Universitat Rovira i Virgili (URV) y Universitat Oberta de Catalunya (UOC)

Master in Computational and Mathematical Engineering

FINAL MASTER PROJECT

Area: Computational Graph Theory

Optimized Path Dataset Representation

Author: Daniel Pérez Cervera

Tutor: Hebert Pérez Rosés

Barcelona, July 2, 2023

Dr./Dra. (name) ....., certifies that the student (name) ..... has elaborated the work under his/her direction and he/she authorizes the presentation of this memory for its evaluation.

Director's signature:

Credits/Copyright

The contents of this thesis and the application developed belongs to the author of this thesis together with *Aimsun S.L.U.*. The contribution of the latter is the dedication of 80 hours of work of the author and providing the computational setup to partially implement and execute the derived application with the purpose of obtaining the results shown. The author's contribution is the rest of the work of this thesis and the implementation of the derived solution.

- **Open Street Maps**: the source of data from which the raw road network data is obtained,
- **Osmosis**: used to extract raw data associated to different use cases.
- **PyGraphViz**: used to extract data associated to different use cases.
- **NetworkX**: used to manipulate and clean data associated to the use cases.
- **GeonetworkX**: used to manipulate and clean data associated to the use cases.
- **GeoPY Library**: used to manipulate and clean data associated to the use cases.
- **Pandas**: used to manipulate and clean data associated to the use cases.

FINAL PROJECT SHEET

Title:	Optimized Path Dataset Representation
Author:	Daniel Pérez Cervera
Tutor:	Hebert Pérez Rosés
Date :	July 2, 2023
Program:	Master in Computational and Mathematical Engineering
Area:	Computational Graph Theory
Language:	English
Keywords:	path dataset, road network, trie

Abstract

Path-based data is increasingly used in mobility applications. This type of data has multiple uses, among them studying and predicting travelers behavior. Paths are usually represented as a sequence of vertices on a road network graph, sequences of geographical coordinates and omitting its temporal component. In this work, we propose a novel, generic and optimized representation of Path Datasets which can be used for any type of data. Such approach is designed to be efficient for collecting, storing and processing path-based data associated to Road Network trips. This approach is novel, as it considers representing paths collectively by origin vertex using tries to index paths uniquely. In particular, for path storage, where we use two effective techniques: *DFUDS* and *Adaptive Edge Offset Compression*. We evaluate this approach with other baseline approaches using synthetically generated Path Datasets, to assess the gains in performance (computation time, memory, and storage size) and explain them mainly through a quantity we define, the overlap θ. Results show this approach is more efficient for higher overlap among the paths stored, and never worse than storing for the generated path datasets. The only exception is for the collection stage, in which time and space complexity are higher in some situations.

Keywords: path dataset, trie, road networks

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Path-based data has become more and more popular and useful over time. The proliferation of multiple mobility services, ranging from route recommendation, to fleet planning, and, consequently, the huge amount of data collected, raises the question of how to store such vast amounts of path-base (aka. trajectory-based) data. For the scope of this thesis, we call the collection of individual paths together with any data associated to each of them, a *Path Dataset*. Using the information contained in them, opens the door to analyzing and understanding performances of those applications, as well as discovering patterns in travelers behavior, or more generally, Traffic Modelling. That is why, efficient methods are needed to manage this type of data.

The objective of the present dissertation is developing a efficient representation to collect, store and process Path Datasets in the context Traffic Modeling applications, such as the ones developed by [Aismun]. This implies restricting our research to Path Datasets generated in Road Networks and describing the paths as sequences of vertices on the Road Network Graph. Regarding the latter, this representation is used because when path-based data is collected, it is as a temporal sequence of geographical coordinates $\{(t_i, (x_i, y_i)\}_{i=1}^{n}$. However, this data cannot be directly used to model the traveler choices, and is mapped to a sequence of vertices (choice) on a graph (choice set). We also omit the time component in order not to broaden excessively the scope of this thesis.

We have found few works addressing the problem at hand. In the literature, many of the approaches focus on compressing Path Datasets in the finest detail. This is likely because, those approaches are generic and, unlike us, they know in advance, the use of this data, and therefore

the acceptable level of detail. See [4] and [13] for more information on those approaches. Among the ones matching our use case, we have only found references for efficiently storage of paths: [10], [17] and [14]. The first two use trained machine-learning models in a specific scenario to reduce the size of frequently used sub-paths. Those methods are effective, but one of its major disadvantages is the need of training it, which involves a substantial computational burden and may not be robust to changes of the Path Datasets. The third one relies on robust features of Road Networks to encode the vertices of the vertex sequence. However, is a component for storing (efficiently) and not a full solution.

We propose an approach tailored to our use case, which is representing Path Datasets in a set of origin -Trie pairs, each Trie indexing the data of all paths of the same origin. In this way, each node of the Trie labels a different path and the data associated to it. This representation shares make the path descriptions shared, reducing accordingly the time and space complexities. For simplicity, we will only consider the data associated to each path is the number of instances (f.i. travelers) it is realized in the Path Dataset. However, our approach admits extension to any kind of data. Finally, regarding the store & restore of this data to disk, we proposed an approach combining [14] together with a succinct representation called DFUDS, [3].

This master thesis is divided in the following sections:

- **Theoretical framework**. In this chapter, we provide all the preliminary knowledge and notation needed to comprehend the rest of the thesis.
- **Literature Review**. There, we discuss in detail some of the most relevant works in literature that are relevant for our work.
- **Problem Description**. Where we describe the problem at hand in accurately and specific goals any solution must achieve.
- **Proposed Solution: Compressed Trie Indexation**. This chapter describes our approach in full detail, including the algorithms and complexity analysis of them.
- **Experimental Design**. There, we discuss how to carry out the experimentation to validate our solution consistently, including key aspects of the implementation of our solutions.
- **Analysis of Results**. Where, we analyze the results obtained from applying the Experimental Design on different use cases and discuss them.
- **Conclusions & Future Work**. Finally, there we expose the conclusions based on the results obtained and discuss potential future work built upon the one in this thesis.

Chapter 2

Theoretical Framework

In this section, we provide the preliminary knowledge and notation to understand the topics discussed in the following sections.

2.1 Graph theory

2.1.1 Definitions

Graphs are used to describe problems in multiple applications, ranging from internet connectivity to modelling social relationships or transportation networks. They are used to describing relationships (edges) between objects (vertices). The formal definition of a graph ([9]) is the following one:

Definition 2.1.1 (Graph)**.** A graph $G(V,E)$ a pair (origin and destination) where V is a set whose elements are called vertices (singular: vertex), and E is a set of paired vertices, whose elements are called edges.

We use the notation $e = e(v,w)$ to label an edge $e \in E$ that has as origin vertex $v \in V$ and $w \in V$ as destination vertex. Besides the base definition, more definitions are needed in order to provide a complete description of the operations and algorithms we can use upon them.

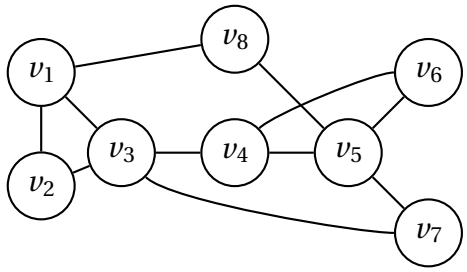


Figure 2.1: Example of graph. The enumerated circles represent vertices and the lines connecting them, edges .

Definition 2.1.2 (Adjacency)**.** $v \in V$ is adjacent to $w \in V$ if there is an edge with v as origin vertex and w as destination vertex or vice versa.

Definition 2.1.3 (In-degree)**.** The in-degree (or incoming-degree) of $v \in V$, $d_i(v)$, is The number of edges with v as destination vertex.

Definition 2.1.4 (Out-degree)**.** The out-degree (or outgoing-degree) of $v \in V$, $d_o(v)$, is The number of edges with v as origin vertex.

Additionally, some more concepts are used to describe the properties and results of graphs:

Definition 2.1.5 (Walk)**.** A walk of length L from $v \in V$ to $w \in V$ is a sequence of edges $\{e_i\}_{i=1,\dots,L}$ for which:

- The origin of the first edge in the sequence is v. $e_1 = e_1(v, v_1)$.
- The destination of the last edge in the sequence is w. $e_L = e_L(v_L, w)$.
- The destination vertex of the i-th edge in the sequence is the origin vertex of the $(i+1)$-th one in the sequence. Except for the last one.

$$\forall\, i < L-1,\ e_i(v_i, v_{i+1})\ \&\ e_{i+1}(v_{i+1}, v_{i+2}) \tag{2.1}$$

We use the following notation to describe walks:

$$e(v_1, v_2)\, e(v_2, v_3)\, e(v_3, v_4)\, \dots\, e(v_L, v_{L+1}) \tag{2.2}$$

Notice a walk of length L traverses $L+1$ vertices. Those vertices may be repeated or not, in case they are not repeated, we call the sequence a Path.

Definition 2.1.6 (Path)**.** A path of length L from $v \in V$ to $w \in V$ is a walk L from $v \in V$ to $w \in V$ without repeated nodes, i.e., without loops.

Paths and walks allows us to describe the combination of edges that connect two vertices and their properties. The definition of connectivity is the generalization of adjacency:

Definition 2.1.7 (Connected vertices)**.** $v \in V$ is connected to $w \in V$ if exists a walk from v to w.

A path may be represented a sequence of vertices $\{v_i\}_{i=1}^{L}$. Alternatively, we can also use *Edge Offsets* (e), which are obtained by enumerating the outgoing edges of a vertex. Therefore, edge offsets range from 0 to $d_o(v_v)$. In this way, the Edge Offset, e representation would be the original vertex v_0 plus a sequence of *Edge Offsets*, $\{e_i\}_{i=1}^{L-1}$. This will be described in more detail in Adaptive Edge Offset Compression.

The previous definitions are applicable to any graph. However, for different uses, different requirements and restrictions arise and with them, different types of graphs. There are multiple ways to categorize graphs, but among all categorizations, there are three categorizations which are the most relevant ones for this work:

- **Directed/Undirected**. In a directed graph, an edge expresses a one-way relationship. Therefore, $x \rightarrow y$ expresses that x is related to y but y is not necessarily related to x. In an undirected graph, if an edge related x to y it also related y to x.
- **Simple/Multigraphs**. Simple graphs are those graphs for which at most one edge exists between any pair of nodes. In contrast to multigraphs, for which a pair of nodes may be connected by more tan one edge.
- **Weighted/Unweighted**. Weighted graphs are the ones which, for each edge, have, additionally, a weight associated. In such a way that, the weight of traversing a set of edges is the sum of the weights of each individual edge. Weights usually describe the penalty of traversing an edge.

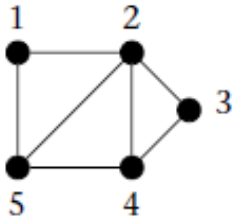
For the scope of this work, only simple directed graphs will be considered. We will additionally consider weighted graphs when modeling the routes chosen by travelers, as they tend to minimize its experience cost (weight).

2.1.2 Graph Representations: CRS

In order to represent a graph, we need to represent two sets: the vertices and the edges. We will discuss only the case for simple directed graphs. Firstly, vertices can be indexed with numbers from 1 to the cardinality of $|V|$. And we also use this indexation to index the vertices implicitly in the data structure. Custom type of indexing vertices can always be built on top of it. However, the main complexity comes from representing the relations between vertices, the edges. There are two main approaches for this:

- **Adjacency Matrix**. A $|V| \times |V|$ matrix where each cell of row r and column c is 1 if there is an edge from vertex labeled as r to vertex labeled as c.
- **Adjacency List**. A list of $|V|$ lists is used. Where each sub-list is associated to an (origin) vertex, r, and it stores of the associated destination vertices c. Therefore, is of size $d_o(v)$.

An example of those representations is:



$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{array}{lcl} 1 & : & 2,5 \\ 2 & : & 1,3,4,5 \\ 3 & : & 2,4 \\ 4 & : & 2,3,5 \\ 5 & : & 1,2,4 \end{array}$$

Figure 2.2: a) Example graph. b) and c) Adjacency matrix and list representations of graph.

Adjacency matrices are in general more efficient for dense graphs, where the in/out-degree of each node is similar than the number of nodes in the graph: $d_i(v) \simeq |V|$ and $d_o(v) \simeq |V|$. In contrast, adjacency lists are more efficient in time and space for very sparse graphs: $d_i(v) \ll |V|$ and $d_o(v) \ll |V|$. Unfortunately, Adjacency list implementations typically suffer from poor locality in memory because they are implemented as pointer to multiple non-contiguous blocks of memory.

To overcome this, the **Compressed Sparse Row**(CRS) format has been developed. This format is a specific type of adjacency lists in which the destination vertices are located for each vertex consecutively in a single list. In this way, two lists are needed: the once containing the destination

vertex labels of size E for directed graphs (Edge Destination) and another to indicate, for each vertex c, the position where the associated destination's range start (Source Degrees). An additional end of range marker is needed in order to indicate the end of the last vertex destinations (END). Because the ranges are consecutive, the (i+1)-th vertex start marks the i-th vert end. Therefore, only $V+1$ indices are needed for the second list. The memory layout for example of the previous figure would be the following one:

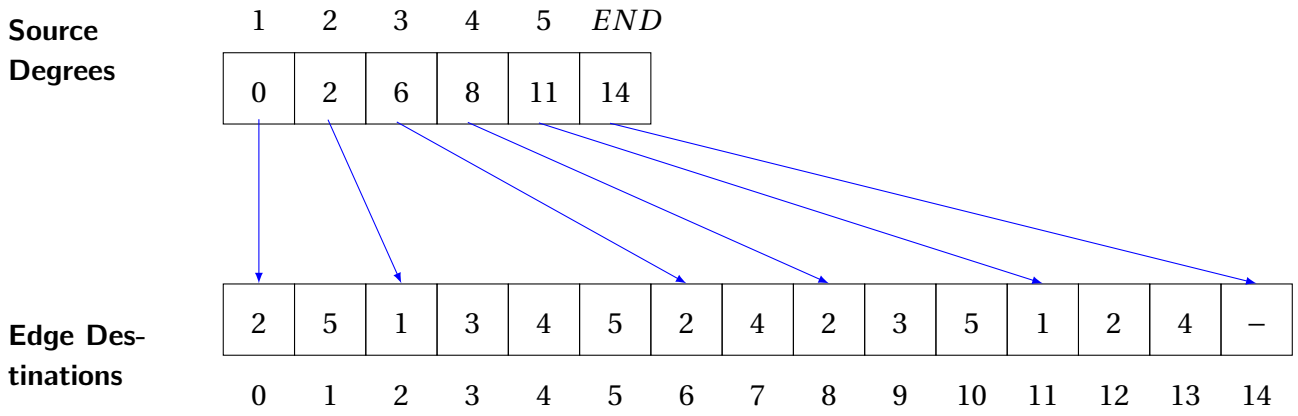


Figure 2.3: Example of CRS graph representation of graph Figure 2.2.

This approach achieves an excellent locality by using just two blocks of contiguous memory. In exchange. however, adding and removing edges can be very expensive, because memory blocks are contiguous and adding or removing the edge at the j-th position implies moving also all the ones after it, in worst case $\mathcal{O}(E)$. Implementations of this can be found in well-known libraries such as Boost.

2.1.3 Trees

A tree is a special type of graph which has many interesting properties and applications. It is defined as follows:

Definition 2.1.8. Tree

A tree (T) is a graph satisfying that between any two nodes (aka vertices) there is a unique path that connects them.

Notice, we refer to nodes instead of vertices in the case of trees. We use this notation in order

to make easier following the notation in this work, as in some cases we will talk simultaneously of tree nodes and other graphs vertices.

It can be proved that this definition implies that all nodes are connected, there are no loops (walks with same initial and final vertex) and that $|E| = |V| - 1$. This is proven in Theorem 2.2 of [9]. An example of tree is the following one:

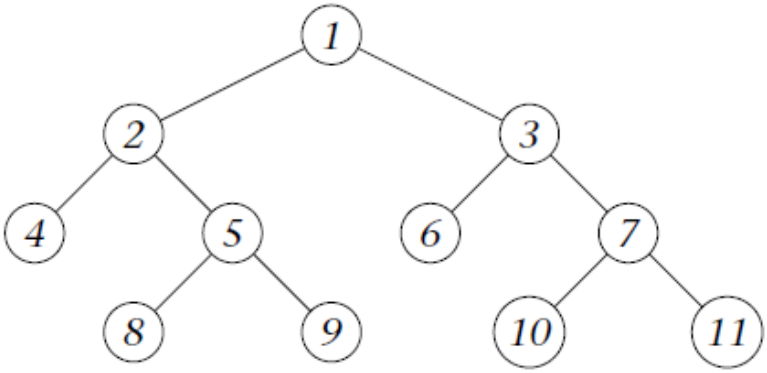


Figure 2.4: Example of tree graph.

We can distinguish two type of nodes, those with degree 1 and the rest. The ones with degree one are called *leaf* nodes and those with more are called *internal nodes*. In particular, the tree described in the previous figure is called a *rooted tree*:

Definition 2.1.9 (Rooted Tree)**.** Let $T = T(V, E)$ be a tree.

- T is a **rooted tree** if one of its vertices has been designated as the **root**.
- The **level** of a vertex, v of a rooted tree, is the length of the path which goes from the root to v. The level of the root is 0.
- Any vertex of a rooted tree that is not a leaf is an **internal vertex**.
- If in a rooted tree, v is the vertex that immediately precedes w on the path from the root to w, then v is the **parent** of w and w is a **child** of v.

Using a rooted tree allows us tree vertices and indices because there is a bijection between paths from the root to each vertex in the tree. This makes trees a structure specially useful to index structured data. Another relevant type of tree are k-ary trees:

Definition 2.1.10 (k-ary tree)**.** A k-ary tree is a tree whose nodes can have up to k child nodes.

In particular, 2-ary trees are also called binary trees.

Finally, we need to describe a special type of operation on trees: tree traversals:

Definition 2.1.11 (Tree Traversal)**.** A tree traversal is an operation that visits each vertex of the tree T exactly once.

These operations are used to iterate all vertices of a tree optimally, and can also be used to generate a global order for all vertices of trees. The different traversal types are defined by the order in which the nodes are visited. There are actually multiple possible orders and therefore types of traversals, but in this work we only consider the most relevant ones: pre-order, post-order, in-order and level-order traversal, which arise from iterating the vertices using *Depth First Search* or *Breadth First Search*. For the details of them, please refer to [9].

2.2 Tries

A Trie, T, of N nodes is a k-ary search tree (of N nodes) in which each edge has a label associated. Each node $n \in T$ represents a sequence of labels: the one obtained when traversing from the root to the target node. An example of Trie is the following one:

1 **rail**
2 **rain**
3 **rift**
4 **ring**
5 **road**
6 **rome**
7 **romb**
8 **row**
9 **rows**

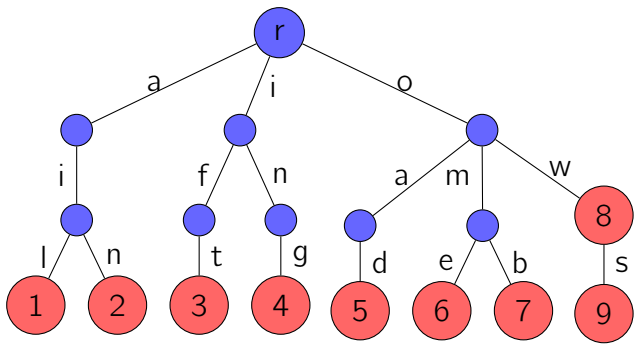


Figure 2.5: Example of Trie representing the list of words on the left side of the image.

The fact that the path between the origin and a node exists and is unique ensures that each sequence is labelled uniquely (bijection). In this way, for a given node, the sequence label represented by the node is a prefix to all the label sequences of the descendant nodes of the Trie. One of the main strengths of this data structure is its ability to exploit the shared nature of trees to index label sequences using an smaller representation. Tries the following operations basic operations:

Operation	Description	Average	Worst
getRoot	gets the root node	$O(1)$	$O(1)$
getChilds	gets a set of pairs of child node and label of the associated edge w.r.t. parent node $n \in T$, $\{(n_i, l_i)\}_{i=1}^{L}$	$O(k)$	$O(k)$
numChilds	returns the number of child nodes of $n \in T$	$O(1)$	$O(1)$
findChild	if exists, returns the child node of n whose associated edge has label $l \in \mathscr{L}$. Otherwise, returns null	$O(1)$	$O(k)$
insertChild	findChild(n,l) and if not found inserts a child to node n with label l	$O(1)$	$O(k)$
removeChild	does findChild(n,l) and, if node is found is leaf, removes it	$O(1)$	$O(k)$

Table 2.1: *Trie operations descriptions and time complexities in Average and in Worst Case.*

And have space complexity of $\mathcal{O}(|N|)$. This will always be equal or lower than storing each of those sequences separately. However, the particular number of N will depend on the composition of the label sequences contained in the Trie. This data structure can be adapted to multiple scenarios. In many cases, it is used to replace hash tables because elements can include in a collision-free manner. Relevant uses of tries are: as auxiliary data structures for storing, full-text search (as suffix tree), web search engines, DNA sequence alignment or internet routing.

A special type of tries are **Compressed Tries** (or **Radix Trees**) in which there are no unused single-child nodes. They can be built from any Trie by accumulating the labels sequences associated to consecutive single-child nodes. This results in a more compact representation of the Trie, i.e, with fewer nodes but encompassing the same set of sequences. An example of this is:

1 **rail**
2 **rain**
3 **rift**
4 **ring**
5 **road**
6 **rome**
7 **romb**
8 **row**
9 **rows**

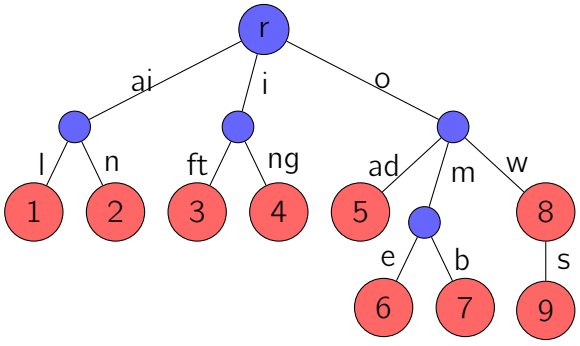


Figure 2.6: Example of radix tree (Compressed Trie) build from the one in 2.5. representing the list of words on the left side of the image.

2.3 Spatial and Road Networks

A defined in [1], a spatial network is a graph in which the vertices or edges are spatial elements associated with objects embedded in space. These type of graphs are used to describe cases where the underlying space is relevant and where the graph's topology alone does not contain all the information. It includes transportation networks (such as bus or airlines), infrastructure networks (road networks, power grid, ...) among others.

Among them, we are particularly interested in Road Networks graphs. Those are directed simple graphs which describe the possible decisions a traveler can make to go from its origin to its destination. Vertices represent the road segments (sections) where the traveler is, while the edges represents the possible changes of road segments (turns at intersections).

