Semantic Web 0 (0) 1
IOS Press

Generating Public Transport Data for the Web based on Population Distributions

Ruben Taelman, Pieter Colpaert, Ruben Verborgh and Erik Mannens
imec - Ghent University - IDLab, Sint-Pietersnieuwstraat 41, 9000 Ghent, Belgium
E-mail: ruben.taelman@ugent.be

Abstract.
Applying Linked Data technologies to geospatial and temporal data introduces many new challenges, such as Web-scale storage, management, and the transmission of potentially large amounts of data. Several benchmarks have been introduced to evaluate the efficiency of systems that aim to solve such problems. Unfortunately, the synthetic data many of these benchmarks work with have only limited realism, raising questions about the generalizability of benchmark results to real-world scenarios. On the other hand, real-world datasets cannot be configured as freely, and often cover only certain aspects. In order to benchmark geospatial and temporal RDF data management systems with sufficient external validity and depth, we designed PODIGG, a highly configurable generation algorithm for synthetic datasets with realistic geospatial and temporal characteristics comparable to those of their real-world variants. The algorithm is inspired by real-world public transit network design and scheduling methodologies. This article discusses the design and implementation of PODIGG and validates the properties of its generated datasets. Our findings show that the generator achieves a sufficient level of realism, based on the existing coherence metric and new metrics we introduce specifically for the public transport domain. Thereby, PODIGG provides a flexible foundation for benchmarking RDF data management systems with geospatial and temporal data.

Keywords: Public Transport, Dataset Generator, Benchmarking, RDF, Linked Data

1. Introduction

The Resource Description Framework (RDF) and Linked Data technologies enable distributed use and management of semantic data models. [9,4] Datasets with an interoperable domain model can be stored by different data owners, but different information architectures with varying trade-offs exist. Architectural choices may for example allow certain questions to be answered quickly over the Web of data, while other queries may return slow or even no results. Therefore, benchmarking different information architectures – with different sizes of datasets and varying characteristics – can show the weaknesses and strengths of such choices.

Regardless of whether existing data-driven benchmarks use real or synthetic datasets, the *external validity* of their results can be too limited, which makes a generalization to other datasets difficult. Real datasets, on the one hand, are often only scarcely available for testing, and only cover very specific scenarios, such that not all aspects of systems can be assessed. Synthetic datasets, on the other hand, are typically generated by *mimicking algorithms* [5,17,23,24], which are not always sufficiently realistic [14]. Features that are relevant for real-world datasets may not be tested. As such, conclusions drawn from existing benchmarks do not always apply to the envisioned real-world scenarios. One way to get the best of both worlds is to design mimicking algorithms that generate realistic synthetic datasets.

The area of geospatial [15] and temporal [1,18] benchmarking for RDF storage and querying systems provides no mimicking algorithms for generating realistic synthetic dataset with geospatial and temporal characteristics. There is a use for benchmarks that evaluate systems handling datasets with geospatial and temporal characteristics, but it is important that the datasets they use are realistic.

The *public transport* domain provides data with both geospatial and temporal properties, which makes this

an especially interesting source of data for benchmarking. Its representation as Linked Data is valuable because i) of the many shared entities, such as stops, routes and trips, across different existing datasets on the Web. ii) These entities can be distributed over different datasets and iii) benefit from interlinking for the improvement of discoverability.

In order to solve the need of realistic synthetic datasets with geospatial and temporal characteristics, we introduce a mimicking algorithm for generating realistic public transport data, which is the main contribution of this work. We observed a significant correlation between transport networks and the population distributions of their geographical areas, which is why population distributions are the driving factor within our algorithm. Our algorithm is subdivided into five sequential steps, inspired by existing methodologies from the domains of public transit planning [16] as a means to improve the realism of the algorithm's output data. These steps include the creation of a geospatial region, the placement of stops, edges and routes, and the scheduling of trips. Our second main contribution is an implementation of this algorithm, with different parameters to configure the algorithm. Finally, we confirm the realism of datasets that are generated by this algorithm using the existing generic structuredness metric [14] and new metrics that we introduce ourselves, which are specific to the public transport domain.

In the next section, we introduce the related work on dataset generation, transit network design, and transit feed formats. In Section 3, we introduce the main research question and hypothesis of this work. Next, our algorithm is presented in Section 4, followed by its implementation in Section 5. In Section 6, we present the evaluation of our implementation, followed by the a discussion and conclusion in Section 7 and Section 8.

2. Related Work

In this section, we present the related work on spatiotemporal and RDF dataset generation, existing public transit network planning methodologies and formats for exchanging transit feeds.

2.1. Spatiotemporal Dataset Generation

Spatiotemporal database systems store instances that are described using an identifier, a spatial location and a timestamp. In order to evaluate spatiotemporal indexing and querying techniques with datasets, automatic means exist to generate such datasets with predictable characteristics [21].

Brinkhoff [6] argues that moving objects tend to follow a predefined network. Using this and other statements, he introduces a spatiotemporal dataset generator. Such a network can be anything over which certain objects can move, ranging from railway networks to air traffic connections. The proposed parameter-based generator restricts the existence of the spatiotemporal objects to a predefined time period $[t_{\min}, t_{\max})$. It is assumed that each edge in the network has a maximum allowed speed and capacity over which objects can move at a certain speed based. The eventual speed of each object is defined by the maximum speed of its class, the maximum allowed speed of the edge, and the congestion of the edge based on its capacity. Furthermore, weather conditions or other external events that can impact the movement of the objects are represented as two-dimensional areas that exist for a period in time over the network, which apply a *decreasing factor* on the maximum speed of the objects in that area. The existence of each object that is generated starts at a certain timestamp, which is determined by a certain function, and *dies* when it arrives at its destination. The starting node of an object can be chosen based on three approaches:

dataspace-oriented approaches Selecting the nearest node based to a position picked from a two-dimensional distribution function.

region-based approaches Improvement of the dataspace oriented approach where the data space is represented as a collection of cells, each having a certain chance of being the place of a starting node.

network-based approaches Selection of a network node based on a one-dimensional distribution function.

