 1998, pp. 111–122.

[18] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao, Modeling and querying moving objects, in: Proc. 13th International Conference on Data Engineering (ICDE), 1997, pp. 422–432.

[19] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, M. Vazirgiannis, A foundation for representing and querying moving objects, ACM Transactions on Database Systems 25 (1) (2000) 1–42. doi:10.1145/352958.352963.

[20] R. H. Güting, M. Schneider, Moving Objects Databases, Morgan Kaufmann, 2005.

[21] M. L. Damiani, H. Issa, R. H. Güting, F. Valdés, Symbolic trajectories and application challenges, SIGSPATIAL Special 7 (1) (2015) 51–58. doi:10.1145/2782759.2782768.

[22] R. H. Güting, V. Teixeira de Almeida, Z. Ding, Modeling and querying moving objects in networks, The VLDB Journal 15 (2) (2006) 165–190. doi:10.1007/s00778-005-0152-x.

[23] Z. Ding, B. Yang, R. H. Güting, Y. Li, Network-matched trajectory-based moving-object database: Models and applications, IEEE Trans. Intelligent Transportation Systems 16 (4) (2015) 1918–1928. doi:10.1109/TITS.2014.2383494.

[24] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD), 1984, pp. 47–57.

[25] M. A. Nascimento, J. R. Silva, Towards historical R-trees, in: Proc. ACM symposium on Applied Computing (SAC), ACM, 1998, pp. 235–240.

[26] V. P. Chakka, A. Everspaugh, J. M. Patel, Indexing large trajectory data sets with SETI, in: Proc. 1st Conference on Innovative Data Systems Research (CIDR), 2003, pp. 1–12.

[27] J.-W. Chang, M.-S. Song, J.-H. Um, TMN-tree: new trajectory index structure for moving objects in spatial networks, in: Proc. 10th International Conference on Computer and Information Technology (CIT), 2010, pp. 1633–1638. doi:10.1109/CIT.2010.289.

[28] D. H. T. That, I. S. Popa, K. Zeitouni, TRIFL: A generic trajectory index for flash storage, ACM Transactions on Spatial Algorithms and Systems 1 (2) (2015) 6. doi:10.1145/2786758.

[29] N. Meratnia, R. A. de By, Spatiotemporal compression techniques for moving point objects, in: Proc. 9th International Conference on Extending Database Technology (EDBT), LNCS 2992, 2004, pp. 765–782.

[30] M. Potamias, K. Patroumpas, T. Sellis, Sampling Trajectory Streams with Spatiotemporal Criteria, in: Proc. 18th International Conference on Scientific and Statistical Database Management (SSDBM), 2006, pp. 275–284. doi:10.1109/SSDBM.2006.45.

[31] H. Cao, O. Wolfson, G. Trajcevski, Spatio-temporal data reduction with deterministic error bounds, The VLDB Journal 15 (3) (2006) 211–228. doi:10.1007/s00778-005-0163-7.

[32] K.-F. Richter, F. Schmid, P. Laube, Semantic Trajectory Compression: Representing Urban Movement in a Nutshell, Journal of Spatial Information Science 4 (1) (2012) 3–30. doi:10.5311/JOSIS.2012.4.62.

[33] G. Kellaris, N. Pelekis, Y. Theodoridis, Map-matched Trajectory Compression, Journal of Systems and Software 86 (6) (2013) 1566–1579. doi:10.1016/j.jss.2013.01.071.

[34] S. Funke, R. Schirrmeister, S. Skilevic, S. Storandt, Compass-based navigation in street networks, in: Proc. 14th International Symposium on Web and Wireless Geographical Information Systems (W2GIS), LNCS 9080, 2015, pp. 71–88.

[35] B. Krogh, N. Pelekis, Y. Theodoridis, K. Torp, Path-based queries on trajectory data, in: Proc. 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL), 2014, pp. 341–350.

[36] S. Koide, Y. Tadokoro, T. Yoshimura, SNT-index: Spatio-temporal index for vehicular trajectories on a road network based on substring matching, in: Proc. 1st International ACM SIGSPATIAL Workshop on Smart Cities and Urban Analytics (UrbanGIS@SIGSPATIAL), 2015, pp. 1–8. doi:10.1145/2835022.2835023.

[37] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. 41st IEEE Symposium on Foundations of Computer Science (FOCS), 2000, pp. 390–398.

[38] F. Claude, G. Navarro, A. Ordóñez , The wavelet matrix: An efficient wavelet tree for large alphabets, Information Systems 47 (2015) 15–32.

[39] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948. doi:10.1137/0222058.