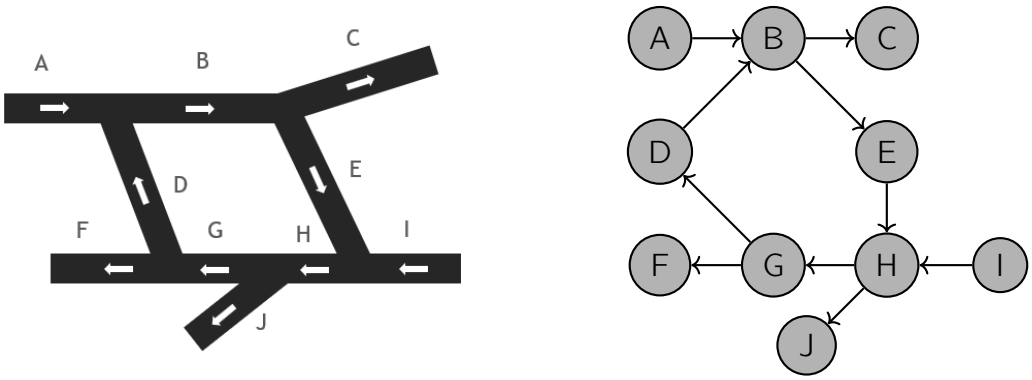


Figure 2.7: Example of road network (left) and its associated graph reprensetation (right). Labels A-J correspond to the different road segments in the network.

In road networks, the spatial embedding generates very particular features besides some of which are very relevant for this master thesis. Firstly, the spatial distribution of the nodes also determines the connectivity of the graph. Because the probability of two nodes being connected is inversely proportional to the distance among them. This produces the formation of *connectivity hubs*: clusters of relatively high degree vertices which are highly connected and even form cliques.

Those hubs are usually connected among them by longer road segments WITH higher capacity and lower degree vertices. This can be observed clearly in the following map of highways in Spain:

Those properties are translated into a degree distribution, $p(k)$, which tends to have a strong bias towards lower degrees with few nodes highly connected (hubs). This makes $p(k)$ a quantity of

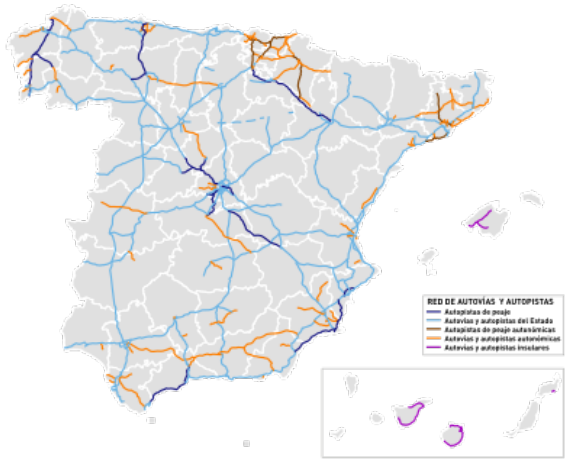


Figure 2.8: Spain highway road networks map obtained from Wikipedia. High capcity roads are coloured in this map and, as can be observed, there are connected hubs close to or at densely populated regions.

interest for this type of network, and empirical observations ([1]) show it approximately follows:

$$p(k) = \left\{ \begin{array}{l} \propto k^{-\alpha},\ if\ k \leq \hat{k} \\ \propto 0,\ if\ k > \hat{k} \end{array} \right\} \tag{2.3}$$

$\hat{k}$ is the maximum degree in the graph and $\alpha > 0$. Unlike other types of graphs, it exhibits a limit in number of connections. For instance:

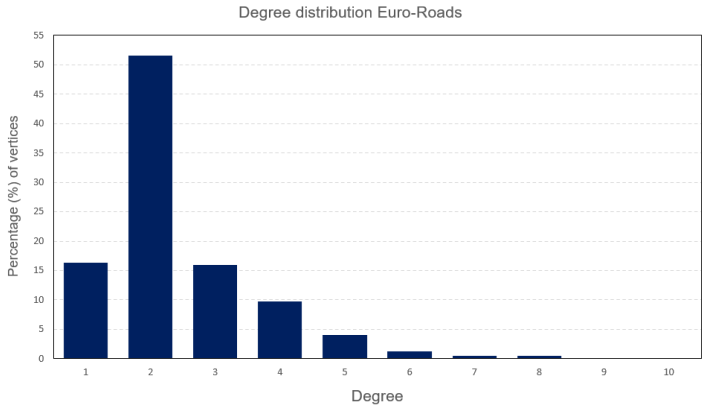


Figure 2.9: Example of degree distribution of european road networks based on the data extracted from Network Repository.

2.4 Data Compression

Data compression is the field of Computer Science that studies how to represent information using fewer data. The information is an abstraction that refers to anything that can provide knowledge when interpreted, and data is the medium in which this information is represented, which is measured in bits. The methods implementing compression and decompression are called: Encoder/Decoder. The following picture exemplifies this:

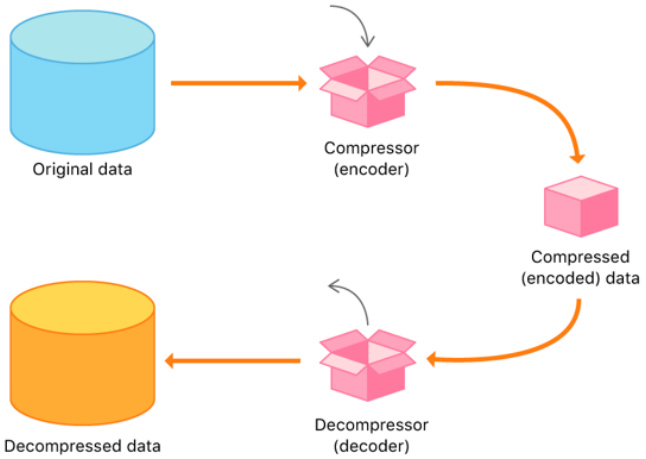


Figure 2.10: Compression/Decompression process description. Image extracted from Apple Compression for Developers.

The main objective of data compression is minimizing the size of the data, which represents some information. And there are multiple applications in which the size data occupies is problematic: multimedia compression, DNA sequencing storage, file compression, etc. However, the requirements for them may be different in one crucial aspect: whether information can be lost or not. This classifies algorithms in two main categories:

- **Lossless**: no information can be lost. This first category is described by the field of *Information Theory* , which was invented by Claude Shannon. This field attempts to quantify the amount of bits required to store a certain amount of information. For that, the Shannon-Entropy is defined as:

  Definition 2.4.1 (**Shannon Entropy**). Given X a random variable with $\{v_i\}_{i=1}^{N}$ possible values, with associated probabilities p_i, the Entropy of X is defined as:

$$H(x) = -\sum_{i=1}^{N} p_i \log_2(p_i) \tag{2.4}$$

This formula provides the way to know the minimal number of bits needed to represent a set of variables. This is stated in the *Source-Coding Theorem*:

Theorem 1 (**Source-Coding Theorem**). *Let $\{X_j\}_{j=1}^{M}$ be a set of independent identically-distributed random variables each with entropy $H(X)$. They cannot be compressed into more than $N \cdot H(X)$ bits without losing information in the limit $N \to \infty$.*

This result is theoretical, and it may not be achieved in real implementations of compression algorithms. However, it sets a common ground to compare the efficiency of compression algorithms.

- **Lossy**: information may be lost. Contrary, to the first category, the restriction is relaxed and instead of not losing information, a certain loss (distortion), R, is accepted. This is described by the *Rate-Distortion* theory. However, this theory is not as successful as the previous one because *distortion* is not an easily-measurable property. It usually depends on the device that extracts information from the data (f.i. human perception and the different image formats).

Multiple compression methods exist of each type, some of them better tailored to certain problems than others. To give some examples: *Huffman Coding*, *Arithmetic Coding*, *LZW*. Also, combinations among them are possible, such as *DEFLATE*, which combines *LZW* and *Huffman coding* and is one of the most effective and used lossless compression algorithms nowadays. It does so by defining a sliding window in which *LZW* dictionary is build and latter Huffman minimizes the code words associated to the words in this dictionary.

In the following subsections, we will give a brief overview on some of those methods and concepts related to this. For a more detailed explanation of those methods, please refer to [16] and [12].

2.4.1 Individual Variables Compression

In this subsection, we focus on those algorithms relevant for this thesis that compress individual symbols independently rather than sequences of them. In particular, two of the approaches used in this thesis: *Binary Coding* and *Unary Coding*.

2.4.1.1 Binary Coding

The binary number system consists on representing any natural number in 2-based system: representing any natural number ($n \in \mathbb{N}$) as a sum of powers of 2.

$$n = \sum_{i}^{\infty} b_i \cdot 2^i \tag{2.5}$$

When b_i is the i-th bit a value (0 or 1). A straightforward result is that the range of i can be reduced to that such that $n < 2^i$, as, any b_i larger than that would be only a 0 in the bit representation. In other words, with just B bits are used, all numbers from 1 to 2^B can be represented. Therefore, a more efficient way to store to express the previous relation is:

$$n = \sum_{i}^{B} b_i \cdot 2^i \tag{2.6}$$

Where B is the number of bits used to represent the number. It must be greater than $\lceil \log_2(n) \rceil$, which is the integer obtained from rounding up (ceiling) the result of the logarithm (which may not be an integer). For instance, the number 23 needs $\log_2(23) = \lceil \log_2(4.52) \rceil = 5$ bits to be expressed in binary:

$$23 = 1 + 2 + 4 + 0 + 16 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 \leftrightarrow (11101) \tag{2.7}$$

The number of bits used can be adapted to the range of sizes of the variables in use.

For the scope of this work, only integer indices from a set of R possibilities to be binary encoded/decoded. The following algorithms transform indices to/from sequences of 0s and 1s.

Algorithm 1 encodeNumberBinary

Input: n a natural number to be coded and R is the maximum value an index can take.
Let q be a natural number.
Let S be a sequence of bits.
while S.size < $\lceil \log_2(R) \rceil$ **do**
    S.append(n % 2)
    $n \leftarrow n/2$
end while
Output: S

Algorithm 2 decodeNumberBinary

Input: Let S be a sequence of $\lceil \log_2(R) \rceil$. Where R is the maximum value an index can take.
Let n be a natural number, from 0 to R.
Let b be a natural number.
Let S be a sequence of bits.
b ← 1
while $S.isEmpty()$ is false **do**
    b ← S.popLast()
    if b **then** == 1:
        $n \leftarrow n + b$
    end if
    $b \leftarrow b * 2$
end while
Output: n

In the algorithms above, we take the convention that the power-base of each bit is in descendant order.

2.4.1.2 Unary Coding

Binary format, assumed the range of values is known. When it is not the case, or if the range of values is much larger than the natural numbers we need to encode, other methods can be used: for instance, *Unary Coding*.

Unary Coding consists of storing a stream of bits in which a number n is encoded using $n-1$ consecutive 1-bits before a final 0-bit indicating the end of the bit sequence. Therefore, for storing number n requires n bits. For instance, the number 9 is encoded as:

$$9 \leftrightarrow 111111110 \tag{2.8}$$

Using a 32-bit integer for this number would be a waste in this case. Here it is clear, one of the advantages of this method: it is adaptive. The larger the number, the more bits it uses. The formal algorithms to encode and decode are:

Algorithm 3 encodeNumberUnary

Input: v an natural number value to be coded.
Let i be a natural number.
Let S be a sequence of bits.
i ← 0
while $i < v$ **do**
    S.append(1)
    i ← i+1
end while
S.append(0)
Output: S

Algorithm 4 decodeNumberUnary

Input: Let S be a bit sequence.
Let v be a natural number $v > 0$.
Let S be a sequence of bits.
v ← 0
b ← S.popFront()
while b == 1 **do**
    v ← v+1
    b ← S.popFront()
end while
v ← v+1
Output: v

When comparing this approach to the binary system, n v.s. $\lceil \log_2(V) \rceil$ is asymptotically extremely inefficient. However, if the range at hand is large (or unbounded), but values tend to be around the lower bound of the range. For instance, a range $[1, 2^{32}]$ where most values are below 10 is a suitable example of this.

2.4.2 Data deduplication

Data deduplication is the process of replacing the duplicated pieces of data by a shared copy plus reference to it. With this technique we can achieve significant reductions in the size of the data depending on the case. Consider, for instance, the paths of travelers on a graph. It is highly likely that many of those are repeated in densely-populated areas, in which people trips tend to people have the same origins and destination. Instead of storing multiple times the same path, a single instance of the path plus references to it would be more efficient.

This step can achieve a great reduction in size. How much will depend on the trade-off between and, that is, on the size of the pointer from the original positions to the shared copy of the data chunks.

As a side effect, deduplication can deeply affect the compression algorithm by changing the frequency of patterns that appear, avoiding overrepresenting of repeated sequences. This is exemplified in the following figure:

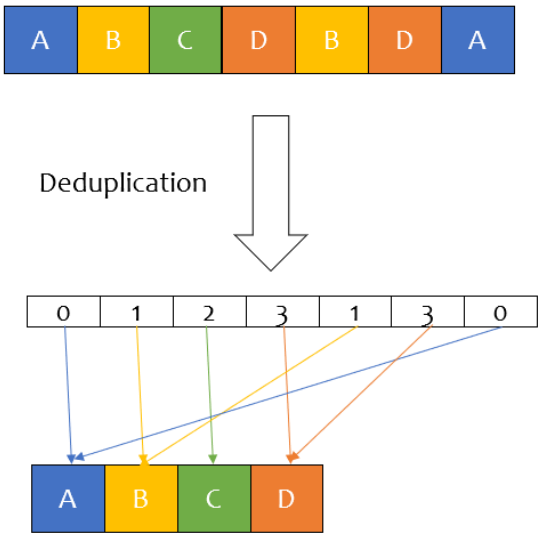


Figure 2.11: Example of data deduplication. As can be observed, all blocks can be stored only once in exchange of references to this block. This is optimal if the blocks are big enough for all cases too, except for C, which will be stored only once anyway but needs to store an additional pointer.

Chapter 3

Literature Review

This section attempts to explain in detail the main features of the most relevant works of literature used or references in this work. It is divided into three different sections, one covering each of them.

3.1 Adaptive Edge Offset Compression

Besides ideas in the academic literature, there is a simple, robust and effective technique IBM has patented. An adaptive compression method based on to compress paths on scale-free graphs ([14]). The approach represents paths as an edge offset sequence plus an initial vertex. The range of each of the edge offsets is determined by the vertex the edge belongs to. This is used to adapt the number of bits used to binary-code the edge offset sequence of the path. As each node of degree k can have an edge offset from 0 to $k-1$, the number of bits needed is $\lceil(\log_2(d)\rceil$, while for $d=1$ no bit is needed, because there is either none or a single possible outcome. Finally, the case $d=0$ is never realized, by contradiction: if a vertex has out-degree 0 the paths traversing it can only end there. The formal description of the encode and decode algorithms are the following:

Algorithm 5 encodeEdgeOffset

Input: A data stream which D, a graph $G(V,E)$, a vertex $v \in V$ and the edge offset l
$v \leftarrow G.getOutgoingDegree(v)$
if d is not equal to 1 **then**
    writeToBinary(D,l,d)
end if
Output:

Algorithm 6 decodeEdgeOffset

Input: A data stream which D, a graph $G(V,E)$, a vertex $v \in V$
$d \leftarrow G.getOutgoingDegree(v)$
if d is not equal to 1 **then**
  $l \leftarrow$ readFromBinary(D,l,d)
else
end if
Output: l

For instance, the edge offset 1 in a vertex of 4 out-edges can be coded as the bits 10 with 2 bits $0 \rightarrow 00$, $1 \rightarrow 10$ $2 \rightarrow 10$ $3 \rightarrow 11$. A more complex example is the one shown in the patent itself:

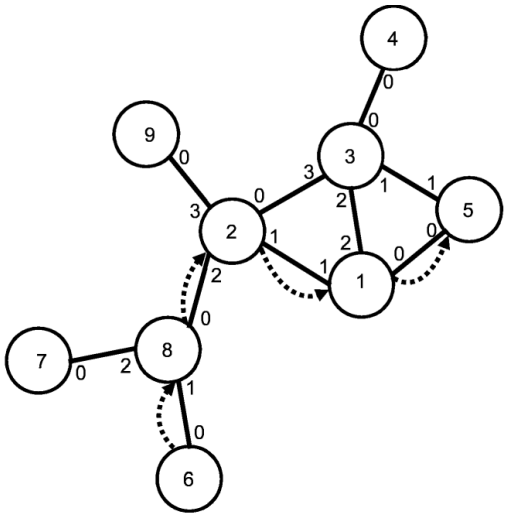


Figure 3.1: Example of adaptive edge offset compression. Edges show their labels as outgoing and incoming edges, and the dashed lines indicate the path considered. The edge offset sequence of this path is 0010. When encoding it, the first, second and fourth edge offset are not needed, because the out-degrees of the associated nodes are either 0 or 1. Only the third edge offset is coded, but using a single bit.

To encode/decode a sequence, we need to apply those algorithms and calculate the associated vertices recursively:

Algorithm 7 encodeEdgeOffsetSequence

Input: A data stream which D which can be written in bits, a graph $G(V,E)$, the initial vertex v of the edge offset sequence and sequence itself, H.
L $\leftarrow$ H.size
I $\leftarrow$ iterator at begin of H
Let w be a vertex of graph G.
encodeNumberUnary(D,L)
w $\leftarrow$ v
while I.hasNext() **do**
  I $\leftarrow$ I.next()
  I $\leftarrow$ encodeEdgeOffset(D,w,I)
  w $\leftarrow$ G.getOutgoingVertex(w,I)
  i $\leftarrow$ i+1
end while
Output: w

Algorithm 8 decodeEdgeOffsetSequence

Input: A data stream which D which can be read in bits, a graph $G(V,E)$, and the initial vertex v of the sequence.
L ← decodeNumberUnary(D)
w ← v
i ← 0
while i < L **do**
    l ← decodeEdgeOffset(D,w,G)
    H.append(l)
    w ← G.getOutgoingVertex(w,l)
    i ← i+1
end while
Output: (w, H)

This approach is specially well-suited for road networks. Since the degrees are on usually very low, except a few ones which are large.

As a final comment, notice that two additional pieces of data are needed to compress the whole path: the length of the path and the initial vertex. The authors propose using fixed-length variables to encode both of them, this step is actually depending on the available context of each situation as is not dependent on the properties of spatial graphs.

3.2 Compression of multisets of sequences

The efficiency of compression methods depends, in part, on the prior knowledge about the data being compressed. In some situations, this data is arranged in specific formats and specific methods can be designed to exploit it. This is the case of *Multisets of sequences*.

Definition 3.2.1 (Multiset of sequences)**.** A multiset of sequences (M) over alphabet Σ is a set of sequences of symbols of Σ.

This definition does not require sequences to be any specific order or have some relationship among them. The general problem of storing this (order-agnostic) set is what is known as the *bag-of-words problem*. This problem arises in many areas: natural language processing, information retrieval (IR) or computer vision.

Besides being free of choosing the order in which data is stored, solving this problem can also exploit another property of multiset sequences. Given that sequences are defined over finite alphabets, it is possible (and in some cases likely) that data deduplication can be applied over repeated subsequences.This opens the possibility of further reducing the size of the data, specially for smaller alphabets and/or larger sequences.

All in all, this makes tries a natural way to represent multisets of words. Because this data structure naturally deduplicates the prefixes of the sequences stored in them. And is, therefore, the common way in which data is organized before compression in the literature [6].

Using this representation, many methods are proposed to compress this data structure. The first one to do this was Reznik et al. [15], who proposed storing the multiset of words as a regular k-ary tree. Using enumerative coding to identify the trees, instead of storing the tree explicitly. Even though the space-saving is up to $\log(\Sigma!)$, the approach may underperform for non-regular trees.

In order to outperform Reznik's approach, [8], proposed an alternative way of storing multisets of sequences. In this case, the multiset of m binary sequences from n bits (a 2^n space) are used to build a binary Trie. Even if it was described for the binary alphabet, it can be generalized to any alphabet by converting each symbol individually to binary. The approach consists of building a Trie of the n sequences and converting it into a binary sequence formed by taking the differences in suffixes of each branch of the Trie with respect to the previous one in lexicographical order. The encoding step encoding takes $\mathcal{O}(m(n+\log(m)))$ steps. The decoding phase consists of undoing this conversion, which has a complexity of $\mathcal{O}(\Sigma n)$. One of the best results of this approach is that the compression ratio tends asymptotically with a large n up to $5/3$ of the estimated entropy for an independently and randomly distributed bag of m words. This work only considered equal-sized words, but the extension to multiple words with different length is expected to be straightforward.

3.3 Succinct Representation of Cardinal Trees

Borrowing the definition from [3], a Cardinal Tree is defined as:

Definition 3.3.1 (Cardinal Tree)**.** By a cardinal tree (or Trie) of degree k, we mean a rooted tree in which each node has up to k children and each child of a given node is labeled by a unique integer from the set {1, 2, . . . , k}.

This type of tree is generic and usable on any alphabet. For instance, a Trie collecting the words in a dictionary or one that models all possible sequence of choices over a finite set of possibilities.

Cardinal trees may be represented in multiple ways. However, the fact that each child of a node can be labelled to an ordered set allows some optimizations to be done in its representation. Different approaches have been developed to use this knowledge to avoid representing explicitly the tree structure: *succinct* representations.

Instead of representing a cardinal tree using explicit pointers between nodes, succinct cardinal tree representations make use of the order in which a traversal visits the nodes of the Trie. In this way, only the labels of the child nodes are stored explicitly. Here we only give an overview of them because all of them are discussed in detail [3].

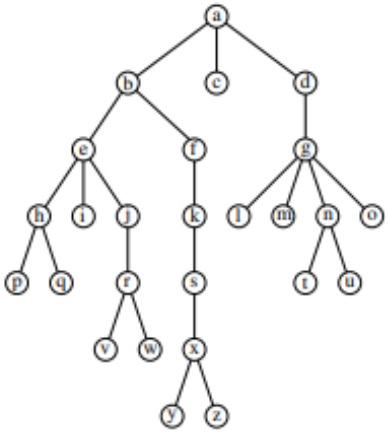
- **Level-Order Unary Degree Sequence representation (LOUDS)**. The number of child nodes of each node is encoded as a prefix of d+1 bits, d 1 and a 0 to mark the end (similarly to *Unary Coding*). On top of it, a second set of d bits is used to indicate if the i-th child out of the k possible child nodes actually exists or not. However, level-order (Breadth First Search) encoding scatters the node information across the encoding. Indeed, the higher the node, the farther apart their child nodes are. This hinders the performance of caching strategies in memory or disk.

- **Balanced Parenthesis representation (BP)**. Consists essentially on recording the information of the pre-order traversal (Depth-First Search) of the tree starting from the root. That is parenthesis, indicating if the move along the tree was up (")") or down ("(") and the Trie nodes visited. Its main drawback is that labels of child nodes are scattered across the encoding, accessing from the i-th label the next one requires traversing the whole subtree of i.

- **Depth-First Unary Degree Sequence representation (DFUDS)** The clever combination of the good of the previous approaches. This approach follows a pre-order traversal as BP however, at each node visited for the first time it codes the labels associated to the edge to its child nodes. Therefore, the labels of a node are always stored in a contiguously (good for caching) but keeps the node data belonging to the same branches close.

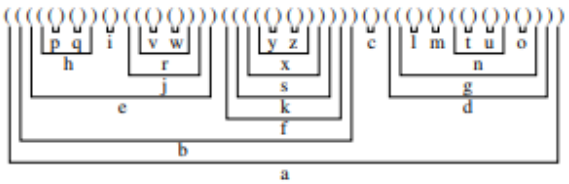
The space complexity of this algorithm is N symbols of the alphabet plus N counters indicating the number of child nodes of each node and its time complexity $\mathcal{O}(N)$. However, as explained, they have substantial practical differences which make DFUDS the best one among them.