Determining the destination node using one of these approaches leads to non-satisfying results. Instead, the destination is derived from the preferred length of a route. Each route is determined as the fastest path to a destination, weighed by the external events. Finally, the results are reported as either textual output, insertion into a database or a figure of the generated objects.

In order to improve the testability of Information Discovery Systems, a generic synthetic dataset generator [20] was developed that is able to generate synthetic data based on declarative semantic graph definitions. This graph is based on objects, attributes and relationships between them. The authors propose to

generate new instances based on a set of dependency rules (called "rules" [20]). They introduce three types of dependencies for the generation of instances:

independent Attribute values that are independent of other instances and attributes.

intra-record (horizontal) dependencies Attribute values depending on other values of the same instance.

inter-record (vertical) dependencies Relationships between different instances.

Their engine is able to accept such dependencies as part of a semantic graph definition, and iteratively create new instances to form a synthetic dataset.

2.2. RDF Dataset Generation

The need for benchmarking RDF data management systems is illustrated by the existence of the Linked Data Benchmark Council [2] and the HOBBIT[1] H2020 EU project for benchmarking of Big Linked Data. RDF benchmarks are typically based on certain datasets that are used as input to the tested systems. Many of these datasets are not always very closely related to real datasets [14], which may result in conclusions drawn from benchmarking results that do not translate to system behaviours in realistic settings.

Duan et al. [14] argue that the realism of an RDF dataset can be measured by comparing the *structuredness* of that dataset with a realistic equivalent. The authors show that real-world datasets are typically less structured than their synthetic counterparts, which can results in significantly different benchmarking results, since this level of structuredness will have an impact on how certain data is stored in RDF data management systems. In order to measure this structuredness, the authors introduce the *coherence* metric of a dataset D with a type system $\mathcal{T}$ that can be calculated as follows:

$$\mathrm{CH}(\mathcal{T}, D) = \sum_{\forall T \in \mathcal{T}} \mathrm{WT}(\mathrm{CV}(T, D)) * \mathrm{CV}(T, D) \quad (1)$$

The type system $\mathcal{T}$ contains all the RDF types that are present in a dataset. $\mathrm{CV}(T, D)$ represents the *coverage* of a type T in a dataset D, and is calculated as the fraction of type instances that set a value for all its properties. The factor $\mathrm{WT}(\mathrm{CV}(T, D))$ is used to weight this sum, so that the coherence is always a value between 0 and 1, with 1 representing a perfect structuredness. A maximal coherence means that all instances in the dataset have values for all possible properties in the type system. Based on this metric, the authors introduce a generic method for creating variants of real datasets with different sizes while maintaining a similar structuredness.The authors describe a method to calculate the coverage value of this dataset, which has been implemented as a procedure in the Virtuoso RDF store [24].

2.3. Public Transit Planning

The domain of public transit planning entails the design of public transit networks, rostering of crews, and all the required steps inbetween. The goal is to maximize the quality of service for passengers while minimizing the costs for the operator. Given a public demand and a topological area, this planning process aims to obtain routes, timetables and vehicle and crew assignment. A survey about 69 existing public transit planning approaches shows that these processes are typically subdivided into five sequential steps [16]:

1. **route design**, the placement of transit routes over an existing network.
2. **frequencies setting**, the temporal instantiation of routes based on the available vehicles and estimated demand.
3. **timetabling**, the calculation of arrival and departure times at each stop based on estimated demand.
4. **vehicle scheduling**, vehicle assignment to trips.
5. **crew scheduling and rostering**, the assignment of drivers and additional crew to trips.

In this paper, we only consider the first three steps, which lead to all the required information that is of importance to passengers in a public transit schedule. We present the three steps from this survey in more detail hereafter.

The first step, route design, requires the topology of an area and public demand as input. This topology describes the network in an area, which contains possible stops and edges between these stops. Public demand is typically represented as *origin-destination* (OD) matrices, which contain the number of passengers willing to go from origin stops to destination stops. Given this input, routes are designed based on the following objectives [16]:

area coverage The percentage of public demand that can be served.

route and trip directness A metric that indicates how much the actual trips from passengers deviate from the shortest path.

[1] http://project-hobbit.eu/

demand satisfaction How many stops are close enough to all origin and destination points.

total route length The total distance of all routes, which is typically minimized by operators.

operator-specific objectives Any other constraints the operator has, for example the shape of the network.

historical background Existing routes may influence the new design.

The next step is the setting of frequencies, which is based on the routes from the previous step, public demand and vehicle availability. The main objectives in this step are based on the following metrics [16]:

demand satisfaction How many stops are serviced frequently enough to avoid overcrowding and long waiting times.

number of line runs How many times each line is serviced – a trade-off between the operator's aim for minimization and the public demand for maximization.

waiting time bounds Regulation may put restrictions on minimum and maximum waiting times between line runs.

historical background Existing frequencies may influence the new design.

The last important step for this work is timetabling, which takes the output from the previous steps as input, together with the public demand. The objectives for this step are the following:

demand satisfaction Total travel time for passengers should be minimized.

transfer coordination Transfers from one line to another at a certain stop should be taken into account during stop waiting times, including how many passengers are expected to transfer.

fleet size The total amount of available vehicles and their usage will influence the timetabling possibilities.

historical background Existing timetables may influence the new design.