[40] R. Grossi, J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, in: Proc. 32nd ACM Symposium on Theory of Computing (STOC), 2000, pp. 397–406.

[41] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS), 1989, pp. 549–554.

[42] I. Munro, Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 1180, 1996, pp. 37–42.

[43] G. Navarro, V. Mäkinen, Compressed Full-text Indexes, ACM Computing Surveys 39 (1) (2007) article 2. doi:10.1145/1216370.1216372.

[44] A. Fariña, N. R. Brisaboa, G. Navarro, F. Claude, Á. S. Places, E. Rodríguez, Word-based Self-Indexes for Natural Language Text, ACM Transactions on Information Systems 30 (1) (2012) article 1:. doi:10.1145/2094072.2094073.

[45] T. Gagie, G. Navarro, S. J. Puglisi, New algorithms on wavelet trees and applications to information retrieval, Theoretical Computer Science 426 (2012) 25–41.

[46] G. Navarro, Compact Data Structures – A practical approach, Cambridge University Press, 2016.

[47] G. Navarro, Wavelet trees for All, Journal of Discrete Algorithms 25 (2014) 2–20. doi:10.1016/j.jda.2013.07.004.

[48] A. Golynski, R. Grossi, A. Gupta, R. Raman, S. S. Rao, On the size of succinct indices, in: Proc. 15th Annual European Symposium on Algorithms (ESA), LNCS 4698, 2007, pp. 371–382.

[49] R. Raman, V. Raman, S. S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002, pp. 233–242.

[50] D. A. Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the IRE 40 (9) (1952) 1098–1101. doi:10.1109/JRPROC.1952.273898.

[51] P. Ferragina, R. González, G. Navarro, R. Venturini, Compressed text indexes: From theory to practice, Journal of Experimental Algorithmics 13 (2009) 1–12.

[52] J. Barbay, G. Navarro, On compressing permutations and adaptive sorting, Theoretical Computer Science 513 (2013) 109–123. doi:10.1016/j.tcs.2013.10.019.

[53] T. C. Hu, A. C. Tucker, Optimal computer search trees and variable-length alphabetical codes, SIAM Journal on Applied Mathematics 21 (4) (1971) 514–532. doi:10.1137/0121057.

[54] T. M. Cover, J. A. Thomas, Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing), 2nd Edition, Wiley-Interscience, 2006.

[55] Y. Horibe, An improved bound for weight-balanced tree, Information and Control 34 (2) (1977) 148–151. doi:10.1016/S0019-9958(77)80011-9.

[56] E. N. Gilbert, E. F. Moore, Variable-length binary encodings, Bell System Technical Journal 38 (4) (1959) 933–967. doi:10.1002/j.1538-7305.1959.tb01583.x.

[57] F. Claude, G. Navarro, Practical rank/select queries over arbitrary sequences, in: Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE), LNCS 5280, 2008, pp. 176–187.

[58] A. Ordóñez, Statistical and repetition-based compressed data structures, Ph.D. thesis, Department of Computer Science, University of A Coruña (2016).

[59] A. Fariña, T. Gagie, G. Manzini, G. Navarro, A. Ordóñez, Efficient and Compact Representations of Some Non-canonical Prefix-Free Codes, LNCS 9954, 2016, pp. 50–60.

[60] C. Morency, M. Trépanier, B. Agard, Measuring transit use variability with smart-card data, Transport Policy 14 (3) (2007) 193–203. doi:10.1016/j.tranpol.2007.01.001.

[61] A. El-Geneidy, D. Levinson, Place rank: Valuing spatial interactions, Networks and Spatial Economics 11 (4) (2011) 643–659. doi:10.1007/s11067-011-9153-z.

[62] G. Wang, Y. Zhong, C.-P. Teo, Q. Liu, Flow-based accessibility measurement: The place rank approach, Transportation Research Part C: Emerging Technologies 56 (2015) 335–345. doi:10.1016/j.trc.2015.04.017.

[63] G. Nong, S. Zhang, W. H. Chan, Two efficient algorithms for linear time suffix array construction, IEEE Transactions on Computers 60 (10) (2011) 1471–1484.

[64] N. J. Larsson, K. Sadakane, Faster suffix sorting, Theoretical Computer Science 387 (3) (2007) 258–272. doi:10.1016/j.tcs.2007.07.017.

[65] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX), 2007, pp. 60–70. doi:10.1137/1.9781611972870.6.

[66] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, L. Damas, Predicting taxi–passenger demand using streaming data, IEEE Transactions on Intelligent Transportation Systems 14 (3) (2013) 1393–1402.