While each method has its strong and weak points, the authors prove in [3], the DFUDS is the one providing the best space and runtime complexities. In the following figures, we can see an example of how each method encode the example cardinal tree:



(a) The Ordinal Tree

1110,110,0,10,1110,10,11110,110,0,10,10,
0,0,110,0,0,0,110,10,0,0,0,0,110,0,0



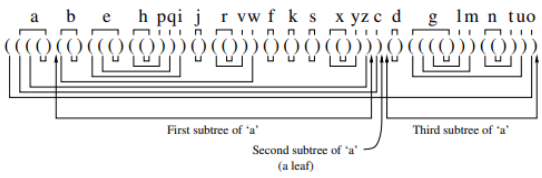


Figure 3.2: Cardinal tree example (top left) and the representation of the different ways to encode in [3]. *LOUDS* (top right), *BP* (bottom left) and *DFUDS* (bottom right).

Chapter 4

Problem Description

The problem considered in this dissertation is developing a generic representation of a collection of path data (Path Dataset) obtained from a road network graph, $G(E, V)$, which can be used to efficiently collect data once, store it and process it multiple times. Given this formulation, it is assumed no prior knowledge or auxiliary data is available on this data, besides $G(E, V)$. Such conditions are met in traffic simulation or when developing a scalable solution which collect and process path data.

Before proceeding any further, we provide a formal definition of what we call a Path dataset:

Definition 4.0.1 (**Path Dataset**)**.** A Path Dataset (P) is a collection of path-related data indexed by the description of paths realized on a graph $G(E, V)$.

Examples of Path Datasets are: the set of paths followed by commuters in a city in during a day, the paths followed by public transport users during a day or the set of airplane routes around the globe. All those example refer to Spatial Network graphs, however, we are only considering the Road Network type.

We consider that path descriptions are vertex sequences starting at origin and ending at the destination. And, for simplicity, we only consider the number of instance of a given path is considered as travel data. Extending this problem to arbitrary types of individual traveler data is straightforward.

The definition of *Path Dataset* emphasizes that traveler data associated to paths, rather than the paths associated to data instances. Understanding it in this way eases thinking of a single representation for path descriptions. The following figure exemplifies this:

Figure 4.1: Usual representation of travel data representation (left) and representation of Path dataset of travel data (right).

After discussing the concept of Path Dataset, we are prepared to delve in the details of ho Path Datasets are used. As previously explained, in our problem we consider Path datasets are generated by collecting paths once and consumed multiple times. The use of this pattern can be divided into three consecutive steps: collection, store & restore and process. The following figure represents each of them:

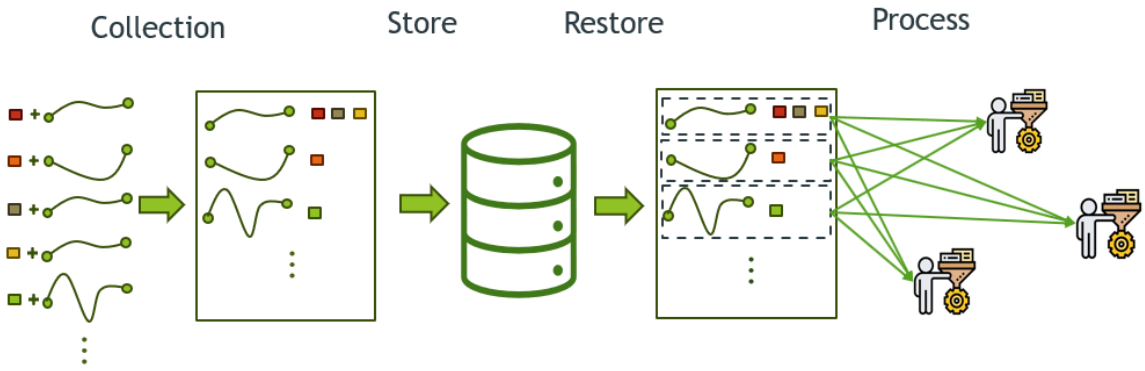


Figure 4.2: Representation of the different stages in the usage pattern of the Path Datasets. The coloured squared boxes represent the data associated to the path; the curved lines between circles.

More specifically, each stage may be described as follows:

- **Collection**. Creation, out of individual instance of path descriptions and the related data, the data structures representing the Path Dataset. Besides efficiency in time and space, the solution should allow the possibility of collecting paths in an undetermined order without compromising its efficiency.

- **Store & Restore**. Storing and restoring in long-term storage (disk) of the Path Dataset. The objective is providing a format in which data may not be accessible but occupies less space. This is stage is understood as an encode/decode a pair of algorithms. Both of them, should allow, by extension, encoding and decode a fraction of the whole dataset. To be able to process parts of the Path Dataset and avoid loading/unloading all of it.

  Regardless of the solution used, we have to consider also the possibility of using additional compression tools such as 7zip, WinRAR or libraries such as *ZLIB* together. And understand if those algorithms can further reduce the long-term storage size.

- **Process**: providing the path description of all collected paths and the associated data. With this, we assume that all paths restored want to be processed. Regarding order, we assume that as the data instead associated to each path is and independent, and the solution is free to use the most suitable order to iterate it.

4.1 Mitigation Strategies

In this section, we describe some generic approaches which may attempt to solve the problem and discuss why none of this problem for us.

- **Standard compression approaches**. Using common commands or tools (*DEFLATE*, Linux compress or WinZip among others) can greatly reduce the size of the data. However, this approach alone will not be able to dissipate the performance bottlenecks of the *Collection* and *Process* stages. Moreover, being general-purpose, they may not capture effectively the patterns and correlations Path Dataset exhibit.

- **Distributed representation**. A divide and conquer strategy. Instead of storing the whole dataset in the same device, it is divided in subset (probably segmented and organized). Each subset is stored separately (f.i. in different devices). This is a brute-force possibility which, in exchange, requires adding multiple devices. While this is always a possibility, it increases the cost of the application at hand and, therefore, should be a last resort.

- **Path processing on-the-fly**. By processing the paths as soon as they are collected and then releasing the resources representing them. However, while this approach can fit the single-collection multiple process pattern, it is not a practical way to tackle the vast majority of problems.

Chapter 5

Proposed Solution: Compressed Trie Indexation

In this section, we discuss the solution proposed to efficiently collect, store & restore and process a Path Dataset. Our proposal consists of using Compressed Tries (with *CT* as acronym) representing a collection of paths as edge offset sequences. To exemplify it, consider the following case in which the following set of paths:

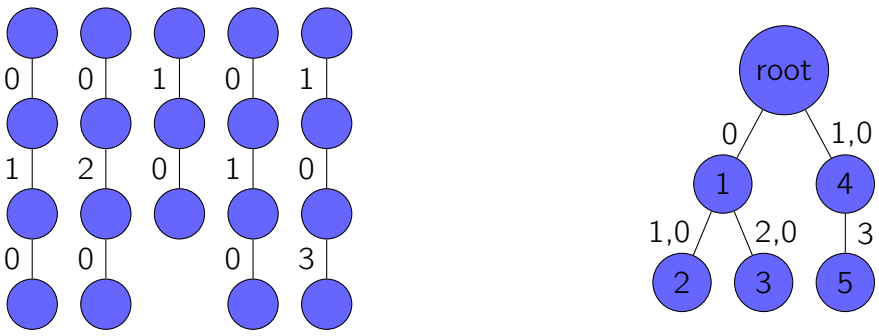


Figure 5.1: Collection of individual paths shown as a sequence of edge offsets (left) compared to Trie collective representation (right). In the first case 19 nodes and 14 edge offsets are represented, while in the second only 9 nodes and 8 edge offsets are.

We chose this design for several reasons:

- **Single description of realized paths**. If multiple identical paths are collected, they are mapped to the same node in the Trie. In this case, only an instance of the path description is stored.

- **Shared path description for overlapping paths**. Paths descriptions which overlap will share also the Trie nodes which describe them in their overlap. For instance, the sequences $\{0,1,0\}$, $\{0,1\}$ and $\{0,2,0\}$ in Figure 5.1. This will occur in Road Networks, because it is likely that travelers with the same origin have similar paths close to it: the number of possible paths is limited by space. This situation is symmetric to the destination.

- **Robustness due to Edge Offsets** As discussed in Adaptive Edge Offset Compression. Edge Offsets in Road Networks have usually a low range of values; bounded by the out-degree regardless of the size of the network. Paths represented as Edge Offset sequences will occupy the same regardless of the size of the network: unlike vertices, which are labelled from 1 to V (the number of vertices in the graph). Using Edge Offsets we can have efficient implementations which will be robust and admit any graph size.

- **Compact Edge Offset Sequences descriptions representation** As it is a Compressed Trie, each edge has associated a sequence of labels rather than a single one. This reduces the amount of vertices to be represented.

The Trie representation may be used for different groups of paths. At first sight, the idea of accumulating as many paths as possible in a single Trie is appealing because the more paths added, the more likely there will be some overlap. However, this also results in a loss of context: we will now less about the paths in the Trie. Which specially relevant for compression. Therefore, we consider three possible levels of grouping for tries:

- **None**

  In this case, the Compressed Trie may encode all the possible edge offset sequences. This maximizes the amount of data that is shared. However, we will need to represent the original vertex of each path explicitly. Moreover, the maximum number of child nodes will be undetermined (loss of context) and to compress or serialize the Trie we will need to assume this number is bounded by the maximum degree of the whole graph. For a road network, this number may significantly larger than for the vast majority of vertices, possible causing a significant downgrade of the Adaptive Edge Offset Compression efficiency.

- **Origin (or Destination)**

  This means grouping the paths collected in multiple Compressed Tries, one for each origin vertex (or, symmetrically, destination). Clearly, The total number of Trie nodes and edges needed to describe all tries would be larger than in the previous case. In exchange, we would only need to store the starting vertex once for all paths in each Compressed Trie. Additionally, in this case, each node in each Trie can be mapped to a vertex in the graph by just following the corresponding edge offset sequence from the root to the node. In this way, the context is always carried along and the Adaptive Edge Offset Compression compression is the optimal.

- **Origin-Destination**

  This option is the most disaggregated case. Therw would be a Trie for each origin-destination pair. However, this disaggregation does not provide any additional information we know how to exploit. Because, in a Trie, a branch splits, their child branches never join again. This is contrary to what occurs in paths with common origin and destination.

On the whole, which is the best option will depend on the Path Dataset in particular and experimentation should be carried out. However, given the discussion above, we think **the best choice and our choice is grouping tries by origin vertex**.

The discussion until now gives us an overall understanding of how paths will be represented in our proposed solution. In the forthcoming sections, we will discuss the details of this solution for each stage.

5.1 Collection

In this stage, tries are used to collect paths with a common origin vertex incrementally. The collection works in two consecutive steps: *Collection of Trie* the first one adding paths to a Trie, and *Trie Compression*, by building a *Compressed Trie*, in which there are no single-child nodes. The details of both steps are explained in detail in the next sections.

5.1.0.1 Collection of Dense Trie

The collection of each Trie can be decomposed in multiple calls of the operation *insertSequence*, which creates the required nodes to represent the input path and flags whether they represent a collected sequence or not (sequence node flag). This process can be further decomposed into finding the Trie nodes pending to add and adding them to the Trie (Trie.insertChild).

Algorithm 9 insertSequence

Input: the Trie T and the label input sequence S.
Q, n = findMissingEdges(T, S)
while Q is not empty **do**
  $l \leftarrow Q.pop()$
  $n \leftarrow T.insertChild(n, l)$
end while
Flag n as sequence node
Output: T

We define the first operation as findMissingEdges: find the sequence of missing edges in the Trie for a path with edge offset sequence $P = e_1, ..., e_L$ and a Trie T. This operation consists of iterating the path sequence forward and find which is the node associated to the maximal sub-path represented starting from the origin of the path. The output is the division of the path in what is represented in the Trie and what is not. The formal description of the algorithm is the following:

Algorithm 10 findMissingEdges

Input: the Trie T and the input sequence S. Let Q be a queue containing the labels of S_I in sequence order. Let n be a node of the Trie. Null if no such node exists.
$n \leftarrow T.getRoot()$
while n not null || Q not empty **do**
  $l \leftarrow Q.pop()$
  $n \leftarrow T.findChild(n, l)$
end while
Output: Q , n

It is clear that the complexity of the findMissingEdges and the calls to *insertChild* in algorithm *insertSequence* always sum exactly L, because the number of nodes in the Trie and those not are complementary sets and sum L. Therefore, the overall complexity of storing a set of N sequences of lengths L_i is $\mathcal{O}\left(\sum_{i=1}^{N}(L_i)\right)$ operations.

To exemplify the *Collection of Trie* step, consider the collection of the following sequences $\{0,1,0\}$, $\{0,2,0\}$, $\{1,0\}$, (again) $\{0,1,0\}$ and $\{1,0,3\}$. The sequence of operations would be the following, in **green**, the added nodes in each step, and, in **red**, nodes representing paths.:

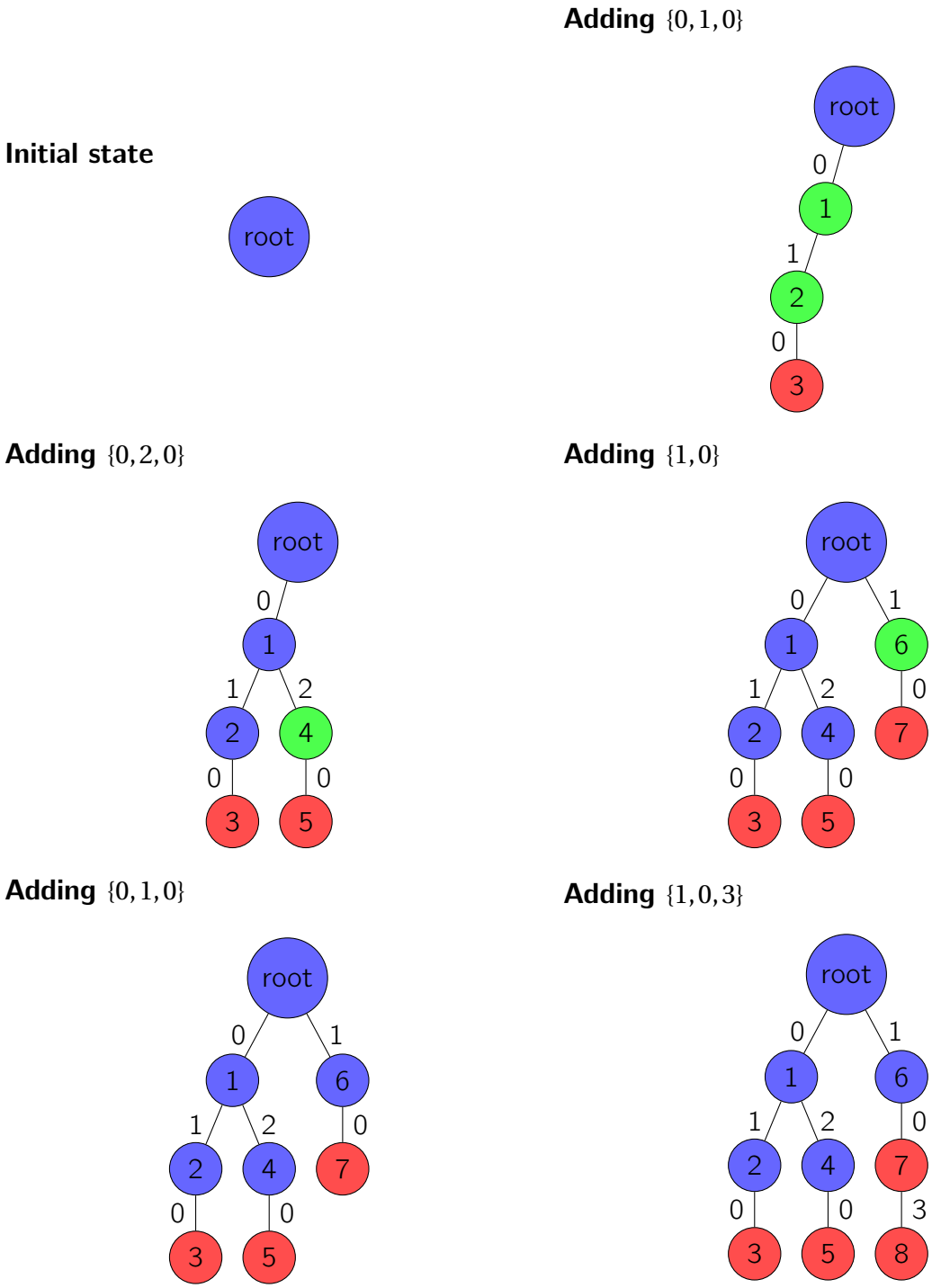


Figure 5.2: Exemplification of path-collection process on a Trie based on input sequences of Figure 5.1.

To implement those operations, we choose representing the Trie as a graph in CRS format and a vector with the data associated to each node. In this case, the data is just the times the path appears in the dataset.

The CRS format supports efficiently the addition of full nodes $\mathcal{O}(d+1)$ for a node with degree d. However, it doesn't support modifications of an already added node; as it may imply moving all elements in one of the destinations array, which can have a complexity of $\mathcal{O}(E)$ for a graph $G(V,E)$. And, in our case, we don't know the total number of child nodes for a Trie node until the end of the collection. Then, in order to avoid adding partial nodes and adding edges, we directly add nodes densely: by reserving space for all the possible child nodes (given by the outgoing edges of the graph node) and flagging edges which are not used in the Trie. Using this approach, the final state of the collected Trie of the previous example would be:

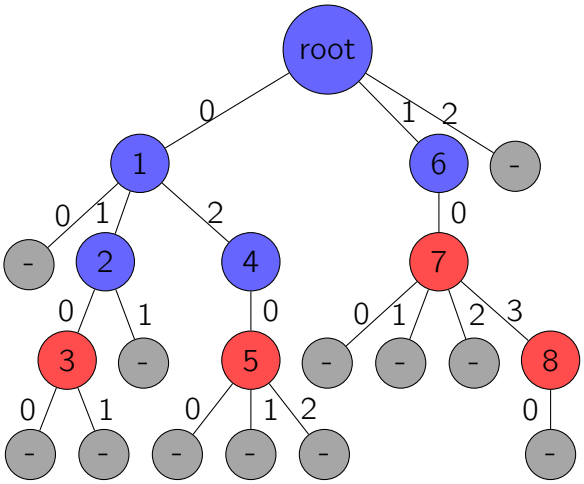


Figure 5.3: Representation of Dense Trie resulting from collecting sequences in Figure 5.2. In **gray** we have all those nodes that are not flagged as used. Nodes are labelled in addition order. As we can see, more nodes than needed will be added, depending on the possible child labels each node can have. Nevertheless, this inefficiency is temporary and will be handled in the Trie compression substage.

5.1.0.2 Trie Compression

In this subsection, we explain how to build a Compressed Trie out of a Dense Trie build. This process is essentially removing all redundant data after collection, without affecting the path data represented in it. For this, there are three pieces of data which can be removed:

- **Unused nodes**: Those nodes which do not represent a path may be included in hyperedges in order to reduce the size of the Trie.
- **Unused edges**: during the Construction of the Dense Trie. For each node, all its possible child nodes are pre-allocated, but some will not represent any edge offset sequence in the end.

With the objective of having a minimal representation of the Trie. Still, we have some freedom to aggregate nodes in different ways. In the following example, we present two ways of doing so for the Trie obtained in Figure 5.3:

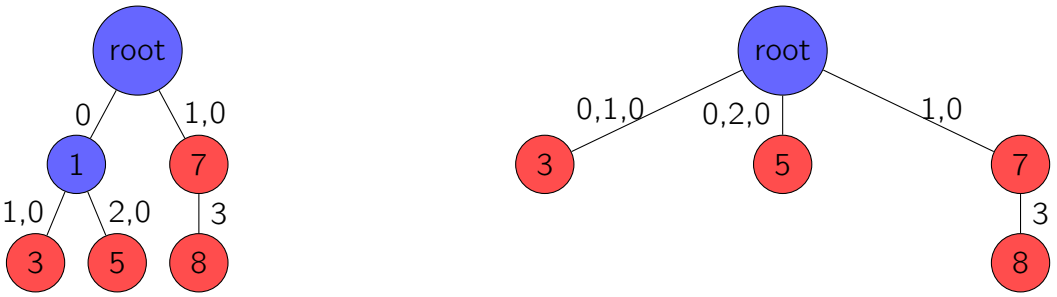


Figure 5.4: Representation of Trie compression alternatives based on Trie from Figure 5.1. The left option compressed unused nodes and edges, but only if they are single-child nodes. The right one does so regardless of the number of child nodes.

In both cases, we have removed redundant nodes and included them in redundant edges. The difference, however, is that in the right case, removing node "1" had two side effects: the root has one more child node and a label was duplicated. The first goes in detriment of the use (later) of the Adaptive Edge Offset Compression method, and the second may lead to a large repeated label sequences in the Compressed Trie. For this reason, we choose to compress using the left approach: only removing nodes that are unused and are single-child nodes.

After this discussion, we can finally define a Compressed Trie Hyperedge as:

Definition 5.1.1 (**Compressed Trie Hyperedge**)**.** Let T be a Trie, a Trie hyperedge h is an edge that connects a node v to one of its descendants w in T which fulfill the following conditions:

- v has more than one child.
- w has a number of child nodes different from one.
- h is a label sequence obtained from accumulating the labels along the path from v to w in T.

Applying this definition, we can define an algorithm that identifies hyperedges using the following algorithm:

Algorithm 11 collectHyperEdge

Input: the Trie T, s a node of T such that $T.numChilds(s) == 1$.
Let h be a label sequence
Let n be a Trie node
Let l be a label
$n \leftarrow s$
while T.numChilds(n) is 1 and T.numUsed(n) = 0 **do**
    n,l ← T.getChilds(e).front()
    h.append(l)
end while
Output: n, h

The complexity of this algorithm is exactly the size of H; the length of the sequence of only-child nodes plus one. By construction, all edges will be in a hyperedge. This algorithm collectHyperEdge can be combined with a Breadth-First Search in order to efficiently collect all hyperedges that would be built from a Dense Trie. This is the aim of the next algorithm:

Algorithm 12 collectAllTrieHyperEdges

Input: the Trie T and R a processor of hyperedges.
Let Q be a queue
Let v be a Trie node
v ← root
Q.enqueue(v)
while Q is not empty **do**
    $v \leftarrow$ Q.dequeue()
    for **do**w, l in T.getChilds(n):
        $n, h \leftarrow$ collectHyperEdges(v,l)
        h.prepend(l)
        Call R(v,w,h)
        if n is flagged as used **then**
            Q.enqueue(n)
        end if
    end for
end while

This algorithm will require $\mathcal{O}(M_u)$ steps to complete, where M_u is the number of used edges in the Dense Trie. Because it will necessarily traverse all the used edges of the Trie T in order to collect all the edges in hyperedges. In other words because $\sum_e \sum H_{(v,e)} = M_u$. Notice we could have also developed this algorithm as a DFS-like version and the result and complexities would be the same. However, the BFS approach will be more efficient for building a CRS-like Trie.