2.4. Transit Feed Formats

The de-facto standard for public transport time schedules is the General Transit Feed Specification (GTFS)[2]. GTFS is an exchange format for transit feeds, using a series of CSV files contained in a ZIP file. The specification uses the following terminology to define the rules for a public transit system:

Stop is a geospatial location where vehicles stop and passengers can get on or off, such as platform 3 in the train station of Brussels.

Stop time indicates a scheduled arrival and departure time at a certain stop.

Route is a time-independent collection of stops, describing the sequence of stops a certain vehicle follows in a certain public transit line. For example the train route from Brussels to Ghent.

Trip is a collection of stops with their respective stop times, such as the route from Brussels to Ghent at a certain time.

The ZIP file is put online by a public transit operator, to be downloaded by route planning [11] software. Two models are commonly used to then extract these rules into a graph [22]. In a *time-expanded model*, a large graph is modeled with arrivals and departures as nodes and edges connect departures and arrivals together. The weights on these edges are constant. In a *time-dependent model*, a smaller graph is modeled in which vertices are physical stops and edges are transit connections between them. The weights on these edges change as a function of time. In both models, Dijkstra and Dijkstra-based algorithms can be used to calculate routes.

In contrast to these two models, the Connection Scan Algorithm [12] takes an ordered array representation of *connections* as input. A connection is the actual departure time at a stop and an arrival at the next stop. These connections can be given a URI, and described using RDF, using the Linked Connections [8] ontology. For this base algorithm and its derivatives, a connection object is the smallest building block of a transit schedule. In this work, we use the Linked Connections ontology to describe our generated time schedules.

3. Research Question

In order to generate public transport networks and schedules, we start from the hypothesis that both are correlated with the population distribution within the same area. More populated areas are expected to have more nearby and more frequent access to public transport, corresponding to the recurring demand satisfaction objective in public transit planning [16]. When we calculate the correlation between the distribution of stops in an area and its population distribution, we discover

[2] https://developers.google.com/transit/gtfs/

a positive correlation[3] of 0.439 for Belgium and 0.459 for the Netherlands, thereby validating our hypothesis with a confidence of 99%. Because of the continuous population variable and the binary variable indicating whether or not there is a stop, the correlation is calculated using the point-biserial correlation coefficient[4]. For the calculation of these correlations, we ignored the population value outliers. Following this conclusion, our mimicking algorithm will use such population distributions as input, and derive public transport networks and trip instances.

The main objective of a mimicking algorithm is its ability to create *realistic* data. We will measure dataset realism by first comparing the level of structuredness of real-world datasets compared to their synthetic variants using the *coherence metric* introduced by Duan et al. [14]. Furthermore, we will measure the realism of different characteristics within public transport datasets, such as the location of stops, density of the network of stops, length of routes or the frequency of connections. We will quantify these aspects by measuring the distance of each aspect between real and synthetic datasets. This macroscopic coherence metric together with domain-specific microscopic metrics will provide a way to evaluate dataset realism.

Based on this, we introduce the following research question for this work: "Can population distribution data be used to generate realistic synthetic public transport networks and scheduling?"

4. Method

In order to formulate an answer to our research question, we designed a mimicking algorithm that generates realistic synthetic public transit feeds. We based it on techniques from the domains of public transit planning, spatiotemporal and RDF dataset generation. We reuse the route design, frequencies setting and timetabling steps from the domain public transit planning, but prepend this with a network generation phase.

Figure 1 shows the model of the generated public transit feeds, with connections being the primary data element. We consider different properties in this model based on the independent, intra-record or inter-record dependency rules [20], as discussed in Section 2. The arrival time in a connection can be represented as a fully intra-record dependency, because it depends on the time it departed and the stops it goes between. The departure time in a connection is both an intra-record and inter-record dependency, because it depends on the stop at which it departs, but also on the arrival time of the connection before it in the trip. Furthermore, the delay value can be seen as an inter-record dependency, because it is influenced by the delay value of the previous connection in the trip. Finally, the geospatial location of a stop depends on the location of its parent station, so this is also an inter-record dependency. All other unmentioned properties are independent.

In order to generate data based on these dependency rules, our algorithm is subdivided in five steps:

1. **Region**: Creation of a two-dimensional area of cells annotated with population density information.
2. **Stops**: Placement of stops in the area.
3. **Edges**: Connecting stops using edges.
4. **Routes**: Generation of routes between stops by combining edges.
5. **Trips**: Scheduling of timely trips over routes.

These steps are not fully sequential, since stop generation is partially executed before and after edge generation. The first three steps are required to generate a network, step 4 corresponds to the route design step in public transit planning and step 5 corresponds to both the frequencies setting and timetabling. These steps are explained in the following subsections.

4.1. Region

In order to create networks, we sample geographic regions in which such networks exist as two-dimensional matrices. The resolution is defined as a configurable number of cells per square of one latitude by one longitude. Network edges are then represented as links between these cells. Because our algorithm is population distribution-based, each cell contains a population density. These values can either be based on real population information from countries, or this can be generated based on certain statistical distributions. For the remainder of this paper, we will reuse the population distribution from Belgium as a running example, as illustrated in Figure 2.