As for Dense Tries, we will represent a Compressed Trie as a CRS-like tree with the different that in this case, we require an auxiliary data structure to indicate the labels associated to edges of the Compressed Trie. The label of each hyperedge in this case is a sequence of edge offsets. In order to represent it, we also use a CRS format to have all sequences compacted in the

contiguous block of memory. The data layout of the Compressed Trie in (left) figure Figure 5.4 would be:

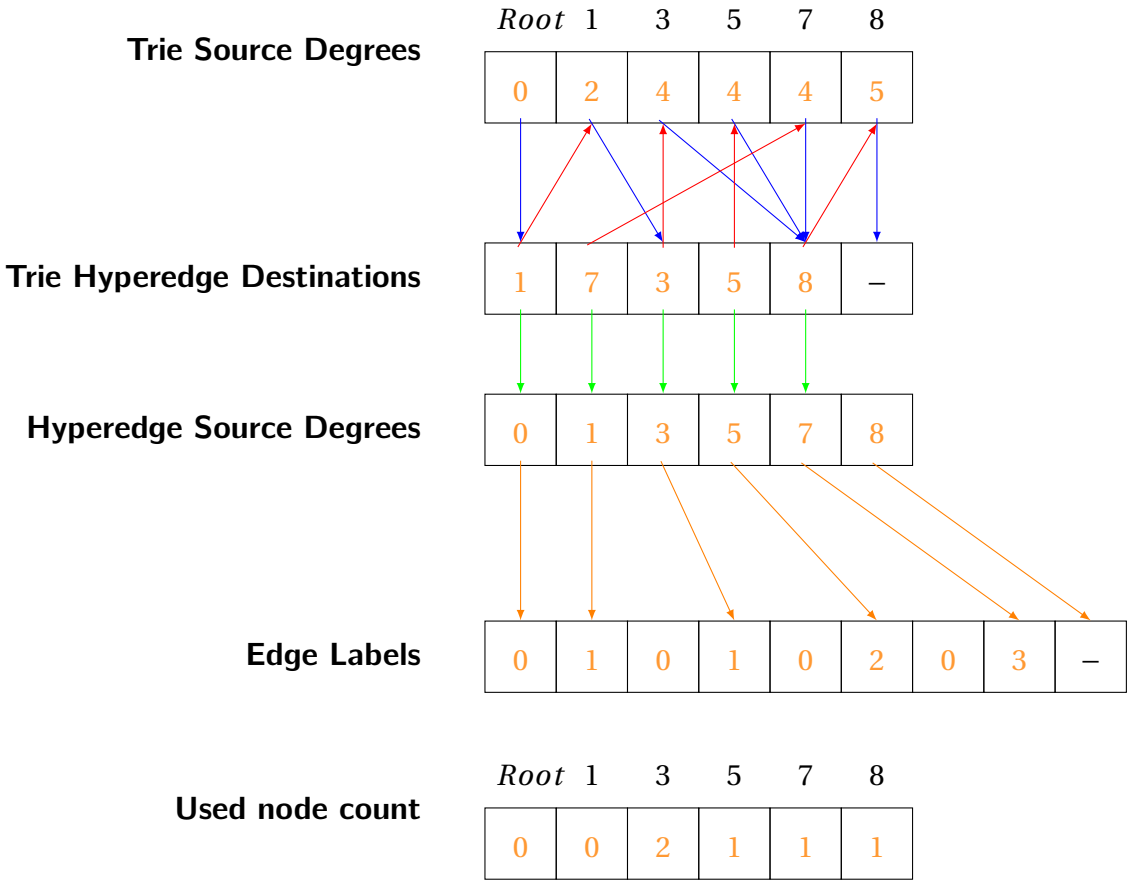


Figure 5.5: Example data layout of Compressed Trie.

Finally, the algorithm to build the Compressed Trie is a direct extension collectAllTrieHyperEdges for which, the observed $R(v, w, h)$ once it has collected all child hyperedges of v as with a normal CRS graph. Regarding the edge offset sequences of hyperedges, they are added consecutively in a CRS-like fashion to the *Hyperedge Source Degrees* and *Edge Offsets*.

Finally, the overall complexity of building a Compressed Trie is of M_u because it will consist of two parts: collecting the hyperedges which are already $\mathcal{O}(M_u)$ and adding the hyperedges to the Compressed Trie. This second part comprises: adding the hyperedges itself to the CRS graph, representing the tree *Hyperedge Source Degree* and *Hyperedge Destinations* which is in total $\mathcal{O}(H)$ and adding each of the hyperedge label sequences to the compacted sequence lists which is $\mathcal{O}(M_u)$. Therefore, the overall complexity of the Compression Trie is $\mathcal{O}(M_u)$.

5.2 Storage

In this section, we explain how our solution stores efficiently our proposal of Path Dataset. The storage, as in the *Adaptive Edge Offset Compression* approach, consists of storing the graph plus data describing the paths. This second part is further divided into two: storing the origin-based indexation and storing the Compressed Tries. Each aspect is discussed in a different subsection of the following ones.

5.2.1 Graph Storage

Storing the graph will depend on the application. The decision of the representation and format of the graph storage is up to the user and is then out of the scope of this work.

The only restriction on this decision is that, to use our implementation, that the graph used to store and retrieve the Path Dataset is identical: included the indexation of edges and nodes. Otherwise, our solution will lead to inconsistent results. This is a minimal and common requirement (see Adaptive Edge Offset Compression) that any implementation should fulfill. Because of this, we will not attempt to optimize its storage, and will directly use CRS graph encoded with fixed width variables (from C++).

5.2.2 Origin — Compressed Trie Storage

In this section, we discuss how stored all the groups of paths encoded as (origin vertex, Compressed Trie) pairs. As with the graph storage, part of the details are customizable. And because our approach splits the paths into origin-tries, different storage and retrieval strategies can be devised.

However, our approach is less sophisticated. It consists: storing these origin-Trie pairs in a Sparse way; serializing each pair consecutively. The layout would be:

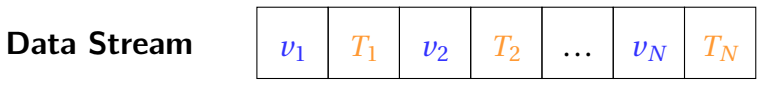


Figure 5.6: Data layour of origin-Trie pair storage.

An alternative would be storing a Dense index, where each entry would be associated to the i-th origin. It would be more efficient if most origins had a Trie associated, but we cannot

evaluate if that would be the case and if that approach is worth the effort. Whatever the case, remember this indexation is customizable, so it is up to a potential user to decide which is the most suitable one for their case.

Another possibility is using metadata to access only to a specific origin-Trie. As depicted in the following diagram:

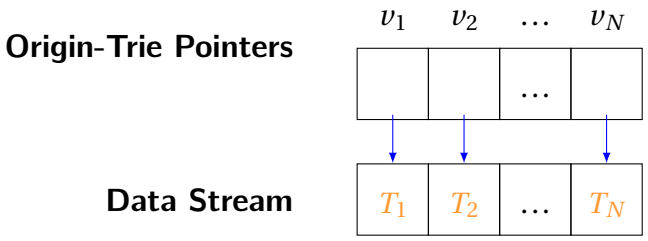


Figure 5.7: Data layout of origin-indexed storage of tries.

This grants the possibility of storing and restoring fractions of the whole Path Dataset.

Even more advanced indexation approaches could be used (such as [2]), but they are out of the scope of this work. The key aspect is that any of those strategies may be built from reading the whole Path Dataset (probably just once).

After this discussion on how tries and vertices are organized in the storage, we need to determine how to store origin vertices and Compressed Tries efficiently. The following two subsections covert each topic.

5.2.2.1 Origin vertex storage

We store the origin vertices using the order in which they are represented in V of graph $G = G(V,E)$. If there are $|V|$ vertices, we will need $\lceil \log_2(|V|) \rceil$ bits for each of them because we store them separately. Therefore, we would need to store $K\lceil \log_2(|V|) \rceil$ for K origin-Trie pairs.

5.2.2.2 Compressed Trie Storage

In this section, we will discuss how the Compressed Trie is stored efficiently with the path-related data. Firstly, we need to consider which information needs to be stored:

- **Trie Structure**: relationships between the Trie nodes.

- **Hyperedge label sequences**. Sequences of edge offsets.
- **Path data (Usage count)**: in our case, just the number of paths. But in a more general case, any data associated to it.

This is achieved by using a DFUDS-like approach, in which we customize the data stored for each node its child hyperedges and the usage count. We chose DFUDS approach instead of any other in section Compression of multisets of sequences. An extension of [8] would be a possible alternative. However, there are three main concerns with it. The first one is that it decontextualizes all the data by converting it into a binary tree and uses a single type of coding, ignoring the features of the data stored: number of child nodes, symbols of edges, length of hyperedges... The second one is that it seems the binary version of the Balanced Parenthesis approach, which is inferior to the DFUDS as discussed previously. The third one is that, DFUDS stores the sequences contained in the Trie in lexicographic order, and storing the data sorted can ease the compression of Trie in algorithms such as *DEFLATE*.

After this discussion, we can continue and describe encode/decode algorithms of our approach:

Algorithm 13 encodeCompressedTrie

```
Input:the Compressed Trie node T, the associated origin graph vertex r, graph G and an output data stream D:
Let n be Trie node.
Let v, w be graph vertices.
Let h, be a label sequence.
Let C be a tuple sequence of Trie nodes and the associated graph nodes.
Let I_C be an iterator over C.
Let S be a stack of I_C and w pairs.

i ← 0
S ← ∅
n ← T.getRoot()
v ← r
C ← T.getChilds( n )
I ← iterator of T.getChilds( n )
S.push( ( I, v ) )

while S is not empty do
    I,w ← S.top()

    if I.hasNext() then
        n,h ← I.next()
        v ← G.getOutgoingVertex(w,h)
        encodeCompressedTrieNode( D, G, T, n, v )

        if T.numChilds(n) > 0 then
            u ← encodeUsage( T.getUsage(n) )
            C ← T.getChilds( n )
            I ← iterator of T.getChilds( n )
            S.push( ( I, v ) )
        end if

    else
        v,I ← S.pop()
    end if

end while
```

The decoding algorithm is symmetric to this one. But, additionally, takes advantage of the fact that all the node description is encapsulated through encodeCompressedTrieNode in order to build the Compressed Trie on-the-fly as described in Trie Compression. The algorithm description is:

Algorithm 14 decodeCompressedTrie

Input: the origin-vertex associated to a (non-empty) Compressed Trie r, the graph G and an input data stream D.
Let n be a Trie node.
Let v, w be graph vertices.
Let h be a label sequence.
Let T be a Compressed Trie.
Let H be a set of hs.
Let C be a tuple sequence of Trie nodes and the associated graph nodes.
Let I_C be an iterator over C.
Let S be a stack of I_C and v pairs.

```
i ← 0
S ← ∅
n ← T.getRoot()
v ← r
decodeTrieHeader(D,G)
H ← decodeCompressedTrieNode(D,G,v)
T.insertChilds( n, H )
C_i ← T.getChilds( n )
I_{C_i} ← iterator at begin of C_i
S.push( ( I_{C_i}, v ) )

while S is not empty do
    I, w ← S.top()

    if I.hasNext() then
        i ← i + 1
        n,h ← I.next()
        v ← G.getOutgoingVertex(w,h)
        H ← decodeCompressedTrieNode(D,G,v)

        u ← decodeUsage(D)
        T.setUsage(n,u)

        if H is not empty then

            T.insertChilds( n, H )
            C_i ← T.getChildNodes( n )
            S.push( I_{C_i}, w )
        end if

    else
        S.pop()
    end if

end while
```

Output: T

As both algorithms are extensions of a DFS traversal, each of them run in $O(N_c)$ steps plus the call in each of those steps, the encodeCompressedTrieNode and decodeCompressedTrieNode and *encodeUsage* and *decodeUsage*, respectively. Therefore, the overall complexity will be determined once those functions are so.

A great advantage of this approach is that we can encapsulate all the information related to a node inside those two calls or at with an additional third one if a header is used to encode/decode common properties of the Trie. This opens the door to customizing the previous algorithms.

For the present case, *decodeUsage* and *encodeUsage* will only be used to indicate the number of repeated paths. We will encode it with encodeNumberUnary and decodeNumberUnary algorithms for simplicity and because, the marginal cost of adding a path is just one bit, which is the lowest possible value. However, there is possibility of using better suited approaches like Variable-Length-Quantity coding. However, the evaluation of more sophisticated approaches of encoding arbitrary numbers is left for future work.

The last piece of information missing in the coding of Compressed Tries are nodes and their child hyperedges. We can visualize for instance as:

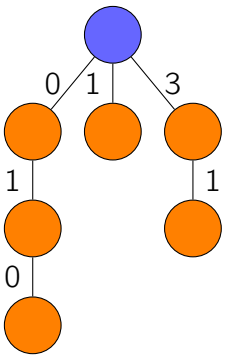


Figure 5.8: Representation of an isolated Trie node (blue) with three child hyperedges whose edge offset sequences are: $\{0,1,0\}$, $\{1\}$ and $\{3,1,2,3\}$.

In order to store the node and all its hyperedges we exploit the fact that the Compressed Trie represents a set of paths and each node can be mapped to a vertex in the graph that generated it. As a result of this, the number of child nodes of a Trie node can have is bounded by the degree of the vertex. Hence, we can apply Adaptive Edge Offset Compression in order to encode and decode the number of child nodes optimally using the following algorithms:

Algorithm 15 encodeSizeSubsetOutgoingEdges

Input: A data stream which D, a graph $G(V,E)$ AND a vertex $v \in V$. and the edge offset l
d← G.getOutgoingDegree(v)
writeToBinary(D,l,d+1)
Output:

Algorithm 16 decodeSizeSubsetOutgoingEdges

Input: A data stream which D, a graph $G(V,E)$ and a vertex $v \in V$.
d← G.getOutgoingDegree(v)
l ← readFromBinary(D,l,d+1)
Output: l

They are almost equal to encodeEdgeOffset and decodeEdgeOffset. The difference is because

edge offsets for a node with out-degree N range from 0, to $N-1$ while the size of a subset of outgoing edges goes from $\mathbf{0}$ to N.

Finally, we can describe the algorithms encodeCompressedTrieNode and decodeCompressedTrieNode. They are just the encoding and decoding of a list of hyperedge label sequences in the same order:

Algorithm 17 encodeCompressedTrieNode

Input: the Trie node $n \in T$, the Trie T, a graph G, the associated graph vertex v and a data stream D:
Let v, w be graph vertices.
Let m be Trie node.
Let h be a label sequence (edge offsets).
Let C be a sequence of Trie node and hyperedge label pairs.
Let P be a sequence of Trie node and graph vertices.

```
P ← ∅
C ← T.getChilds(n)
encodeSizeSubsetOutgoingEdges( D, I_C.numNexts(),v,G)
for (m,h) in C do
    w ← encodeEdgeOffsetSequence( D, v, h )
    W.append( ( m, w ) )
end for
Output : W
```

Algorithm 18 decodeCompressedTrieNode

Input: the Trie node $n \in T$, the Trie T, a graph G, the associated graph vertex v and a data stream D:
Let v, w be graph vertices.
Let h be a label sequence (edge offsets).
Let Q be a sequence of hyperedge edge offsets.

```
Q ← ∅
K ← decodeSizeSubsetOutgoingEdges( D, v, G )
while i < K do
    w ← decodeEdgeOffsetSequence( D, v )
    Q.append( ( w, H ) )
    i ← i+1
end while
Output : Q
```

Now that are pieces are in place, we are able to determine the complexity of encode and decode a Compressed Trie both in space and time.

Theorem 2 (**Encode Compressed Trie Time Complexity**)**.** *For a Compressed Trie of N_c and total number of edge offsets M, the time complexity of the encodeCompressedTrie algorithm is:*

$$\mathcal{O}(N_c + M)$$

Proof. The algorithm encodeCompressedTrie does $\mathcal{O}(N_c)$ steps. Each of those steps includes a call to *encodeUsage* and encodeCompressedTrieNode. The first one having a complexity of is

$\mathcal{O}(1)$. And the second one $\mathcal{O}(\sum_{i \in H_n} |h_i|)$ for each node n, where, H_n and h_i the associated label sequences of the i-th child edge. Therefore, the total complexity is:

$$\mathcal{O}\left(\sum_{n \in T_c}\left(1 + \sum_{i \in H_n} |h_i|\right)\right)$$

The statement of this theorem is obtained from simplifying this formula using $N_c = \sum_{n \in T}$ and $M = \sum_{i \in H_n} |h_i|$. □

Theorem 3 (**Decode Compressed Trie Time Complexity**)**.** *For a Compressed Trie of N_c and total number of edge offsets M, the time complexity of the decodeCompressedTrie algorithm is:*

$$\mathcal{O}\left(N_c + M\right)$$

Proof. Analogous to the proof for encodeCompressedTrie. □

Regarding spatial complexity, however, cannot be evaluated exactly because edge offsets are compressed using Adaptive Edge Offset Compression and because of the usage number being coded using *Unary Coding*. However, we can give an upper bound of without includes the usage counts, which can be arbitrarily big.

Theorem 4 (**Coding Compressed Trie Space Complexity**)**.** *For a Compressed Trie of N_c and total number of edge offsets M, the space complexity of the storing a Compressed Trie Node using the encodeCompressedTrie algorithm is upper-bounded by:*

$$N_c \lceil \log_2(d+1) \rceil + M\left(1 + \lceil \log_2(d) \rceil\right)$$

Where d is the maximum degree found on the graph G, used to describe the edge offsets coded in T.

Proof. The result is obtained from summing the contribution of each of the algorithms. The first term comes from the fact that each node needs to store the number of child nodes, which requires $\lceil \log_2(d_i + 1) \rceil$ bits, where d_i is the vertex of the graph associated to this Trie node. In total, this is:

$$\sum_i \lceil \log_2(d_i + 1) \rceil$$

Using the fact that $d_i < d$, the sum can be written as the first of this theorem statement.

$$\sum_i \lceil \log_2(d_i + 1) \rceil \leq \sum_i \lceil \log_2(d + 1) \rceil = N_c \lceil \log_2(d + 1) \rceil$$

The second term comes directly from encoding the edge offset sequences of each hyperedge. Each sequence will require the coding of its edge offsets and of the length, which, as we use *Unary Coding* is a bit per edge offset in the sequence. That means, each hyperedge sequence, h, requires:

$$\sum_{i=1}^{|h_i|} (1 + \lceil \log_2(d_i + 1) \rceil)$$

Using the fact that $d_i < d$, the sum can be written as the first of this theorem statement.

$$\sum_{i=1}^{|h_i|} (1 + \lceil \log_2(d_i + 1) \rceil) \leq \sum_{i=1}^{|h_i|} (1 + \lceil \log_2(d + 1) \rceil) = M \left(1 + \lceil \log_2(d) \rceil\right)$$

Using, again, the fact that $d_i < d$ we can simplify this expression as:

$$N_c \lceil \log_2(d + 1) \rceil + M \left(1 + \lceil \log_2(d) \rceil\right)$$

□

With the formulae shown in the previous theorems, we clearly see that all complexities related to the storage we propose depend directly on the size of the Compressed Trie, given by N_c and M.

5.3 Process

In this section, we explain how to process or visit all paths in the Path Dataset; using the algorithm iteratePath. Because paths are grouped by origin vertex in tries, we just need to describe how to obtain the paths contained in each of those tries. This operation implies visiting all nodes in the Trie, $T_c(N)$ and the vertex sequence $v_1, ..., v_L$ associated to each of them.

If we just want to extract the path of a single node, it can be achieved by recursively by taking the parent of each node until the root node and accumulating the labels of each edge traversed in a stack. At the end, popping the stack will generate the path (node sequence). This is described in the following algorithm:

Algorithm 19 iteratePath

Input: the Trie node T_c and a node m, the graph vertex r associated to the root and the graph G:
Let S be a stack of Trie T_c nodes.
Let P be a node sequence.
Let l be a label (edge offset).
Let h be a h sequence.
Let n be a node of the Trie T_c.
Let v be a vertex of the graph G.

$n \leftarrow m$

while $n \neq T.getRoot()$ **do**
    $n, h \leftarrow$ T.getParent(v)
    for $l \in h$ **do**
        $S.push(l)$
    end for
end while

$n \leftarrow$ r
while S is not empty **do**
    $l \leftarrow S.pop()$
    $v \leftarrow$ G.getOutgoingVertex(v,l)
    $P.append(v)$
end while

Output: P

Given an edge offset sequence of length L (that is a path of $L+1$ nodes) this algorithm requires L steps to build the stack and L to build the node sequence from the stack. Therefore, the total number of steps is, $\mathcal{O}(L)$, with at most, $\mathcal{O}(L)$, space complexity due to the stack. If this is used for multiple paths of length $\{L_i\}_{i=1}^{N}$, the time complexity will be $\mathcal{O}(\sum_{i=1}^{N} L_i)$ and the space complexity will be $\mathcal{O}\left(\{L_i\}_{i=1}^{N}\right)$.

However, this is clearly redundant if we need to apply these operations on more than one node.

Because when multiple paths overlap, the nodes on the overlap will be visited multiple times. Therefore, we can devise an accumulative-approach in which we minimize the times a node is visited and reduces this time complexity by exploiting those overlaps without increasing the spatial complexity. We extend pre-order traversal (DFS) to achieve this. By definition, during a pre-order traversal each node is visited exactly once. That is something that could also be achieved using the level-order traversal (Breadth First Search). However, the pre-order traversal, this ensures we only need a single stack to traverse all the paths needed. This is described in the following algorithm, which is a version of the DFS which uses a stack of iterators:

Algorithm 20 iteratePaths

```
Input: a non-empty Compressed Trie node T, the graph vertex r associated to the root, the graph G and path observer R.
Let v, w be graph vertices.
Let n be a Trie node.
Let S be a stack.
Let P be a node sequence.
Let H be a label sequence.
Let v be a node.

v ← r
P.append( r )
C_n ← T.getChilds(n)
I_{C_n} ← iterator at begin of C_n
S.push( (I_{C_n},v) )
while S is not empty do
    l,w ← S.top()

    if l.hasNext() then
        n,h ← l.next()

        for l ∈ h do
            v ← G.getOutgoingVertex(v,l)
            P.append( v )
        end for

        if T then.getUsage(n) > 0:
            Call R(P)
        end if

        C_n ← T.getChilds(n)
        if C_n is not empty then
            I_{C_n} ← iterator at begin of C_n
            S.push( (I_{C_n},v) )
        else
            for l ∈ h do
                P.pop()
            end for
        end if
    else
        n,h ← S.pop()
        for l ∈ h do
            P.pop()
        end for
    end if

end while
```

By virtue of the DFS each node iterator is acquired N_{childs} times, that is a total of $\sum_{v \in T} N_{childs(v)} = N$ and each node is pushed and popped exactly once to the stack: N pushes and N pops in total. The total complexity is $\mathcal{O}(N + N_u \cdot R(P))$.

Notice also that, when implementing this algorithm S and P can easily be implemented by the same data structure (one such as std::vector<> in C++) as the stack will have the node sequence that would be collected in P. Had we used the Breadth First Search algorithm, we would need to store partial path sequences until the leaves of the Trie were reached, and that would heavily increase the space complexity of this approach. This is essentially extracting most part of the edge offset sequences in the Trie simultaneously, which would be much less efficient in space.

An example of the steps of this algorithm follows using the Trie described in Graph Representations: CRS. In this case, the iterator of child edges of v is just the child node id:

Step	Stack (parent node, child node, length)	Path	Call $R(P)$
Initial	$\{root\}$	$\{X\}$	-
1	$\{(root,1,1)\}$	$\{X,0\}$	N
2	$\{(root,1,1),(1,3,2)\}$	$\{X,0,1,0\}$	**Y**
3	$\{(root,1,1),(1,5,2)\}$	$\{X,0,2,0\}$	**Y**
4	$\{(root,1,1)\}$	$\{X,0\}$	NA
5	$\{(root,7,2)\}$	$\{X,1,0\}$	**Y**
6	$\{(root,7,2),(7,8,1)\}$	$\{X,1,0,3\}$	**Y**
7	$\{(root,7,2)\}$	$\{X,1,0\}$	NA
8	$\{\}$	$\{X\}$	NA

Table 5.1: *Process operation on Trie example. X is the initial vertex. Notice that $R(P)$ is only called (Y) once for each used node and that there is no call to $R(P)$ (N) when traversing the node 1 because it was not used. The rest of cases are associated to pops done to the stacks and therefore, no possible call to $R(P)$ can occur (NA).*

Chapter 6

Experimental Design

In this section, we explain in detail the methodology and tools used to assess the solution proposed. That is validating and evaluating the performance both in time and space, considering the characteristics of the current Path Dataset and graph under study. The explanation of each of those aspects is discussed in different subsections.

In Reference Path Datasets Implementation, we discuss in detail the implementation of the implementation of Path Dataset used as references to assess the features of our approach.

Regardless of which representation is used to represent Path Datasets, we use the generic validation strategy discussed in Path Dataset Validation subsection. This consists on ensuring the Path Dataset collected and retrieved are the same regardless of the representation of the Path Dataset. Without it, any comparison or further analysis is, obviously, futile.

To do this comparison and evaluate metrics, we need to consider relevant use cases. On one side, we need to use Road Network graphs from real-world examples. This is done using data from Open Street Maps and explained in the section Graph Extraction From Open Street Maps. On the other side, we need to obtain Path Datasets based on this graph with different compositions (number of paths, length of those paths,...). This is discussed in the Synthetic Path Dataset Generation subsection.

Lastly, in the Measurements subsection, we discuss all the metrics designed to analyze the efficiency of our method. This includes basic performance metrics (computation time, memory, and size in disk) for each of the stages described in the Problem Description as well as characterization metrics to understand them.

6.1 Reference Path Datasets Implementation

As described in Proposed Solution: Compressed Trie Indexation, the two main aims of the design of our methods are deduplicating the data used to represent a Path Dataset and using different compression techniques in order to minimize the size of this data. This is mainly achieved by using *Compressed Tries* and, in serialization, *Adaptive Edge Offset Compression*, respectively.

In this section, we describe the implementations of *Reference Path Datasets* as Path Lists, which will serve us as baseline to assess the Compressed Trie format. For this, this section is split into the *Representation*, *Collection* and the *Storage Formats*.

6.1.1 Representation

In order to represent efficiently Path Lists and as similarly to the *CT* format, we also represent the lists in CRS format. The memory layout is analogous to the one shown in 5.5:

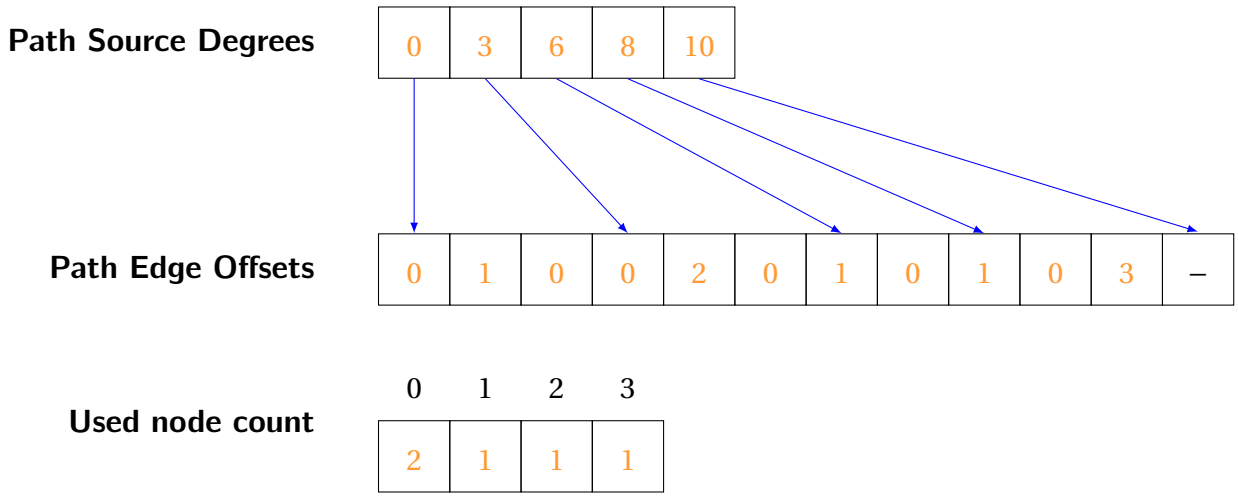


Figure 6.1: Example data layout of a Path List format built from the Paths in 5.2.

6.1.2 Collection

As explained in Problem Description, the *Collection* stage consists of accumulating path data and path descriptions which arrive individually into a Path Dataset, which, by definition, has a single description (entry) of each path. This consolidation is done naturally in tries, but for Path Lists we need to perform additional operations.

We split the *Collection* stage in an *Accumulation* step and a *Consolidation* step. In the former, paths and their data are collected and stored in a list, being possible that paths descriptions are duplicated. In the latter, using this path lists, all paths are sorted lexicographically. That groups equal path descriptions in $\mathcal{O}(N\log(N))$ for N collected paths. Then we just need to consolidate this data in $\mathcal{O}(N)$. This is exemplified with lists of integers in the following figure:

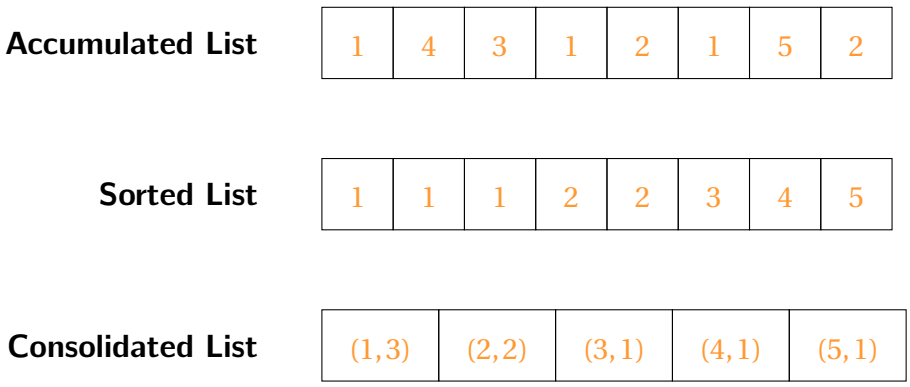


Figure 6.2: Exemplification accumulation and consolidation stages for Path List format using a list of natural numbers. In the *Consolidated List* we see the numbers appearing in the *Accumulated List* and their frequency

This can be then added to the Path Dataset efficiently. The alternative would be doing a linear search over all the paths every time we need to add a path which sums up to $\mathcal{O}(N^2)$.

6.1.3 Storage

In order to assess the *Storage* quality of our approach, we need to understand the impact of *Compressed Trie* format and *Adaptive Edge Offset*. And for this, different implementations are proposed in this section, which isolate the impact of each of those. The rest of aspects which are not evaluated are implemented analogously to what is described in Compressed Trie Storage:

- **Graph Storage**. Exactly the same as the one described in Graph Storage.

- **Origin Paths Storage**. In our approach, we store a single origin vertex and all edge offset sequences associated to paths starting at it. In this way, a single origin vertex needs to be stored.

- **Paths Storage** The approach used here is representing each path as an individual and independent sequence of vertices or edge offsets. This, together with the previous point, is essentially the base scenario on top of which we propose the Compressed Trie approach. Both of them can be jointly represented as:

$$\{v_i, k_i, \{n_j.e_j\}_{j=1}^{h_i}\}_{i \in \hat{V}} \tag{6.1}$$

  Where v_i represents the coding of vertex v_i in $\hat{V} \subset V$ and k_i is the number of paths starting at this vertex. Then $n_{i,j}$ is the coding of number of occurrences of path $j = 1, .., k_i$, both encoded using *Unary Coding* (encodeNumberUnary and decodeNumberUnary). Finally, $e_{i,j}$ is the coding of the edge offset sequences. The following is an example of this layout:

v_9	k_9	$n_{9,1}$	$e_{9,1}$	...	n_{9,k_9}	e_{9,k_9}	...	v_N	k_N	$n_{N,1}$	e_{N,k_N}	...	n_{N,k_N}	e_{N,k_N}

Figure 6.3: Representation of Reference Path Dataset serialized data layout.

Over this base description, we can build the different variants that will allow us understanding the efficiency of our approach. The variants are summarized in the following table:

Acronym	Path Order	Edge Offset Coding
PL UNSORTED FEO	Unsorted (randomly)	Fixed width variable: std::uint8_t
PL UNSORTED AEO	Unsorted (randomly)	Adaptive Edge Offset Compression
PL SORTED AEO	Sorted (lexicographically)	Adaptive Edge Offset Compression

Table 6.1: *Different variations of Reference Path Dataset Implementations.*

Among the Path List formats, the closest version to the features to *CT* (Compressed Trie) is *PL SORTED AEO* and is the one expected to have a more similar performance. On the other side, *PL UNSORTED FEO* and *PL UNSORTED AEO* can be used to evaluate the efficiency of Adaptive Edge Offset Compression. While *PL UNSORTED AEO* and *PL SORTED AEO* can be used to consider the impact of sorting when used together with the implementation of the DEFLATE algorithm from ZLIB.

6.2 Path Dataset Validation

The validation of the Path Dataset consists on ensuring the Path Dataset after storage and retrieval is the same for all approaches used. This is a compulsory test for any lossless compression method to be valid. In our case, the reference will be the Path Dataset generated in Synthetic Collection Profile Generation, while the case of target to compare will be the same dataset but in our proposed solution. To compare them, we will need to put them in a common format.

This format is a CSV, where each row codes a path and the usage count of this path (the data). The first column is this number, the second the origin vertex and the rest are the edge offsets of the path. In order to compare efficiently, all rows are sorted in Lexicographic order. In this way, the paths instances will be in strict total ordering. And we will be able to do a row-by-row comparison to determine if the reference output is equal to the original or not. An example of this format is the following:

$n_{3,1}$; v_3 ; 0 ; 0 ; 0 ;
$n_{3,2}$; v_3 ; 0 ; 0 ; 1 ;
$n_{3,3}$; v_3 ; 0 ; 1 ;
$n_{3,4}$; v_3 ; 1 ; 2 ;
$n_{3,5}$; v_3 ; 1 ; 1 ;
$n_{3,1}$; v_1 ; 1 ; 0 ; 1 ; 4;
$n_{3,2}$; v_1 ; 1 ; 0 ; 1 ; 4;
$n_{3,1}$; v_7 ; 0 ; 0 ; 1 ;
$n_{7,1}$; v_7 ; 1 ; 0 ;
$n_{7,2}$; v_7 ; 1 ; 1 ; 0 ;

Figure 6.4: Representation of Path List export format. Each row represents a realized path together with the number of times it is realized. All rows are sorted lexicographically.

The line-by-line comparison can then be done easily using tools such as KDiff3.

6.3 Graph Extraction From Open Street Maps

Open Street Maps is an open-data service that allows exporting map-based data from anywhere in the world. Its data is rich, accurate, and validated by a committed community, and it is used for many applications and research. For large regions, auxiliary Planet OSM and Osmosis are needed (and used) in order to extract filtered subsets of data. We use it in order to obtain road network data for the use cases in section Analysis of results.

Once this data is exported, we use NetworkX and GeonetworkX to import a directed graph and apply the following modifications to build a consistent road network graph with the information required:

- **Remove isolated vertices** . Those vertices cannot belong to any non-redundant path and probably arise from missing data in the OSM database.
- **Remove self-loops edge**. Those are spurious edge that cannot represent any path in a road network. They have not been observed directly but are preventively removed.
- **Collect vertex attributes** . Vertex attributes will be its geographical coordinates (latitude and longitude) and flags indicating whether vertices may be origin or destination. To decide which vertices are origins and/or destinations, we use the OSM edge property *highway*, which indicates the type of road. We assume trips endpoints are either *living_streets* or *residential*, consistently with OSM Documentation. Vertices which have outgoing access edge may be origins and those that have incoming access edges may be destinations of trips.
- **Collect edge attributes,** Only the weight is relevant, which will be the accumulation of the Open Street Maps ways lengths.

- **Contracting redundant vertices**. We define as redundant vertices as those with in-degree and out-degree exactly one and are neither origin nor destination vertices. Because they do not provide any information on the decisions made by travelers, as there is no other possible choice than moving forward. For instance, the one indicated as blue:

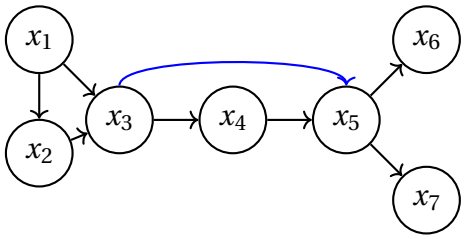


Figure 6.5: In this graph example can be observed, for trips not starting or ending at x_4 this vertex plays no role in this graph. Therefore contracting all verices along the blue arrow (x_3, x_4, x_5) in a single vertex reduces the size of the graph without losing any information.

This result translated to the coding of edge offset sequences, this contraction leads to the shortest sequences by removing edge offsets associated to 1-out-degree vertices. Therefore, it can only reduce the coded size of edge offset sequences. In the case of *Adaptive Edge Offset Compression* however, this will have no effect as edge offsets associated to 1-out-degree vertices occupy 0 bits.

The final format used as input to our processes is the In DOT format. Which is a generic graph format supported by many visualizers and graph libraries such as Boost Graph Library .

6.4 Synthetic Path Dataset Generation

A Path Dataset is obtained from collecting paths on a graph. Characterizing a Path Dataset from real traffic is a complex task which depends on the geometry, land use (f.i. population density), destination choice and route choice strategy of travelers. And, unfortunately, we do not have access to this type of data and in the amounts and variety we would need.

Our approach is generating realistic Path Datasets synthetically. Besides overcoming this limitation, this approach also adds the possibility of sampling sensible paths datasets from different scenarios and, thus, having Path Datasets of different compositions. In this way, we can extract conclusions which are more robust than using a small sample of Path Datasets. The main challenge of this approach is sampling cleverly over the large number of possible Path Datasets which can be realized on the same graph.

We divide this approach into three consecutive steps: synthetic origin-destination generation, synthetic path generation and synthetic collection order generation.

Figure 6.6: Overview of the steps involved in Syhnethic the Path Dataset Generation. As can be appreciated, the output of OD generation is an input for Path Generation. And the output of the latter is required to produce a Collection profile.

Each of the steps will be discussed in detail in the forthcoming sections.

6.4.1 Synthetic Origin-Destination Generation

As with Path Datasets, real origin-destination data is not available in this thesis, mainly because we have not found a free source of this data. The reason is likely that either complex models or large surveys need to be carried out to obtain it. All of this makes it a valuable asset for transport consulting companies and Departments of Transport.

Instead, we generate this data by selecting non-repeated origin-destination non-repeated pairs of vertices out of the possible origins and destinations. As explained previously, we only consider those vertices flag accordingly as origins and destination. And then, using a uniform distribution to select the origin/destination vertices

$$p(v_i, v_j) = \left\{ \begin{array}{l} 0, \ if \ i = j \\ \frac{1}{|V|(|V|-1)}, \ otherwise \end{array} \right\} \tag{6.2}$$

This uniform probability has the disadvantage of not accounting for the distance between origin and destination or their location. In spite of this, this approach allows us having a Path Dataset with the same number of paths per origin, on average. And this homogeneity eases the analysis of the results. Additionally, some filtering is used in order to discard unrealistic paths, and control the amount and similarity among them:

- **Distance Filtering** (λ): by calculating the length between origin and destination if the distance is below the minimum, λ. This minimum models, how eager are travelers to walk, as for shortest distance, the main transportation mode is walking.
- **Geographical Filtering** (r): by determining a circle of radius r from a reference point, p, we determine the candidate origins and destinations that may be eligible as origin and destination. With this we can influence the possible number of origins and destinations and the number of vertices in a path.
- **Density Filtering** (ρ, κ): by setting a fraction, ρ, we select the number of OD pairs considered out of all the possible ones and control the similarity of paths in the dataset. Complementary to this, we can also use a maximum number of queries κ for this purpose. Depending on the case, one or the other may be more useful for describing the situation in mind.

Those filters allow us to control the composition of Path Datasets with just three meaningful parameters r, λ and ρ. The only care we need to take is ensuring $\lambda < 2\rho$, otherwise there will be no origin-destination pair.

6.4.2 Synthetic Path Generation

Once the origin-destination targets are known, the paths need to be generated and collected. This is done by calculating the shortest paths between the pairs of the previous vertices. This is an approximation as mentioned in [17] and [1], as travelers tend to minimize their travel costs, which are, in general, related to distance. Nevertheless, users do not always follow the exact shortest path, but may follow variations of it. In order to emulate this, we introduce some noise in the costs using a truncated Gaussian Distribution on the weights of each edge:

$$p(x) = \left\{ \begin{array}{l} 0, \, if \; x \leq 0 \\ \frac{1}{\sigma\sqrt{\pi}} e^{\frac{-(x-\mu)^2}{2\sigma}}, \, if \; x > 0 \end{array} \right\} \tag{6.3}$$

Where x is the weight of the edge, which cannot be negative. Where μ is the average cost of the edge and σ the variance of the costs and will be coded as $\sigma = \alpha\mu$. Where α is called the *deviation*, a common factor to all edges for simplicity.

6.4.3 Synthetic Collection Profile Generation

The theoretical analysis, suggests the order of collection does not have an impact on the algorithm complexity. However, in practice because this may not be true due to two factors: cache misses and memory allocations. Both of them penalize runtimes and, thus, characterizing their effect of the collection order is necessary. Because in a real-world use case, we can expect paths will be collected in an undetermined order.

For this, running multiple performance measurements on the same Path Dataset is necessary but collecting the paths in different orders. In order to achieve it, our approach consists on randomly shuffling the input paths. This is done by combining the algorithm std::random_shuffle and the std::random_device to provide a pseudo-random order from the C++ Standard Library in the implementation.

6.5 Measurements

In this section, we describe meaningful metrics to evaluate if our Proposed Solution: Compressed Trie Indexation is acceptable given our Problem Description. Therefore, we need metrics to measure the performance of each stage of the problem and other metrics to characterize the scenario under study.

Those measurements will sometimes be affected by processes out of our control. This is specially true for performance measurements, as nowadays computers perform multiple processes concurrently. In those cases, multiple measurements should be taken and their variability assessed. Due to this, we will consider taking their average and standard deviation. For derived measurements, the deviation will be calculated using a simplified version of *Propagation of Errors* (formula 2.10 of [11]).

6.5.1 Characterization Measurements

These measurements will be used to characterize the use case at hand. We aim to provide a set of measurements that allows to explain all the behaviors observed in the performance measurements, at least qualitatively. In this section, we provide the formal definitions for each of them. They are divided into Graph Measurements, Path Dataset Measurements and Compressed Trie Measurements.

6.5.2 Graph Measurements

- **Number of vertices**. Number of vertices in the graph.
- **Number of edges**. Number of edges in the graph.
- **In-degree distributio**n. Histogram of the number of vertices of each possible in-degree in the graph.
- **Out-degree distribution**. Histogram of the number of vertices of each possible out-degree in the graph.
- **Degree distribution**. Histogram of the number of vertices of each possible degree (in-degree plus out-degree) in the graph.

6.5.2.1 Path Dataset Measurements

- **Number of different paths**. Total number of different paths contained in the Path Dataset.
- **Path Size**. The number of edges of the realized path sizes in the Path Dataset.

6.5.2.2 Compressed Trie Measurements

We defined a single measurement for Compressed Tries to understand qualitatively the reduction of size by using a Trie representation compared to an individual representation of sequences (paths):

Definition 6.5.1 (**Overlap**)**.** The Overlap, $\theta \in [0,1]$, of a Compressed Trie, $T(P)$ representing a set of paths $P = \{p\}_{i=1}^{|P|}$ with common origin, is the ratio over the number of edges of the Trie over the sum of the number of edges of each individual path:

$$\theta\,(T(P)) = \; 1 \; - \left(\frac{e_{T(P)}}{\sum\limits_{p \in P} e_{T(p)}} \right)$$

Where e_T is the number of edges of Trie T.

When the $\theta \simeq 1$, the number of edges in the Trie are much fewer than those needed to represent each path separately. On the opposite case, $\theta \simeq 0$, the Trie representation requires almost the same number of edges to represent paths. In the extreme case, $\theta = 0$, we find, for instance, when there is a single path, in which case using a Trie or a list of individual paths has the same size. The fewer labels a Trie requires, the more efficient it is compared to a Path List.

However, in order to make the analysis of this measurement practical, we will consider its aggregate version for a Path Dataset which, in general, contains many tries:

Definition 6.5.2 (**Total Overlap**)**.** The Total Overlap, $\theta \in [0,1]$, of a Path Dataset ($\mathscr{P}$) composed of a Compressed Trie $T(P_o)$ for each realized vertex origin $o \in O$ which represents all paths starting at this vertex P_o is:

$$\theta\,(\mathscr{P}) = \; 1 \; - \left(\frac{\sum\limits_{o \in O} e_{T(P_o)}}{\sum\limits_{o \in O} \sum\limits_{p \in P_o} e_{T(p)}} \right)$$

Where e_T is the number of edges of Trie T.

6.5.3 Performance Measurements

The performance measurements to be considered in this work are direct measurements of time and space. The performance measurements will be the following ones:

- **Duration (ms)**: time elapsed between start and end of stage. Implemented using std::chrono.
- **Memory Usage (bytes)**: difference between the memory used by the application at the beginning and the one at the end of the stage. Implemented using the PSAPI.
- **Size (bytes)**: size of file storing the Path Dataset in disk. Only applicable to the Storage Stage, including their combination with *DEFLATE*. Implemented using std::filesystem.

Those measurements will be applied to the different stages described in the Problem Description. However, a distinction must be made for the *Collection* stage. This stage is actually divided in two for both Compressed Tries and for Path Lists into an *Accumulation* and *Consolidation* steps. In the case of Compressed Trie, the *Accumulation* is the Collection of Dense Trie and in the *Consolidation* is the Trie Compression. Therefore, a single set of measurements for the whole *Collection* stage will not suffice, and we will measure the performance of those steps too.

Chapter 7

Analysis of results

In this section, we analyze the results obtained from applying the Experimental Design to the selected use cases. We have chosen to work on the Metropolitan Area of Barcelona. The reasons behind this decision are the scale, the heterogeneity of the graphs described, as it exhibits urban and interurban traffic networks. And also the fact that is a densely populated region and prone to have a mobility issues such as congestion. Therefore, a target region where path data can be used to solve those problems.