4.2. Stops

Stop generation is divided into two steps. First, stops are placed based on population values, then the edge

[3] p-values in both cases < 0.00001

[4] `https://github.com/PoDiGG/podigg-evaluate/blob/master/stats/correlation.r`

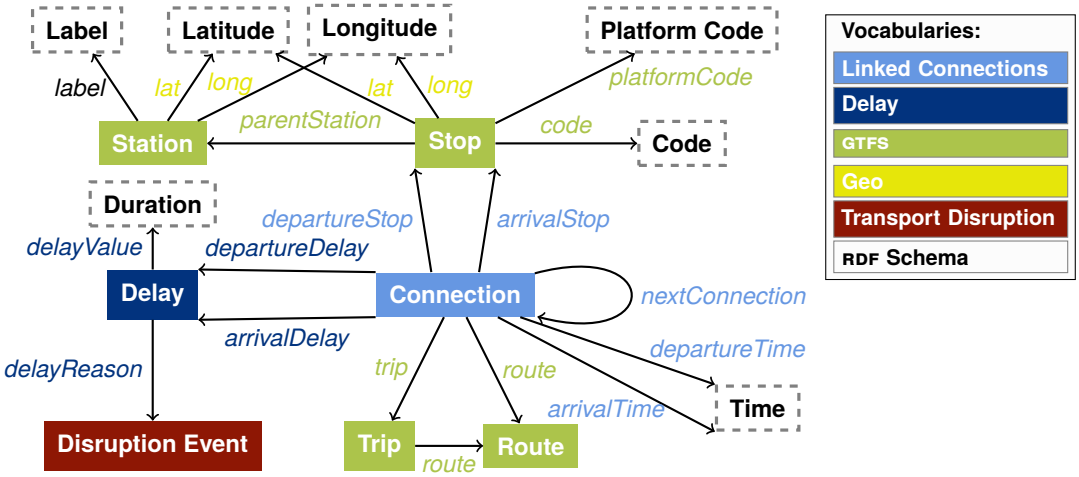

Figure 1. The resources (rectangle), literals (dashed rectangle) and properties (arrows) used to model the generated public transport data. Node and text colors indicate vocabularies.

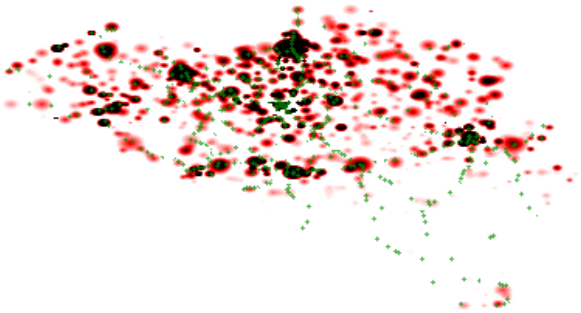

Figure 2. Heatmap of the population distribution in Belgium, which is illustrated for each cell as a scale going from white (low), to red (medium) and black (high). The actual placement of train stops are indicated as green points.

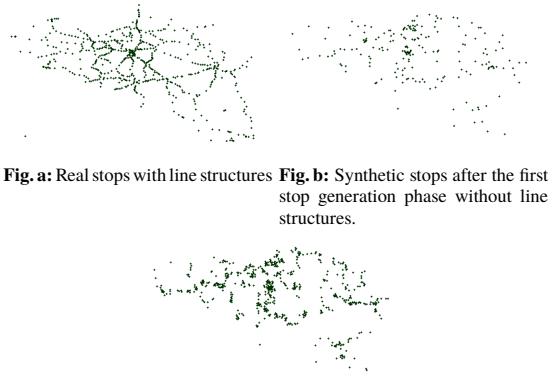

Fig. a: Real stops with line structures **Fig. b:** Synthetic stops after the first stop generation phase without line structures.

Fig. c: Synthetic stops after the second stop generation phase with line structures.

Figure 3. Placement of train stops in Belgium, each dot represents one stop.

generation step is initiated after which the second phase of stop generation is executed where additional stops are created based on the generated edges.

Population-based For the initial placement of stops, our algorithm only takes a population distribution as input. The algorithm iteratively selects random cells in the two-dimensional area, and tags those cells as stops. To make it region-based [6], the selection uses a weighted Zipf-like-distribution, where cells with high population values have a higher chance of being picked than cells with lower values. The shape of this Zipf curve can be scaled to allow for different stop distributions to be configured. Furthermore, a minimum distance between stops can be configured, to avoid situations where all stops are placed in highly population areas.

Edge-based Another stop generation phase exists after the edge generation because real transit networks typically show line artifacts for stop placement. Figure 3a shows the actual train stops in Belgium, which clearly shows line structures. Stop placement after the first generation phase results can be seen in Figure 3b, which does not show these line structures. After the second stop generation phase, these line structures become more apparent as can be seen in Figure 3c. In this second stop generation phase, edges are modified so that sufficiently populated areas will be included in paths formed by edges, as illustrated by Figure 4. Random edges will iteratively be selected, weighted by the edge

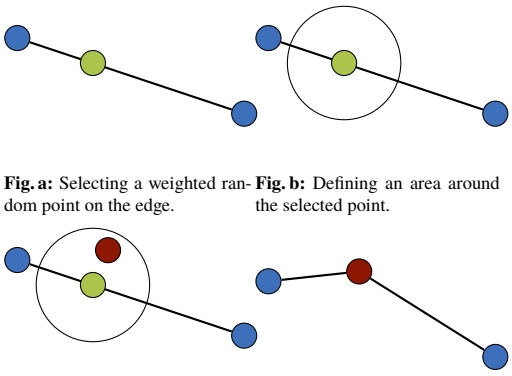

Fig. a: Selecting a weighted random point on the edge.

Fig. b: Defining an area around the selected point.

Fig. c: Choosing a random point within the area, weighted by population value.

Fig. d: Modify edges so that the path includes this new point.

Figure 4. Illustration of the second phase of stop generation where edges are modified to include sufficiently populated areas in paths.

length measured as Euclidian distance[5]. On each edge, a random cell is selected weighed by the population value in the cell. Next, a weighed random point in a certain area around this point is selected. This selected point is marked as a stop, the original edge is removed and two new edges are added, marking the path between the two original edge nodes and the newly selected node.

4.3. Edges

The next phase in public transit network generation connects the stops that were generated in the previous phase with edges. In order to simulate real transit network structures, we split up this generation phase into three sequential steps. In the first step, clusters of nearby stops are formed, to lay the foundation for short-distance routes. Next, these local clusters are connected with each other, to be able to form long-distance routes. Finally, a cleanup step is in place to avoid abnormal edge structures in the network.

Short-distance The formation of clusters with nearby stations is done using agglomerative hierarchical clustering. Initially, each stop is part of a seperate cluster, where each cluster always maintains its *center* point that represents the average location of all stops in that cluster. The clustering step will iteratively try to merge two clusters with their center location distance below a certain threshold. This threshold will increase for each iteration, until a maximum value is reached. This threshold growth is to make sure that nearby clusters are merged before more distant clusters. The maximum distance value indicates the maximum inter-stop distance for forming local clusters. When merging two clusters, an edge is added between the closest stations from the respective clusters. The center location of the new cluster is also recalculated before the next iteration.

Long-distance At this stage, we have several clusters of nearby stops. Because all stops need to be reachable from all stops, these separate clusters also need to be connected. This problem is related to the domain of route planning over public transit networks, in which networks can be decomposed into smaller clusters of nearby stations to improve the efficiency of route planning. Each cluster contains one or more *border stations* [3], which are the only points through which routes can be formed between different clusters. We reuse this concept of border stations, by iteratively picking a random cluster, identifying its closest cluster based on the minimal possible stop distance, and connecting their border stations using a new edge. After that, the two clusters are merged. The iteration will halt when all clusters are merged and there is only one connected graph.

Cleanup The final cleanup step will make sure that the amount of stops that are connected by only one edge are reduced. In real train networks, the majority of stations are connected with at least more than one other stations. The two earlier generation steps however generate a significant amount of *loose stops*, which are connected with only a single other stop with a direct edge. In this step, these loose stops are identified, and an attempt is made to connect them to other nearby stops. For each loose stop, this is done by identifying the direction of the single edge, looking in the opposite direction for other stops, and connecting them with edges if there are such nearby stops. Algorithm 1 explains this mechanism in more detail.

Figure 5 shows an example of these three steps. After this phase, a network with stops and edges is available, and the actual transit planning can commence.

Generator Objectives The main guaranteed objective of the edge generator is that the stops form a single connected transit network graph. This is to ensure that all stops in the network can be reached from any other stop using at least one path through the network.

[5] The Euclidian distance is always used to calculate distances in this work.

Function `RemoveLooseStops`(S, E, N, θ, ϑ)
  Input: Set of stops S;
  Set of edges E between the stops from S;
  Maximum number N of closest stations to consider;
  Maximum average distance θ around a stop to be considered a loose station;
  Radius ϑ in which to look for stops.
foreach $s \in S$ *with degree of 1 w.r.t.* E **do**
  s_x = x coordinate of s; s_y = y coordinate of s;
  Λ = N closest stations to s in S excluding s;
  λ = closest station to s in S excluding s;
  δ = average distance between each pair of stops in Λ;
  if $\delta <= \theta$ *and* Λ *not empty* **then**
    $\phi_x = (s_x - \lambda_x) * \vartheta$; $\phi_y = (s_y - \lambda_y) * \vartheta$;
    $o_x = s_x$; $o_y = s_y$;
    while $i++ < I$ *and distance between* o *and* $s < \delta$ **do**
      o_x += ϕ_x; o_y += ϕ_y;
      s' = random station around o with radius $\delta * \vartheta$;
      if s' *exists*, add edge between s and s' to E and continue next for-loop iteration;

Algorithm 1: Reduce the number of loose stops by adding additional edges.

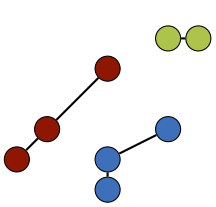

Fig. a: Formation of local clusters

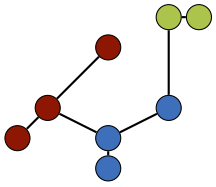

Fig. b: Connecting clusters through border stations

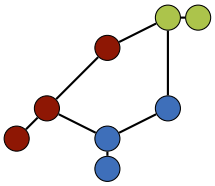

Fig. c: Cleanup of loose stops

Figure 5. Example of the different steps in the edges generation algorithm

4.4. Routes

Given a network of stops and edges, this phase generates routes over the network. This is done by creating short and long distance routes in two sequential steps.

Short-distance The goal of the first step is to create short routes where vehicles deliver each passed stop. This step makes sure that all edges are used in at least one route, this ensures that each stop can at least be reached from each other stop with one or more transfers to another line. The algorithm does this by first determining a subset of the largest stops in the network, based on the population value. The shortest path from each large stop to each other large stop through the network is determined. if this shortest path is shorter than a predetermined value in terms of the number of edges, then this path is stored as a route, in which all passed stops are considered as actual stops in the route. For each edge that has not yet been passed after this, a route is created by iteratively adding unpassed edges to the route that are connected to the edge until an edge is found that has already been passed.

Long-distance In the next step, longer routes are created, where the transport vehicle not necessarily halts at each passed stop. This is done by iteratively picking two stops from the list of largest stops using the network-based method [6] with each stop having an equal chance to be selected. A heuristical shortest path algorithm is used to determine a route between these stops. This algorithm searches for edges in the geographical direction of the target stop. This is done to limit the complexity of finding long paths through potentially large networks. A random amount of the largest stops on the path are selected, where the amount is a value between a minimum and maximum preconfigured route length. This iteration ends when a predetermined number of routes are generated.