The analysis is split in two different kinds of experiment. A **parametric analysis** in some chosen regions in order to understand how our solution performs against the approaches in Reference Path Datasets Implementation. And a **scale analysis**, where we choose the Valles region in order to estimate in a larger and interurban case which are the expected scales of the performance measurements.

All tests and experiments are carried out with a computer provided by Aimsun S.L.U with the following specifications:

Specification	**Description**
Manufacturer	Dell Inc.
Model	Precision 5550
Processor	Intel Core i9-10885H CPU @ 2.40GHz (16 CPUs), 2.4GHz
Memory	64 GB RAM Dual Channel DDR4 (2666 MHz)
Disk	PC601A NVMe SK hynix 512 GB (SSD)
Operative System	Windows 10 Enterprise 64-bit (10.0, Build 19045)

Table 7.1: Specifications of the computation setup used for the experiments run in this master thesis.

7.1 Case Study: Barcelona Center Area

This section is divided in two sections: the Graph Description, where we describe the features of the graph used to generate the Path Datasets. And the Parametric Analysis, where we analyze of the different approaches considered in this thesis, for different scenarios.

7.1.1 Graph Description

The specific road network selected is the one defined by the following bounding box extracted from Open Street Maps:

Figure 7.1: Bounding box of the BCN region road network under study. The path queries considered are only the ones between origins and destination within the circle of 3km radius around the point in red located at (latitude,longitude)=(41.40,2.17). This region is defined as it is in order to mitigate the computation burden of the parametric analysis.

The graph it comprises (after simplification) 33,913 vertices, with 8,690 vertices as origins and 8,694 vertices as destinations, and 68,484 edges.

The degree distributions before simplification are the following ones:

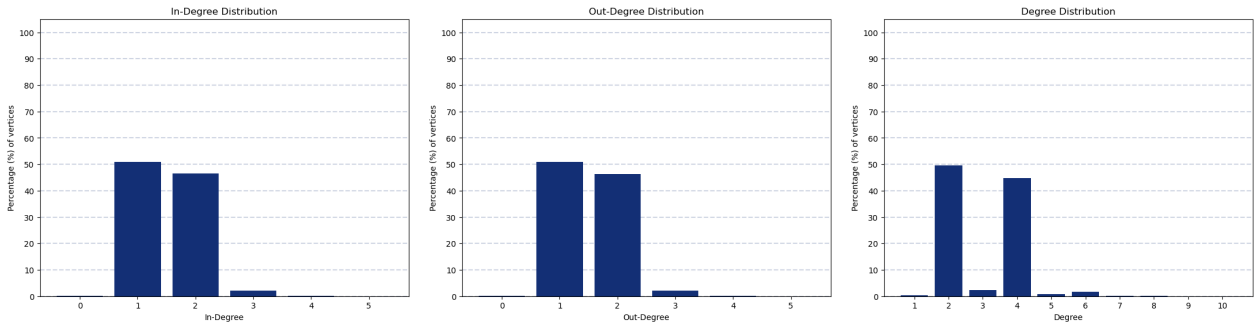


Figure 7.2: Histogram of vertices degree of the BCN road network graph prior to contraction of redundant edges.

After contracting redundant nodes, the distributions become:

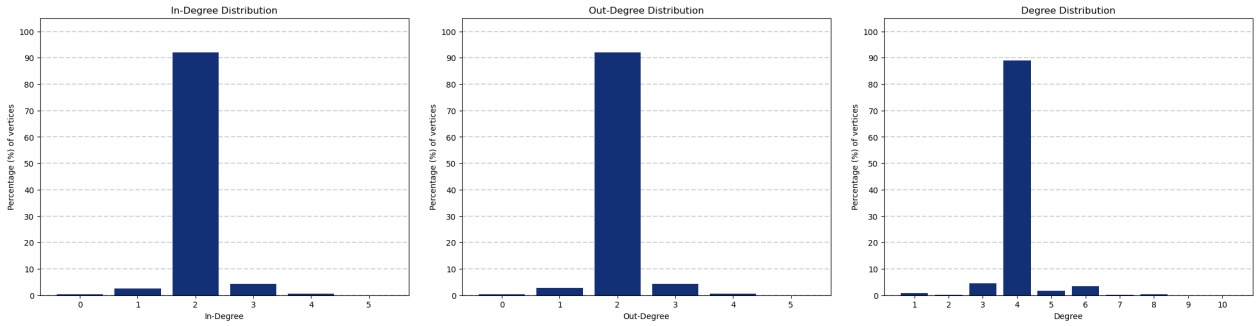


Figure 7.3: Histogram of vertices degree of the BCN road network graph after contraction of redundant edges.

The contraction step provides the expected results, eliminating many single-incoming and/or single outgoing edges and shifting the degree distributions to 2-in and 2-out degrees.

7.1.2 Parametric Analysis

As already commented, the Path Datasets are generated synthetically as explained in Synthetic Path Dataset Generation. Given the number of parameters controlling path generation, the number of combinations and, in consequence, the runtime of those experiments may be excessive. For this reason, we have chosen to use predefine values for some parameters we expect not to give relevant insights. This is summarized in the following table:

Parameter	Pre-defined value
Number of Path Dataset samples	10
Number of Batches	10
Deviation	5 %
Minimum Distance	1 km

Table 7.2: *Parameters set to fixed pre-defined values for Case Study: Barcelona Center Area, Parametric Analysis.*

We chose the Number of Path Dataset samples, number of Batches and the Deviation to be fixed to those values because, manually testing our application, we found it induces some variability in the paths found without incurring in an excessive computational burden. We also assume travelers will choose the option to walk if the origin and destination are below 10 min close, which is corresponds to 1 km at a normal, 5 km/h pace. Even if those values are suitable enough for our case, a more rigorous and exhaustive analysis should be carried out. In order to ensure the bias in the results is not relevant.

In any case, this leaves us with the parameters controlling the *Distance Filtering* (r) w.r.t a certain point and *Density Filtering* (ρ,κ), which help us condition the three main measurements: **Total Overlap** and Path sizes.

In order to characterize the Path Dataset, we evaluate the behavior of the characterization metrics for different parametrization and see the way in which the different metrics evolve. We do it by generating random Path Dataset, fixed r but varying ρ:

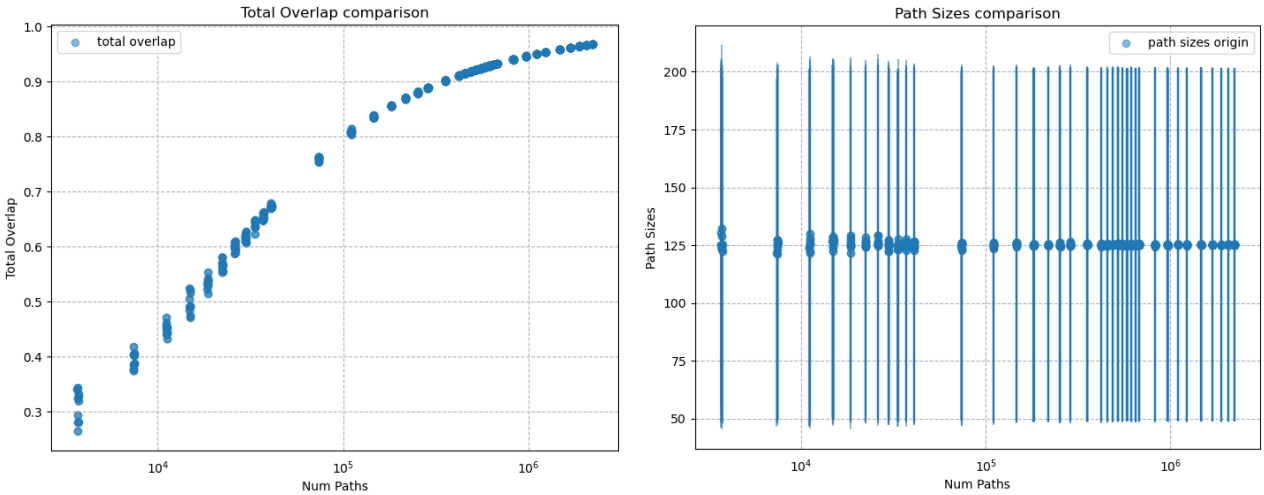


Figure 7.4: Characterization measurements plots for a parametric analysis centered at point 1 considering candidates in radius of $r = 2500.0$ m for several different ρ values.

The correlation between the number of paths and the number of paths on a fixed-radius circle is as expected. The more paths picked without repetition of a fixed region, the more likely are they to overlap. While we expect the Compressed Trie format benefits from an increasing overlap, the path list one is expected to be unsensitive to it. Therefore, this scenario is suitable to test the asymptotic behavior of Compressed Trie format against the Path List format considering the overlap.

We also find path sizes are long but variate regardless of the number of paths. This makes this measurement unsuitable for characterizing the different Path Datasets. But motivates the improvement of our approach, for instance by searching for more efficient alternatives for *Unary Coding* to code path sizes.

Having understood the feature of the battery of Path Datasets generated, we are ready to understand the associated performance measurements of all stages of the Problem Description.

Before this, however, a comment is needed. In all plot, we represent all formats even though the Path Lists ones (*PL UNSORTED FEO*, *PL UNSORTED AEO*, *PL SORTED AEO*) are expected to have exactly the same performance for all stages not involving serialization, as their in-memory representation is the same.

1. **Collection**

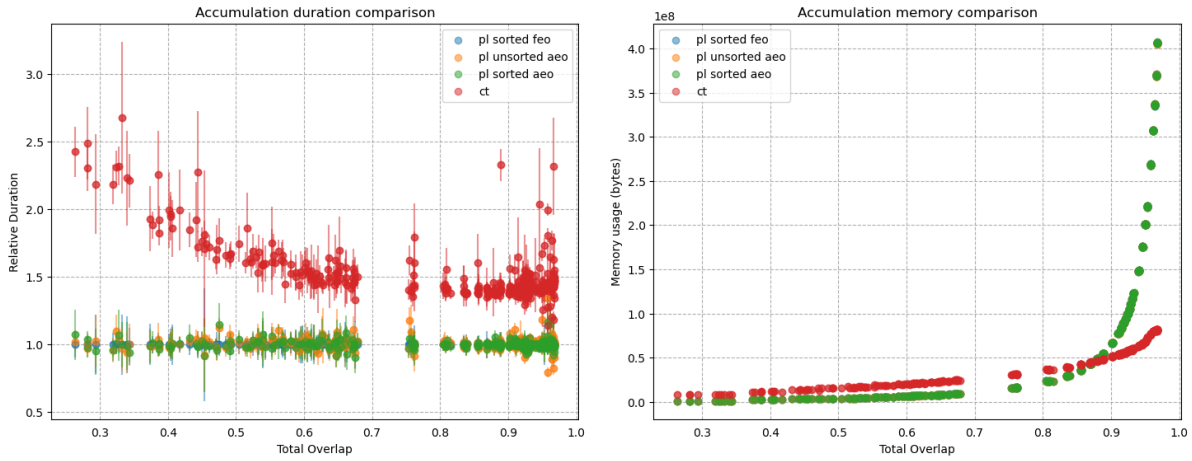


Figure 7.5: Performance plots for Accumulation. The left plot shows the relative duration. Whose values range from 2000 ms to 12000 ms for the base case. The right plot shows the memory consumption in bytes for different overlaps.

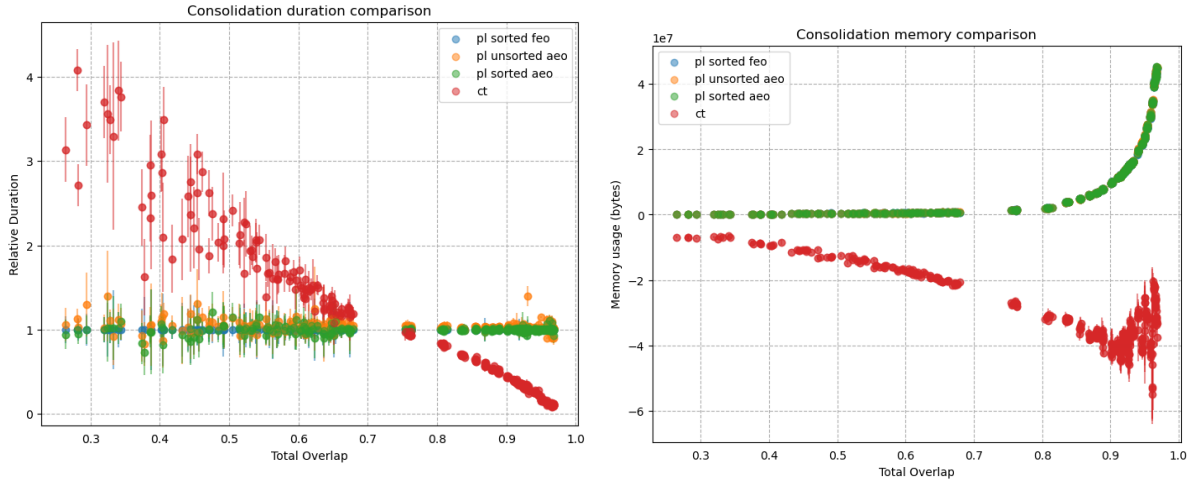


Figure 7.6: Performance plots for Consolidation. The left plot shows the relative duration. Whose values are always below 2000 ms. The right plot shows the memory consumption in bytes for different overlaps.

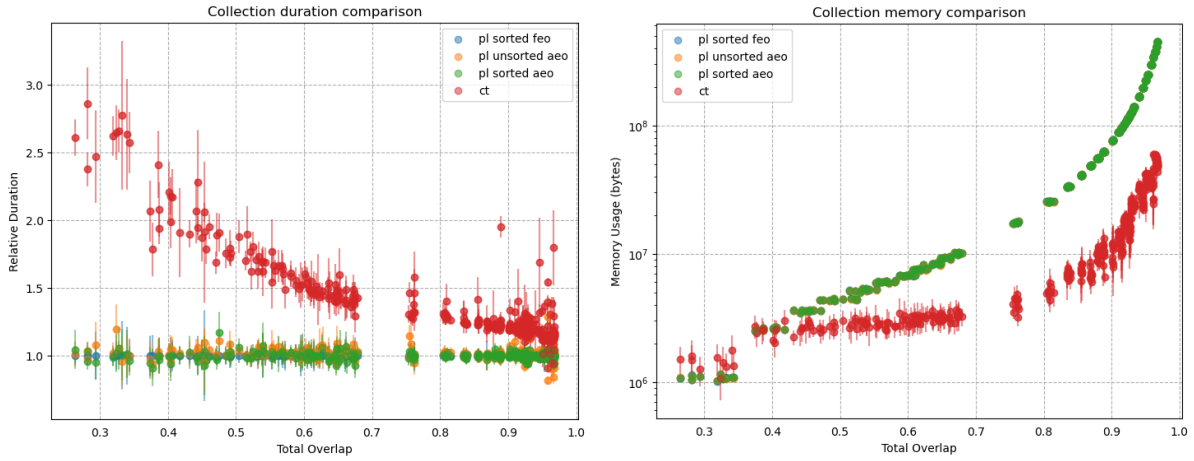


Figure 7.7: Performance plots for Collection (Accumulation + Consolidation). The left plot shows the relative duration. Whose values are below the 12000 ms.

We see that, overall, *Collection* consumes more time than the Path List format. In exchange, the total memory consumption is lower, for $\theta > 0.4$, which could be expected because it is the memory occupies by the Dense Trie against the memory occupies to have a list of paths, which may be repeated.

In general, both time and memory consumption decrease (in relative terms) with the overlap when compared to the Path Dataset, as expected. However, this step cannot be fully understood without analyzing its two sub-steps: accumulation and consolidation.

The Accumulation is slower for the *CT* format regardless of the overlap, even though it reduces for higher overlaps. While Consolidation also speeds up with overlaps, with respect to path lists and at a certain point it overpasses it. However, the Accumulation step is dominant, making the Collection step slower, overall. Regarding memory consumption, the accumulation uses more memory than path lists. After consolidation, it is clear that memory is further reduced in the Compressed Trie case for all cases, even though it is better the higher the overlap.

As a summary, we see that *CT* is slower for collection but asymptotically more efficient for representing Path Datasets in memory. However, we also need to consider the fact that during the steps of Collection, the memory consumption is also higher for a wide range of overlaps.

2. **Store & Restore**

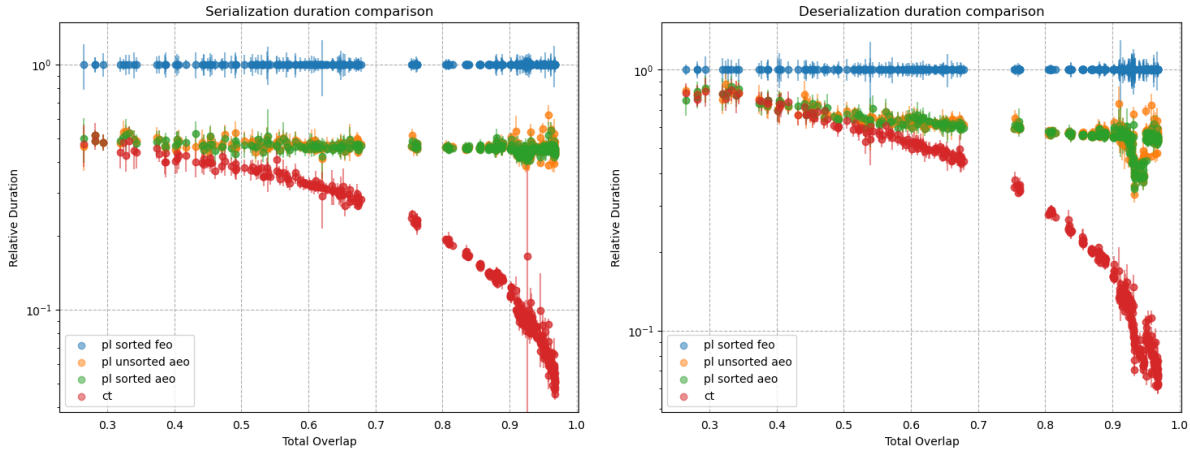


Figure 7.8: Duration comparison plots for the Serialization and the Deserialization steps. In both cases the *PL SORTED AEO* and *PL UNSORTED AEO* overlap, because they are in the same range of durations.

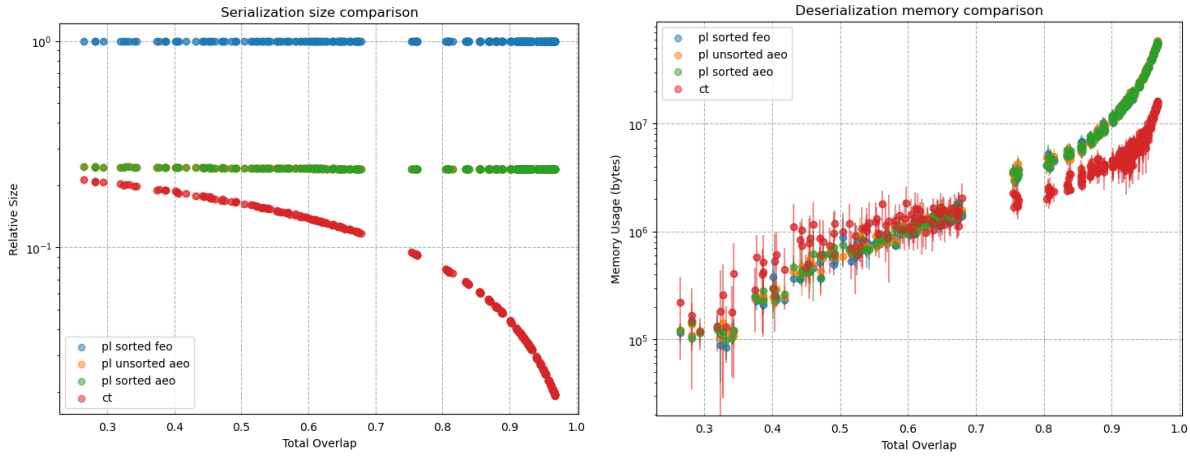


Figure 7.9: Size (left) and memory (right) comparison plots for the the Deserialization step. In the size case, the *PL SORTED AEO* and *PL UNSORTED AEO* overlap, because they are in the same range of durations. The right case, as previously mentioned, all *PL* formats overlap.

Serialization and Deserialization results are consistent among the methods in both duration and size. It is clear, that the *PL SORTED FEO* case is the less efficient among them. Over it, the *PL UNSORTED AEO* and *PL SORTED AEO* provide an improvement independent of the overlap, and, therefore, on the number of paths. The gain in size is between $3.\widehat{3}$ and 5. This is partially due to the *Adaptive Edge Offset Coding*. While *PL SORTED FEO* uses 8 bits for each edge offset, encoding most vertices, which are of 2-degree nodes, use just 1 bit. However, the size of paths are coded using *Unary Coding* so, for each edge offset,

there is a bit indicating whether it is the last one or not. This makes 9 bits per edge offset against 2 bits, approximately a gain of 4.5, matching what has been found. This reduction in size is consistent with the reduction in times of both Serialization and Deserialization. This is probably because the main component of this time are not the operations involved, but the time to write/read the data to/from disk, which is orders of magnitude slower than doing the same in memory. Regarding memory usage of the Deserialization stage, we also see a consistent improvement for higher overlaps for the *CT* formats. Asymptotically, the *CT* is, as expected better. Finally, the Compressed Trie format out bests all of them in time and size, as expected. It ranges from the limit with $\theta \to 0$, in which this format is approximately, a Path List to $\theta \to 1$, with no gain, in which the representation of paths reaches up to a gain of 50. Two aspects are the most important ones. First, *CT* is always more efficient in time and size for serialization than *PL SORTED AEO*. Second, if when increasing the number of paths, the overlap also increases, the size of the *CT* scales at a mitigated rate, probably logarithmic, compared to any Path List format.

3. DEFLATE & INFLATE

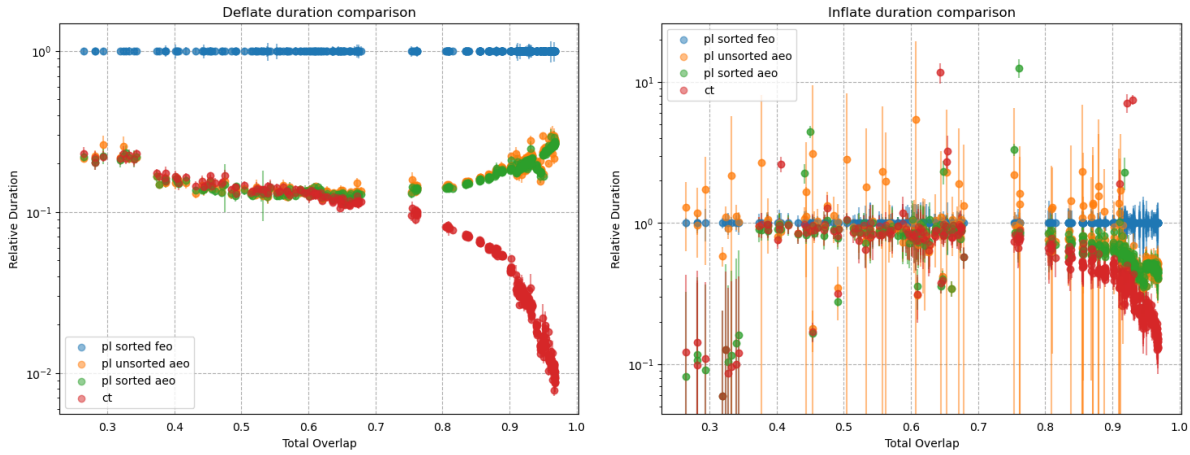


Figure 7.10: Deflate (left) and Inflate (right) comparison against the total overlap, i.e. number of paths in the Path Dataset. In both cases the *PL SORTED AEO* and *PL UNSORTED AEO* overlap, because they are in the same range of durations. However, in the Inflate case the variability in the measurements is significant.