Generator Objectives This algorithm takes into account the objectives of route design [16], as discussed

in Section 2. More specifically, by first focusing on the largest stops, a minimal level of *area coverage* and *demand satisfaction* is achieved, because the largest stops correspond to highly populated areas, which therefore satisfies at least a large part of the population. By determining the shortest path between these largest stops, the *route and trip directness* between these stops is optimal. Finally, by not instantiating all possible routes over the network, the *total route length* is limited to a reasonable level.

4.5. Trips

A time-agnostic transit network with routes has been generated in the previous steps. In this final phase, we temporally instantiate routes by first determining starting times for trips, after which the following stop times can be calculated based on route distances. Instead of generating explicit timetables, as is done in typical transit scheduling methodologies, we create fictional rides of vehicles. In order to achieve realistic trip times, we approximate real trip time distributions, with the possibility to encounter delays.

Trip Starting Times The trips generator iteratively creates new connections until a predefined number is reached. For each connection, a random route is selected with a larger chance of picking a long route. Next, a random start time of the connection is determined. This is done by first picking a random day within a certain range. After that, a random hour of the day is determined using a preconfigured distribution. This distribution is derived from the public logs of iRail[6], a route planning API in Belgium [7]. A seperate hourly distribution is used for weekdays and weekends, which is chosen depending on the random day that was determined.

Stop Times Once the route and the starting time have been determined, different stop times across the trip can be calculated. For this, we take into account the following factors:

- Maximum vehicle speed ω, preconfigured constant.
- Vehicle acceleration ς, preconfigured constant.
- Connection distance δ, Euclidian distance between stops in network.
- Stop size σ, derived from population value.

For each connection in the trip, the time it takes for a vehicle to move between the two stops over a certain distance is calculated using the formula in Equation 4. This formula simulates the vehicle speeding up until its maximum speed, and slowing down again until it reaches its destination. When the distance is too short, the vehicle will not reach its maximum speed.

$$T_\omega = \omega/\varsigma \tag{2}$$

$$\delta_\omega = T_\omega^2 \cdot \varsigma \tag{3}$$

$$\text{duration} = \begin{cases} 2T_\omega + (\delta - 2\delta_\omega)/\omega & \text{if } \delta_\omega < \delta/2 \\ \sqrt{2\delta/\varsigma} & \text{otherwise} \end{cases} \tag{4}$$

Not only the connection duration, but also the waiting times of the vehicle at each stop are important for determining the stop times. These are calculated as a constant minimum waiting time together with a waiting time that increases for larger stop sizes σ, this increase is determined by a predefined growth factor.

Delays Finally, each connection in the trip will have a certain chance to encounter a delay. When a delay is applicable, a delay value is randomly chosen within a certain range. Next to this, also a cause of the delay is determined from a preconfigured list. These causes are based on the Traffic Element Events from the Transport Disruption ontology[7], which contains a number of events that are not planned by the network operator such as strikes, bad weather or animal collisions. Delays are carried over to next connections in the trip, with again a chance of encountering additional delay. Furthermore, these delay values can also be reduced when carried over to the next connection by a certain predetermined factor, which simulates the attempt to reduce delays by letting vehicles drive faster.

Generator Objectives For trip generation, we take into account several objectives from the setting of frequencies and timetabling from transit planning [16]. By instantiating more long distance routes, we aim to increase *demand satisfaction* as much as possible, because these routes deliver busy and populated areas, and the goal is to deliver these more frequently. Furthermore, by taking into account realistic time distributions for trip instantiation, we also adhere to this objective. Secondly, by ensuring waiting times at each stop that are longer

[6] https://hello.irail.be

[7] https://transportdisruption.github.io/

for larger stations, the *transfer coordination* objective is taken into account to some extent.

5. Implementation

In this section, we discuss the implementation details of PODIGG, based on the generator algorithm introduced in Section 4. PODIGG is split up into two parts: the main PODIGG generator, which outputs GTFS data, and PODIGG-LC, which depends on the main generator to output RDF data. We first start by listing a set of functional and non-functional requirements for out implementation. The two generator parts will be explained hereafter, followed by a section on how the generator can be configured using various parameters.

5.1. Requirements

For our implementation, we define a set of functional and non-functional requirements.

Functional Requirement 1: Mimicking transit feeds

PODIGG must be able to generate public transport datasets based on the mimicking algorithm that was introduced in Section 4. This means that given a population distribution of a certain region, the generator must be able to design a network of routes, and determine timely trips over this network. Population distributions must be accepted in a standard format.

Functional Requirement 2: Mimicking querysets

Based on the generated transit feed, PODIGG must be able to generate querysets, mimicking realistic route planning requests. Such querysets can then be used as input to route planning systems [11] in order to evaluate them.

Functional Requirement 3: Configurability

The generator should be able to accept parameters to adjust the characteristics of the generated output across different levels. A *seed* parameter must be present to initialize the pseudorandom generator, which results in deterministic output for equal seeds.

Functional Requirement 4: RDF and GTFS serialization

The generated transit feed should be serialized as RDF using existing ontologies and also in the GTFS format. Serialization in RDF using existing ontologies, such as the GTFS[8] and Linked Connections[9] ontologies, allows this inherently linked data to be used within RDF data management systems, where it can for instance be used for benchmarking purposes. Providing output in GTFS will allow this data to be used directly within all systems that are able to handle transit feeds, such as route planning systems.

Functional Requirement 5: Visualizability

Visualization of the generated transit feed must be provided, which will allow the user to have an overview of the network topology and the density of timely connections.

Non-Functional Requirement 1: Usability

The generator should be able to be used without having to understand the internals of the software or requiring any advanced programming skills. The generator must be able to be invoked with a single command accepting an optional configuration of the desired transit feed output.

Non-Functional Requirement 2: Realism

The generator should be able to be configured so that it can generate realistic output.

Non-Functional Requirement 3: Efficiency

The generation phase must be able to output data at a rate of at least 10 million connections per hour. A high rate like this makes it easier for benchmark developers to quickly generate datasets with different parameters as a way to easily explore a large range of dataset types.

Non-Functional Requirement 4: Portability

The generator should be able to be used on all major operating systems requiring no complex installation phase.

Non-Functional Requirement 5: Open-source

The generator must be available under an open license in order make the software usable for anyone and anything and to promote improvements by the open-source community.

Non-Functional Requirement 6: Transparency

While the generator is running, the user should be able to see the progress of the generation phases. This is to give information to the user on how long the generation will still take, and whether it encountered any issues.

Non-Functional Requirement 7: Composability, Extensibility

[8] http://vocab.gtfs.org/terms

[9] http://semweb.mmlab.be/ns/linkedconnections

The different generator steps must be able to exist on their own and be composable, so that alternative implementations of certain phases can be plugged in. Furthermore, it must be possible to add additional phases to the main generator.

Non-Functional Requirement 8: Generalizability

The generator must be able to output realistic data for at least two distinct transit feed types and for two different regions.

5.2. PODIGG

PODIGG is implemented in JavaScript using Node.js, and is available under an open license on GitHub[10]. Furthermore, PODIGG is available as a Node module on NPM[11] and as a Docker image on Docker Hub[12]. Every sub-generator that was explained in Section 4, is implemented as a separate module. This makes PODIGG highly modifiable and compossable, because different implementations of sub-generators can easily be added and removed. Furthermore, this flexible composition makes it possible to use real data instead of certain sub-generators. This can be useful for instance when a certain public transport network is already available, and only the trips and connections need to be generated.

All sub-generators store generated data in-memory, using list-based data structures directly corresponding to the GTFS format. This makes GTFS serialization a simple and efficient process. Table 1 shows the GTFS files that are generated by the different PODIGG modules. This table does not contain references to the region and edges generator, because they are only used internally as prerequisites to the later steps. All required files are created to have a valid GTFS dataset. Next to that, the optional file for exceptional service dates is created. Furthermore, `delays.txt` is created, which is not part of the GTFS specification. It is an extension we provide in order to serialize delay information about each connection in a trip. These delays are represented in a CSV file containing columns for referring to a connection in a trip, and contains delay values in milliseconds and a certain reason per connection arrival and departure, as shown in Listing 1.

In order to easily observe the network structure in the generated datasets, PODIGG will always produce a figure accompanying the GTFS dataset. Figure 6 shows an example of such a visualization.

File	Generator
`agency.txt`	*Constant*
`stops.txt`	Stops
`routes.txt`	Routes
`trips.txt`	Trips
`stop_times.txt`	Trips
`calendar.txt`	Trips
`calendar_dates.txt`	Trips
`delays.txt`	Trips

Table 1

The GTFS files that are written by PODIGG, with their corresponding sub-generators that are responsible for generating the required data. The files in bold refer to files that are required by the GTFS specification.

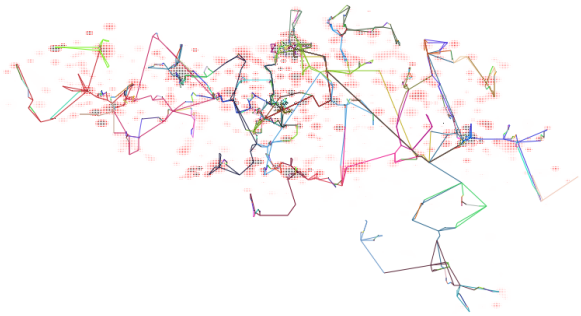

Figure 6. Visualization of a generated public transport network based on Belgium's population distribution. Each route has a different color, and darker route colors indicating more frequent trips over them. The population distribution is illustrated for each cell as a scale going from white (low), to red (medium) and black (high).

Because the generation of large datasets can take a long time depending on the used parameters, PODIGG has a logging mechanism, which provides continuous feedback to the user about the current status and progress of the generator.

Finally, PODIGG provides the option to derive realistic public transit queries over the generated network, aimed at testing the load of route planning systems. This is done by iteratively selecting two random stops weighed by their size and choosing a random starting time based on the same time distribution as discussed in Section 4.5. This is serialized to a JSON format[13] that was introduced for benchmarking the Linked Connections route planner [8].

[10] `https://github.com/PoDiGG/podigg`
[11] `https://www.npmjs.com/package/podigg`
[12] `https://hub.docker.com/r/podigg/podigg/`
[13] `https://github.com/linkedconnections/benchmark-belgianrail#transit-schedules`

```
trip_id,stop_sequence,delay_departure,delay_arrival,delay_departure_reason         ,delay_arrival_reason
100_4  ,0            ,0              ,1405754      ,                               ,td:RepairWork
100_6  ,0            ,0              ,1751671      ,                               ,td:BrokenDownTrain
100_6  ,1            ,1751671        ,1553820      ,td:BrokenDownTrain             ,td:BrokenDownTrain
100_7  ,0            ,2782295        ,0            ,td:TreeAndVegetationCuttingWork,
```

Listing 1: Sample of a `delays.txt` file in a GTFS dataset

5.3. PODIGG-LC

PODIGG-LC is an extension of PODIGG, that outputs data in Turtle/RDF using the ontologies shown in Figure 1. It is also implemented in JavaScript using Node.js, and available under an open license on GitHub[14]. PODIGG-LC is also available as a Node module on NPM[15] and as a Docker image on Docker Hub[16]. For this, we extended the GTFS2LC tool[17] that is able to convert GTFS datasets to RDF using the Linked Connections and GTFS ontologies. The original tool serializes a minimal subset of the GTFS data, aimed at being used for Linked Connections route planning over connections. Our extension also serializes trip, station and route instances, with their relevant interlinking. Furthermore, our GTFS extension for representing delays is also supported, and is serialized using a new Linked Connections Delay ontology[18] that we created.