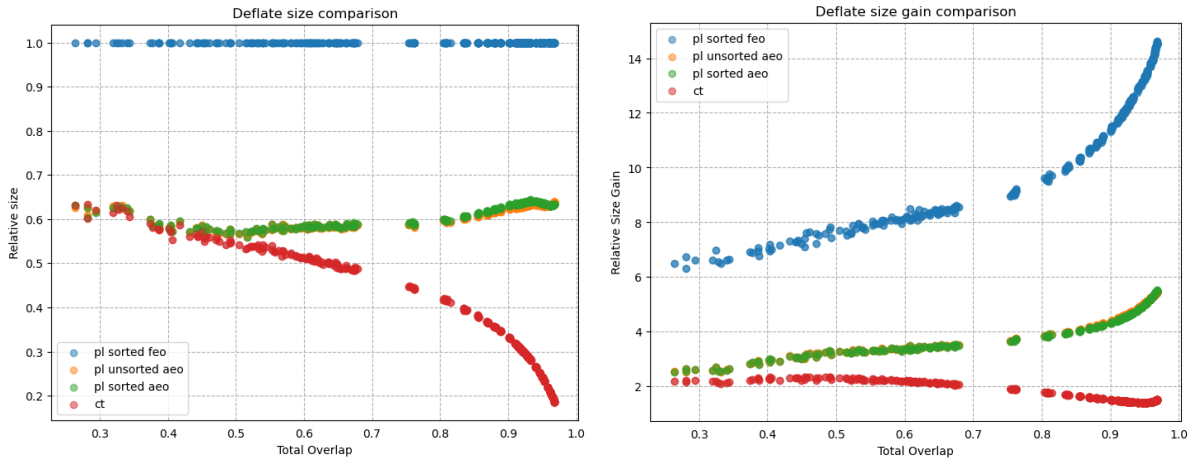


Figure 7.11: Deflate size (left) and size gain with respect to serialization size (right) comparison against the total overlap, i.e. number of paths in the Path Dataset. In both cases the *PL SORTED AEO* and *PL UNSORTED AEO* overlap, because they are in the same range of sizes and gains.

The general trends identified are similar to the Serialization & Deserialization steps. We see the *Adaptive Edge Offset* compression leads to a significant reduction in size and that *CT* format overpasses the rest, specially for higher values of the overlap.

Understanding data compression as an optimization problem, the *Adaptive Edge Offset* and *CT* give a more clever step towards the minimum than *DEFLATE*. In the first case, because the *FEO* representation has many unused bits. For instance, representing an edge offset of a vertex with out-degree 3 requires only 2 bits, but 8 are used, 6 of them as 0. Even *DEFLATE* can exploit this type of redundancies, it cannot do it as effectively as we can do, because it has less context than we do. It occurs similarly with *CT*. By storing shared parts deduplicated, we are reducing the amount of redundant data *DEFLATE* needs to compress and identifying more easily frequent patterns, such as frequently use sub-paths.

This also explains why the use of *DEFLATE* is less effective in the *Adaptive Edge Offset* cases. Therefore, even though, it always reduces the size of the Path Dataset in disk, using those formats may make *DEFLATE* redundant for Path Datasets. Actually, for *CT* we see the gain is up to a factor 2.

Finally, a last result is the fact that storing path lists lexicographically sorted does not produce any neat gain in size. However, we observe is a clear increase in the variability of the *INFLATE* (decode) duration. This could be explained because DEFLATE needs to do more work (keeping a larger dictionary) to achieve the same compression level. But, to the best of our knowledge, this should be reflected in both *DEFLATE* and *INFLATE*

steps. In conclusion, there is no suitable explanation for this behavior.

4. **Process**

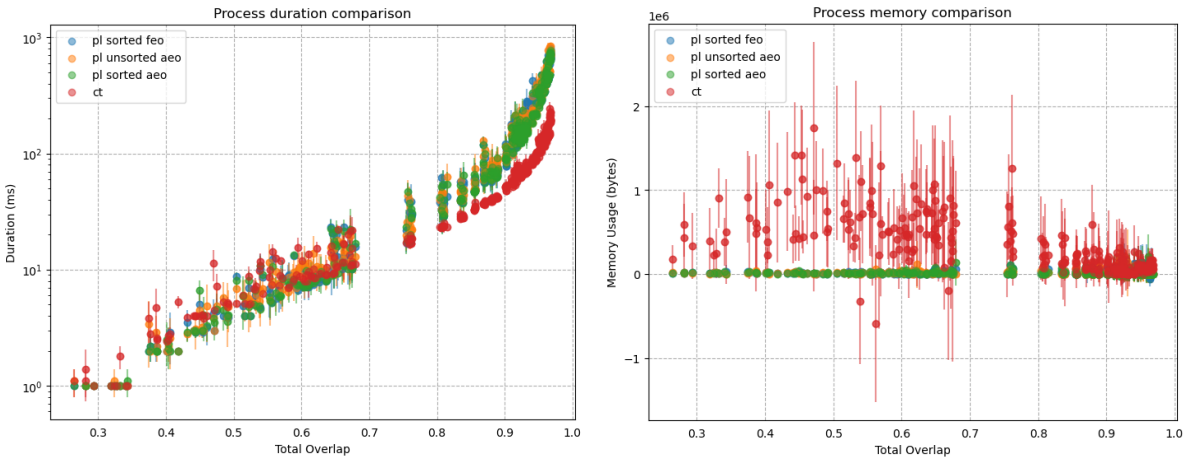


Figure 7.12: Performance plots for Process. The left plot represents the evolution of the duration of the Process with the overlap, for different Path Dataset formats. The right plot represents the same evoluation but for memory instead. In this second case, the plot shows a high variability in the results for low overlaps (and number of paths).

The duration of the *Process* step clearly shows the *CT* format is the fastest $\theta \to 1$. This would, a priory, be expected for any value of θ given the discussion in Process. Regarding memory, *CT* consumes more memory for any overlap.

The explanation for both behaviors is that *CT* uses a stack to run the iteratePaths, unlike Path Lists, which does not require of auxiliary data structures such. The stack implementation and hyperedge label cache lead to some time overhead and certainly to memory overhead. While the stack is needed to run the algorithm, the hyperedge label sequence cache is used to keep the extracted edge offset (labels) from the CRS data structure inside the Compressed Trie to an independent data structure (std::vector). This makes the implementation easier, but maybe a more efficient version could be provided.

Fortunately, this extra memory consumption is around of below 10^6 but which is negligible if we compare the *Deserialization* consumption in *CT* and *PL* formats, which is around 10^7 bytes (see 7.9).

7.2 Case Study: Valles Region

The Vallès region is contained in the BCN Metropolitan Area. Its population is dense but concentrated in separated cities connected among them and with Barcelona by highways (AP-7, C-33, C-17 and A-2). This results in a region with an intense interurban traffic, which causes recurrent congestion. A scenario in which efficient management of Path Datasets is desirable.

The purpose of this case study is assessing the orders of magnitude of the performance measurements for the formats discussed in this thesis in a heterogeneous region. We expect being able to provide an estimation of the efficiency of our approach in practice.

7.2.1 Graph Description

Figure 7.13: Bounding box of the Vallès region road network under study.

The graph it comprises (after simplification) 21,194 vertices, with 5,361 vertices as origins and 5,356 vertices as destinations, and 42,023 edges. The degree distributions before simplification are the following ones:

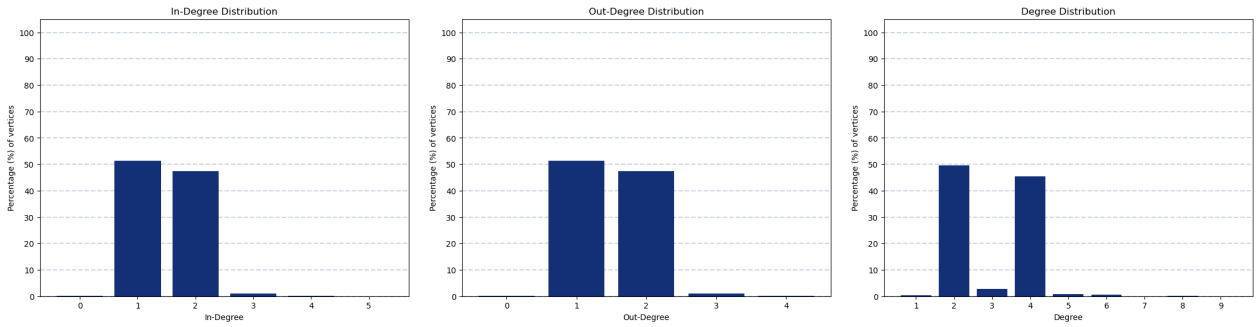


Figure 7.14: Histogram of vertices degree of the Valles road network graph prior to contraction of redundant edges.

After contracting redundant nodes, the distributions become:

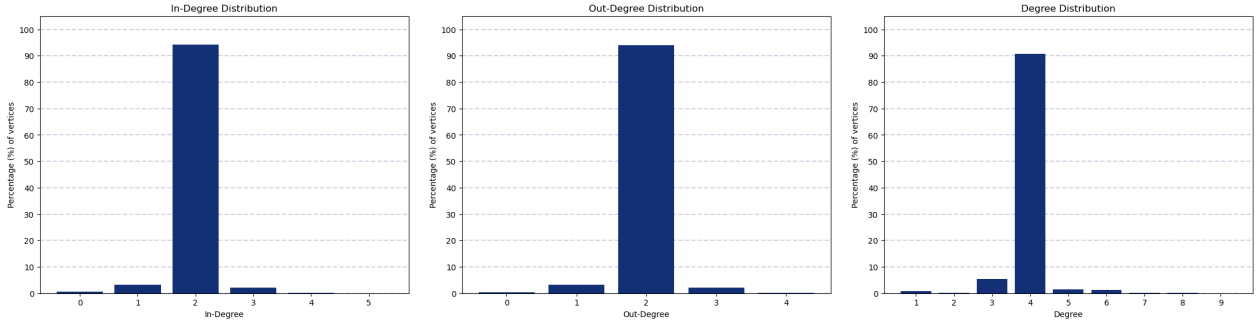


Figure 7.15: Histogram of vertices degree of the Valles road network graph after to contraction of redundant edges.

The contraction step provides the expected results, the same results as in Case Study: Barcelona Center Area.

7.2.2 Scale Analysis

Analogously to Parametric Analysis, out of the numerous parameters, we fix the ones to some fixed values that produce representative results in our experience. In this way, we prevent an excessive computational burden when carrying out the test.

We chose the Number of Path Dataset samples, number of Batches and the Deviation to be fixed to those values because, manually testing our application, we found it induces some variability in the paths found without incurring in an excessive computational burden. We set the λ to 5 km in order to keep only interurban trips in the generatic Path Datasets.This leaves a scenario in which there are at most $5,361 \times 5,356 = 2.8 \cdot 10^7$ possible origin and destination paths. In practice, this number of reduced to approx. $2.3 \cdot 10^7$ due to the limit in λ.

Parameter	Pre-defined value
Number of Path Dataset samples	10
Number of Batches	10
Deviation	5 %
Minimum Distance (λ)	5 km
Radius (r)	∞

Table 7.3: *Parameters set to fixed pre-defined values for Scale Analysis.*

In order to assess the usability of this approx, we will consider the performance measurements of our approach for each stage for different but large number of o-d queries: 100k, 200k, 300k, 400k, 500k, **approximately below the 1% of the total possible o-d pairs**. Which is the case in which managing Path Datasets is problematic.

The obtained Path Datasets have the following characteristics:

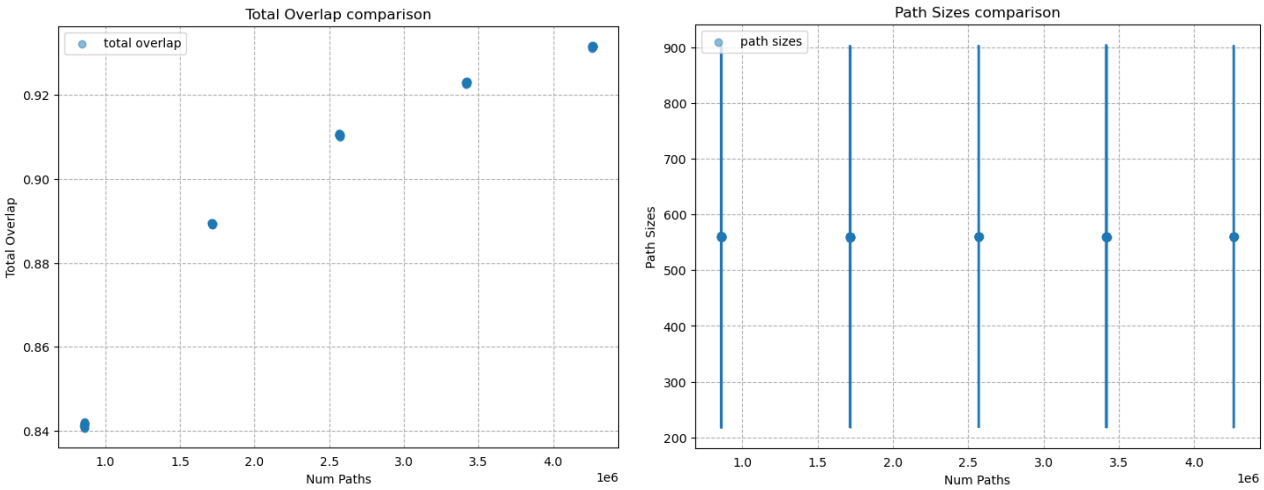


Figure 7.16: Characterization measurements plots for a parametric analysis in the whole Valles region but for different numbers of paths generated.

As in the previous case, it is expected that increasing the number of paths in a fixed region, increases the *Total Overlap*. Regarding the *Path Sizes*, the paths are longer and have more variability than in 7.4. This is because the origin-destinations can be up to approx. 20 km apart, rather than 3 km. This motivates, even more, the search for alternatives to *Unary Coding* to code the length of paths and hyperedge sequences.

After discussing the characterization, we can discuss the performance. In this case, since our objective is evaluating the scale of the performance measurements also, we will show the characteristics compared to the number of paths in the Path Dataset.

But before starting this analysis, we comment that all plots, we represent all formats even though the Path Lists ones (*PL UNSORTED FEO*, *PL UNSORTED AEO*, *PL SORTED AEO*) are expected to have exactly the same performance for all stages not involving serialization, as in-memory representation is the same.

1. **Collection**

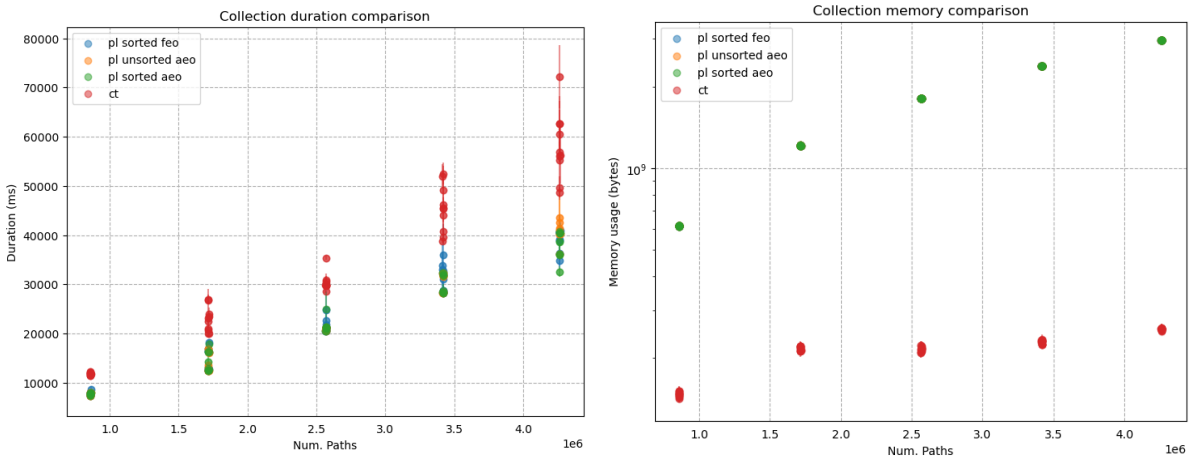


Figure 7.17: Performance plots for Collection (Accumulation + Consolidation). The left plot shows the duration in miliseconds for different number of paths. While the left one shows the memory consumption.

Regarding duration, we see that collection increases in an approximately linear fashion but at a faster pace for *CT*. While the *PL* formats go from 10s to 40s approximately, the *CT* ranges from 10s to 70s, almost twice.

In exchange, the memory consumption is much lower and increases at a slower rate with the number of paths. We see in the case of *CT* the memory consumption is of the order of magnitude of 200 MB, while for the *PL* formats it is a GB (1000 MB). In all cases, it's almost a factor 5.

2. **Store & Restore**

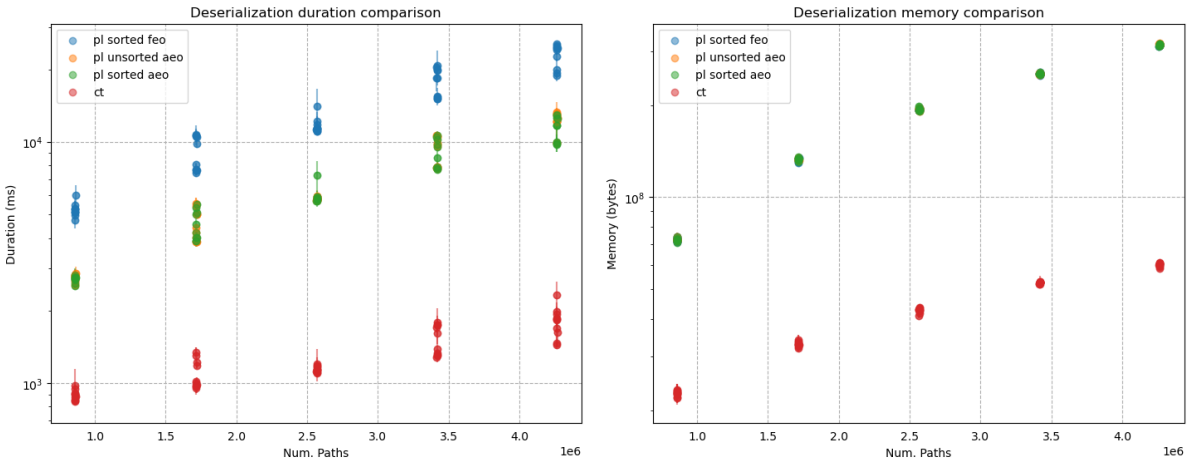


Figure 7.18: Deserialization performance measurements comparison to the number of paths. In the left side, the duration and in the memory usage of this process for the deserialized Path Dataset.

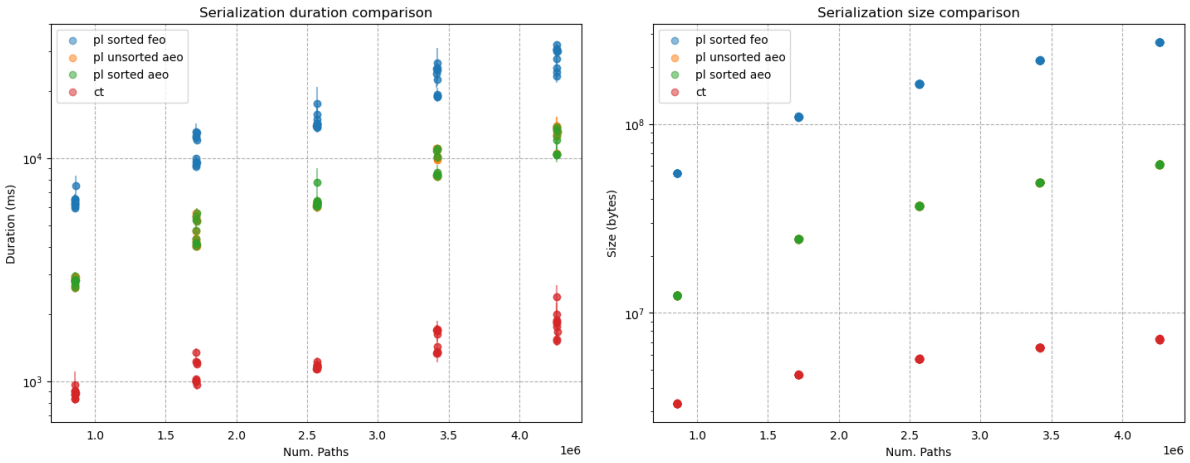


Figure 7.19: Serialization performance measurements comparison to the number of paths. In the left side, the duration and in the right side the disk size of the serialized Path Dataset.

The results show without a doubt the *CT* format is superior to the other cases. The problem size increases with the number of paths, so does the duration, memory usage and size for all cases, but *CT* ones are smaller and grow at a slower pace.

In more detail, the Serialization of *CT* is faster. While *PL SORTED FEO* takes around 6s to 20s, and *PL SORTED AEO* and *PL UNSORTED AEO* take from 3s to 10s, the *CT* format takes from 0.9s to 2s : approx. an order of magnitude of difference. The same range of values is shown for the *Deserialization* step, so the same conclusion is obtained:

CT is superior.

The memory consumption for *PL* formats requires from 70 MB to 300 MB, approximately. However, *CT* requires from 20 MB to 60 MB. Increasing the number of paths, only makes this gain greater. Notice, however, the Collection stage for CT is around the 200 MB. Even though collection is expected to consume more, this proves that there is a safe margin of, between 3.333 and 10 to reduce the collection memory consumption.

Regarding the size in disk, the *PL SORTED FEO* requires from 60 to 300 MB, and *PL SORTED AEO* and *PL UNSORTED AEO* require 10 MB to 60 MB, an order of magnitude. In the case of *CT* even the largest size is lower than the smaller in any other format. It requires from 2 MB to 7 MB to store the whole Path Dataset, another order of magnitude less.

3. **Process**

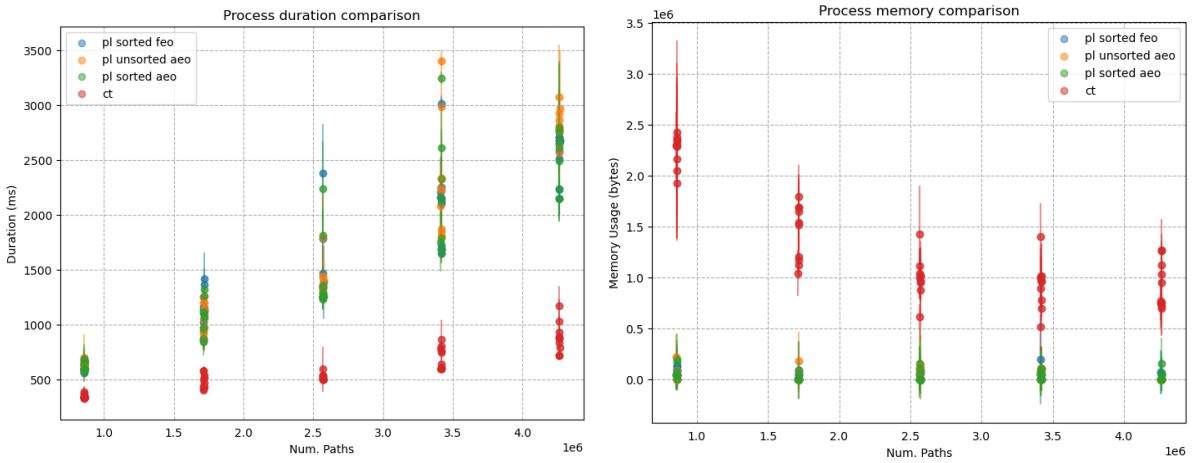


Figure 7.20: Performance plots for Process. The left plot represents the evolution of the duration of the Process with the number of paths, for different Path Dataset formats. The right plot represents the same evoluation but for memory instead.

For the *Process* stage we see the *CT* format takes from below 0.5s to 1.0s approx, to process all paths, while the *PL* formats take from 0.75s to approx. 3.0s. Of course, the duration of any format increases with the number of paths, but the *CT* format does so at a much slower pace.

Memory consumption is higher for *CT*, because of the auxiliary data structures. In particular, the explanation could be the use of the auxiliary std::vector as cache for hyperedge edge offsets. With a higher overlap, it is likely that this cache is smaller, and its reserved

memory is smaller also. However, tests should be carried out in order to evaluate if it is actually the case. But, whichever the case, only a 1-2.5MB are required, while the difference in size of the *Deserialization* step is at least of 70 MB in *PL* to 10 MB in *CT*. Therefore, the overall memory consumption is favorable for *CT*.

Chapter 8

Conclusions & Future Work

In this thesis, we have developed a new and generic approach to represent Path Datasets describing path data associated to Road Networks graphs. Our approach is designed to be efficient for a use pattern with the following consecutive stages: Collection, Storage, Retrieval, and Process. Which is a particular case of the single-collection, multiple processing approach. The proposed representation is based on using Compressed Tries to index the data associated to paths with a common origin vertex. While the data described is the usage count (number of occurrences of the path), it can be customized straightforwardly.

We have provided an exhaustive theoretical description and complexity analysis of this solution in for all the Problem Description stages. Among them, we emphasize the recurrent use of tree traversal algorithms, *CRS* format, *DFUDS* and *Adaptive Edge Offset Coding* as building blocks of this approach.

An experimental design has been developed in order to assess rigorously the strengths and weaknesses of our approach in for realistic cases. This consisted of three main tasks. First, implementing the different representations of Path Datasets and a common way to validate them. Second, providing a meaningful characterization measurement θ (the Overlap) and performance measurements to be assessed them qualitatively and quantitatively. And, third, providing a method to Synthetically Generate Path Datasets which are realistic and have certain characterization among the myriad of possible Path Datasets for the same Road Network.

To set our proposal in practice, have chosen two Road Networks. The first one being the Case Study: Barcelona Center Area a purely urban area. In this first road network, we analyzed the impact of the *Overlap* in the performance of all stages. There are three main outcomes of

this analysis. First, the *Total Overlap* measures the efficiency of the *Compressed Trie* format: the higher, the more efficient. And, second, our approach clearly outperforms the rest for *Serialization & Deserialization* , *DEFLATE & INFLATE* and *Process Time*. However, our is slower than the *PL* format during the *Dense Trie Collection* step. And last, the *Path Sizes* found motivate the research on compression approaches other than *Unary Coding* to encode Path Sizes, because they tend to be variated but also large, which is unfit for this type of compression.

In the Case Study: Valles Region case, we studied the scales and behavior of the different formats considered in this dissertation to understand their efficiency and the scale of the performance measurements in an interurban and larger network. The results show that in this scenario, the *CT* format outperforms, in some cases by orders of magnitude, the *PL* formats for all stages and measurements. And it shows a much slower scaling in them too.

With these two analyses, we have able to identify the strengths and weaknesses of the Compressed Trie Indexation approach. Based on them, we propose the following topics as future work to amplify those strengths and mitigate the weaknesses:

- **Optimization of Compressed Trie Collection**

  Our collection approach has shown an excessive time (and memory, even if did not show to be problematic) consumption because of the pre-allocation of space for the possible child nodes of each Trie node. Additional work is needed to minimize it. A strategy which directly generated a *Compressed Trie* without a *Dense Trie Collection* may overcome this issue, such as the one proposed CRS-Split Collection.

- **Trie Compaction**

  As well as paths may overlap on their prefixes (origin), they may also do so on their suffixes (destination). In both cases, this overlap occurs because the closer to the endpoint, the fewer possible paths to reach/leave it. The problem is essentially symmetric. From our approach, one way to exploit both overlaps is joining origin-Trie branches which diverged but have common suffixes (sub-paths to destination). See Trie Compaction for a more detailed discussion on this proposal.

- **Optimize Edge Offset Coding**

  Adaptive Edge Offset Coding has proved to be a very efficient and robust compression method, adequate for compressing edge offsets for Road Networks. Its main strength is

adapting the alphabet of values for each edge offset using *Binary Coding*. This can be exploited further by using methods such as Huffman Coding which is always at least as efficient as *Binary Coding*. In particular, using a Huffman dictionary for each origin-Trie and outgoing-degree.

- **Sophisticated compression methods for length of sequences**

  In section Storage, we proposed using *Unary Coding* to code and did not attempt to find a more sophisticated approach for simplicity. However, results show that Compressed Trie hyperedges are usually variated but large. *Unary Coding* is less efficient in that situation. Alternative approaches exist between *Unary Coding* and *Fixed Length Coding*, such as Variable Length Quantity coding (VLQ) which are likely more efficient than any of them.

- **Clustered Compression**

  As discussed, *DEFLATE* is very effective for our use case. However, this algorithm relies on the similarity of the processed data in order to efficiently in space and time compress the data. The more similar, the better compression it may achieve. Thus, a potential way to ease compression is splitting the compression in clusters or groups of tries which are similar, for instance tries whose origins are close.

Appendix A

Bit Stream Implementation

In this appendix, we provide the details of the Bit Stream used in this thesis, along with the reasons why it is needed.

Most modern computer architectures perform operations in units of (fixed) *word length*; usually an 8-bits block (byte). Therefore, even if less than those bits are needed to represent a variable, the computer uses a whole block. The number of unused bits will range from 1 to *word length*-1 and can introduce severe inefficiencies when coding variables.

To overcome this limitation, we use a Bit Stream, a data stream which effectively writes and reads individual bits. As this is impossible due to computer architecture, it is done by accumulating individual Bit-wise operations on a block of a size multiple of *word length* and then doing a memory transaction of the whole block.

We achieve this behavior, by combining bit manipulations at the beginning of the block and left shift operations. This, together with a counter indicating the position at which the first bit of the block was written, as shown in the following figure:

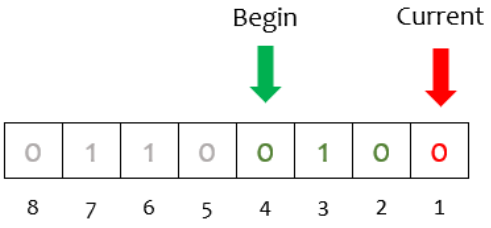


Figure A.1: Representation of Bit Stream block. Each of the cells represents the bits in the byte block, enumerated from left to right. The black arrow indicates the not written or read position.

Each time the first bit is manipulated, a left shift occurs, effectively moving the start of the bits in one place. When it arrives to the end of block: the whole block is written or read without the need of padding. Padding will only occur once the Binary Stream is closed, because the last block may not be written in all bits. For that reason, it is recommended that data is binary encoded in a single long binary stream rather than in multiple smaller ones. In our implementation, those blocks are *char* (C++ Fundamental Types), because it is the smallest variables which can be written without unused bits. To complete the explanation, we describe the operations to write and read the first bit:

- **Write**. In order to write, we need to set and unset operations to encode the binary stream. As in the main loop, we overwrite the state of the bit in the first position, and then they are all shifted. Therefore, bits are needed to be *set* or *unset* only in the first position. Moreover, as we write by blocks, a single *unset* is done once to all bits and after this only using the *set* is used. The unset operations are done by setting the *char* block to *char{0}* (which is 00000000). The *set* operation, is implemented as an OR operation of the mask char{1} (00000001) on the block. This operation is exemplified in the next figure:

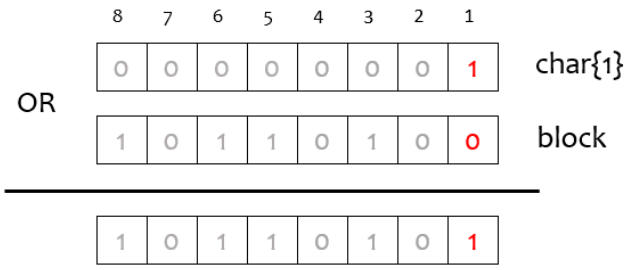


Figure A.2: Representation of Bit Stream write the bit at first position operation. The OR gate applied with char1 leaves all bits unchanged except the first one which, if 0 is set to 1, otherwise it's set to 1.

- **Read**. Given the design of the bit stream, we only need to read the first bit. For this, we can just use an AND gate of the mask char{1} (00000001) on the block. If the result is char{1} (00000001), the bit is 1 if it is *char{0}* (which is 00000000) it is a 0. This operation is exemplified in the next figure:

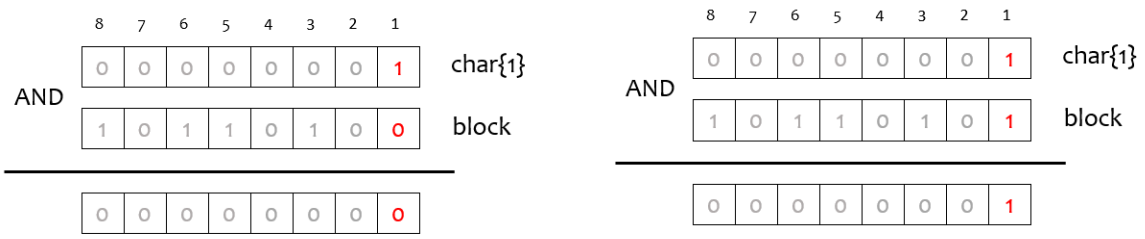


Figure A.3: Representation of Bit Stream read at the first position. The AND fate with *char*1 effectively sets all bits to 0 except the first one which is left unchanged.

A final comment on our implementation is the fact that hard disks usually write data in blocks of 4 kB. That means that, as with word lengths, if the amount of data is below it, it will still commit 4 kB. While this could seem an issue, it is not, as we expect that the total amount of data to be written is much larger than this unit.

Appendix B

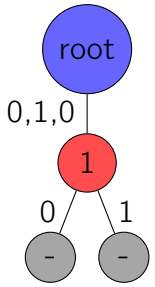
CRS-Split Collection

In this appendix, we give an overview of the idea for solving the excessive memory consumption in Collection of Dense Trie mentioned in Conclusions & Future Work. The idea behind it idea is trying to build the Trie as compressed as possible and, in this way, avoid generating intermediate nodes which would end up condensed inside Compressed Trie Hyperedges.

In order to achieve it, the logic to follow is splitting edge offset sequences as a sequence of existing hyperedges plus a final hyperedge where the current hyperedge and the added sequence differ. To include this last piece, the existing hyperedge will need to be split into two (adding a new node densely in the process) at the point where both sequences diverge. The following example illustrates this procedure:

Initial state

Adding $\{0,1,0\}$



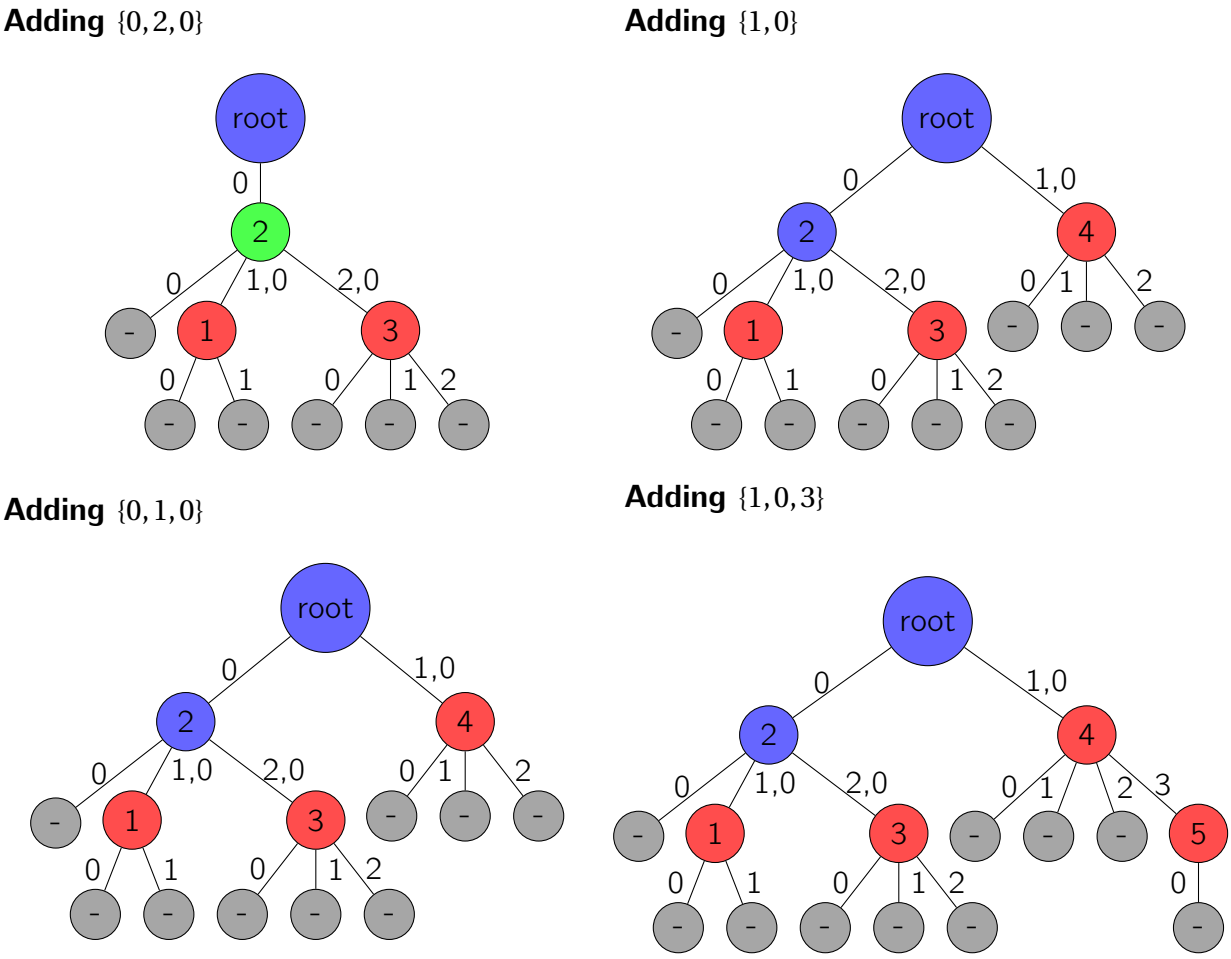


Figure B.1: Exemplification of path-collection process on a Trie based on input sequences of Figure 5.1.

In this example, only two nodes are saved with respect to the twelve in Figure 5.3. We did not provide a more adequate in order to make it familiar to the reader. Nevertheless, we see all hyperedges are already created. In a more realistic case, hyperedges tend to be much longer than in this one. Therefore, the reduction in memory consumption is expected to be significant.

Appendix C

Trie Compaction

In our opinion, a straightforward step to improve the Compressed Trie Indexation approach is exploiting also the overlap of paths close to the destination. This overlap occurs because, as with origins, close to the destination there are fewer lest cost sub-paths to reach it.

An approach to capture this is *compacting the Trie* by attempting to join branches from the bottom of the tree to the root. The following figure exemplifies this process.

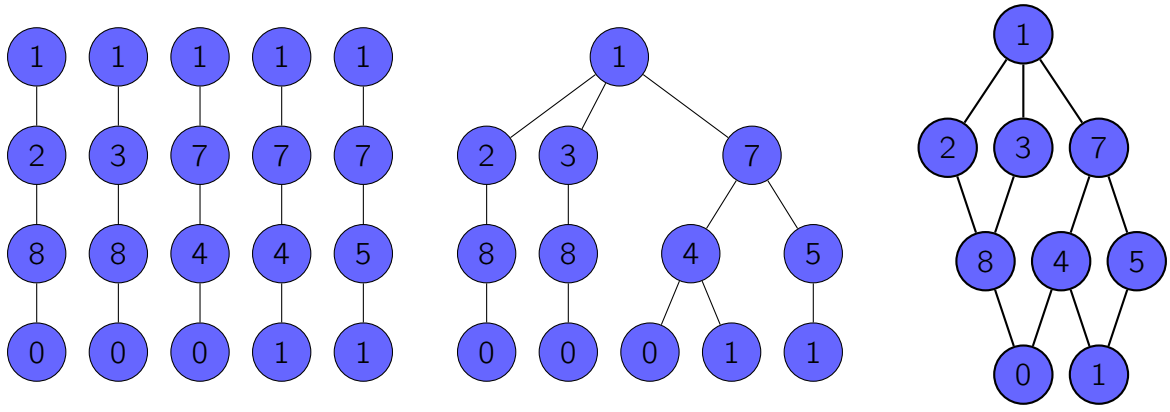


Figure C.1: Example of compaction approach starting from a disjoint representation of path sets, to a Trie one and finally a DAG obtained from compacting the Trie.

This type of compaction can be done efficiently with the algorithms described in [5] and [7] by just identifying as equivalence classes those of the tries which have associated the same vertices in the graph and have the same sub-tries as descendants. However, one of the problems with this representation we lose the ability to map a sequence to a node in the Trie, because there may be more than one path from the root to a node. This would not be a problem if our

intention were just indexing a set of paths collectively. But we also want to store the data associated to those paths. A possibility to overcome this, is indexing the data in the same order as the DFS would traverse the compacted sub-Trie. Another problem, is the need of finding a new encoding/decoding algorithms, because the DFUDS only applies to trees. A possible solution is decomposing the DAG into a combination of tries and coding, each using DFUDS plus additional references among them. For instance, the previous DAG could be decomposed as:

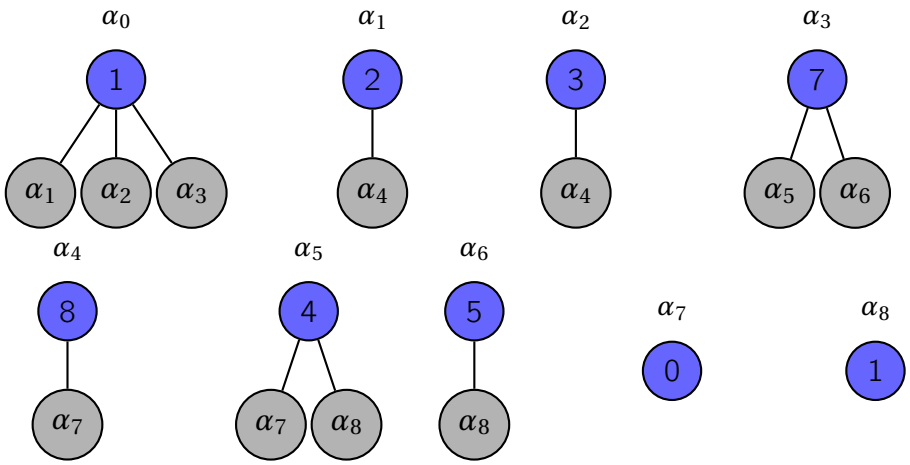


Figure C.2: Example of DAG decomposition in tries of DAG in Figure C.1.

However, a deeper revision is needed of this idea. Specially, regarding referencing among tries.

As a final comment, this is a similar to what is proposed in [18] but in a completely different use. There, the authors proposed using a DAG in order to compactly code the index of a database table over a set of categorical features to access a given data (node) in such a way that it supports multiple ways to access it efficiently. But in this case, nodes are already identifiable, the index just encodes all the ways to access them. Instead, we want to index the nodes themselves uniquely.

Bibliography

[1] Marc Barthélemy. Spatial networks. *Physics Reports*, 499(1-3):1–101, feb 2011.

[2] Gernot Veit Batz, Robert Geisberger, Dennis Luxen, Peter Sanders, and Roman Zubkov. Efficient route compression for hybrid route planning. In Guy Even and Dror Rawitz, editors, *Design and Analysis of Algorithms*, pages 93–107, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[3] David Benoit, Erik D. Demaine, J. Ian Munro, and Venkatesh Raman. Representing trees of higher degree. In Frank Dehne, Jörg-Rüdiger Sack, Arvind Gupta, and Roberto Tamassia, editors, *Algorithms and Data Structures*, pages 169–180, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[4] Chao Chen, Daqing Zhang, Yasha Wang, and Hongyu Huang. *Trajectory Data Compression*, pages 25–46. Springer Singapore, Singapore, 2021.

[5] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, oct 1980.

[6] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), nov 2009.

[7] Philippe Flajolet, Paolo Sipala, and Jean-Marc Steyaert. Analytic variations on the common subexpression problem. In Michael S. Paterson, editor, *Automata, Languages and Programming*, pages 220–234, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[8] Vincent Gripon, Michael G. Rabbat, Vitaly Skachek, and Warren J. Gross. Compressing multisets using tries. *2012 IEEE Information Theory Workshop*, pages 642–646, 2012.

[9] J. L. Gross and J. Yellen. *Graph theory and its applications*. Chapman and Hall/CRC, U.S.A., 5 edition, 1982.

[10] Yunheng Han, Weiwei Sun, and Baihua Zheng. Compress: A comprehensive framework of trajectory compression in road networks. *ACM Trans. Database Syst.*, 42(2), may 2017.

[11] Harry H. Ku. Notes on the use of propagation of error formulas. 2010.

[12] Colt McAnlis and Aleks Haecky. *Understanding Compression: Data Compression for Modern Developers*. O'Reilly Media, Inc., 1st edition, 2016.

[13] Jonathan Muckell, Paul Olsen Jr, Jeong Hwang, Catherine Lawson, and S. Ravi. Compression of trajectory data: A comprehensive evaluation and new approach. *GeoInformatica*, 18, 07 2014.

[14] IBM Raymond S . Glover. Path compression of a network graph, US Patent 9,747,513 B2, Aug. 29, 2017.

[15] Yuriy A. Reznik. Coding of sets of words. In *2011 Data Compression Conference*, pages 43–52, 2011.

[16] Khalid Sayood. *Introduction to Data Compression, Fourth Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2012.

[17] Renchu Song, Weiwei Sun, Baihua Zheng, and Yu Zheng. Press: A novel framework of trajectory compression in road networks. *Proc. VLDB Endow.*, 7(9):661–672, may 2014.

[18] Adobe Systems Incorporated Walter Chang Nadia Ghamrawi, Arun Swami. System and method of efficiently representing and searching directed acyclic graph structures in databases, US Patent 7,580,918 B2, Aug. 25, 2009.