5.4. Configuration

PODIGG accepts a wide range of parameters that can be used to configure properties of the different sub-generators. Table 2 shows an overview of the parameters, grouped by each sub-generator. PODIGG and PODIGG-LC accept these parameters[19] either in a JSON configuration file or via environment variables. Both PODIGG and PODIGG-LC produce deterministic output for identical sets of parameters, so that datasets can easily be reproduced given the configuration. The `seed` parameter can be used to introduce pseudo-randomness into the output.

6. Evaluation

In this section, we discuss our evaluation of PODIGG. We first evaluate the realism of the generated datasets using a constant seed by comparing its coherence to real datasets, followed by a more detailed realism evaluation of each sub-generator using distance functions. Finally, we measure the efficiency and scalability of the generator and discuss practical dataset sizes. All scripts that were used for the following evaluation can be found on GitHub[20]. Our experiments were executed on a 64-bit Ubuntu 14.04 machine with 128 GB of memory and a 24-core 2.40 GHz CPU.

6.1. Coherence

Metric In order to determine the realism of synthetic RDF datasets, the coherence metric [14] measures the structuredness of a dataset. In RDF dataset generation, the goal is to reach a level of structuredness similar to real datasets. We have implemented a command-line tool[21] to calculate the coherence value for any given input dataset.

Results When measuring the coherence of the Belgian railway, buses and Dutch railway datasets, we discover high values, as can be seen in Table 3. These nearly maximal values indicate that there is a very high level of structuredness in these datasets. Most instances have all the possible values, unlike most typical RDF datasets, which have values around or below 0.6 [14]. That is because of the very specialized nature of this dataset, and the fact that they originate from GTFS datasets that have the characteristics of relational databases. Only a very limited number of classes and predicates are used, where almost all instances have the same set of attributes. When generating synthetic datasets using PODIGG with the same amount of stops, routes and connections for the three gold standards, we measure very similar coherence values, with differences ranging from 0.08% to 1.64%. This shows that PODIGG is able to create datasets that are similar in terms of structuredness to real datasets of these types.

[14] https://github.com/PoDiGG/podigg-lc
[15] https://www.npmjs.com/package/podigg-lc
[16] https://hub.docker.com/r/podigg/podigg-lc/
[17] https://github.com/PoDiGG/gtfs2lc
[18] http://semweb.datasciencelab.be/ns/linked-connections-delay/
[19] https://github.com/PoDiGG/podigg#parameters
[20] https://github.com/PoDiGG/podigg-evaluate
[21] https://github.com/PoDiGG/graph-coherence

	Name	Default Value	Description
	seed	1	The random seed
Region	region_generator	isolated	Name of a region generator. (isolated, noisy or region)
	lat_offset	0	Value to add with all generated latitudes
	lon_offset	0	Value to add with all generated longitudes
	cells_per_latlon	100	How many cells go in 1 latitude/longitude
Stops	stops	600	How many stops should be generated
	min_station_size	0.01	Minimum cell population value for a stop to form
	max_station_size	30	Maximum cell population value for a stop to form
	start_stop_choice_power	4	Power for selecting large population cells as stops
	min_interstop_distance	1	Minimum distance between stops in number of cells
	factor_stops_post_edges	0.66	Factor of stops to generate after edges
	edge_choice_power	2	Power for selecting longer edges to generate stops on
	stop_around_edge_choice_power	4	Power for selecting large population cells around edges
	stop_around_edge_radius	2	Radius in number of cells around an edge to select points
Edges	max_intracluster_distance	100	Maximum distance between stops in one cluster
	max_intracluster_distance_growthfactor	0.1	Power for clustering with more distant stops
	post_cluster_max_intracluster_distancefactor	1.5	Power for connecting a stop with multiple stops
	loosestations_neighbourcount	3	Neighbours around a loose station that should define its area
	loosestations_max_range_factor	0.3	Maximum loose station range relative to the total region size
	loosestations_max_iterations	10	Maximum iteration numberto try to connect one loose station
	loosestations_search_radius_factor	0.5	Loose station neighbourhood size factor
Routes	routes	1000	The number of routes to generate
	largest_stations_fraction	0.05	The fraction of stops to form routes between
	penalize_station_size_area	10	The area in which stop sizes should be penalized
	max_route_length	10	Maximum number of edges for a route in the macro-step
	min_route_length	4	Minimum number of edges for a route in the macro-step
Connections	time_initial	0	The initial timestamp (ms)
	time_final	24 * 3600000	The final timestamp (ms)
	connections	30000	Number of connections to generate
	stop_wait_min	60000	Minimum waiting time per stop
	stop_wait_size_factor	60000	Waiting time to add multiplied by station size
	route_choice_power	2	Power for selecting longer routes for connections
	vehicle_max_speed	160	Maximum speed of a vehicle in km/h
	vehicle_speedup	1000	Vehicle speedup in km/(h^2)
	hourly_weekday_distribution	... [1]	Hourly connection chances for weekdays
	hourly_weekend_distribution	... [1]	Hourly connection chances for weekend days
	delay_chance	0	Chance for a connection delay
	delay_max	3600000	Maximum delay
	delay_choice_power	1	Power for selecting larger delays
	delay_reasons	... [2]	Default reasons and chances for delays
	delay_reduction_duration_fraction	0.1	Maximum part of connection duration to subtract for delays
Queryset	start_stop_choice_power	4	Power for selecting large starting stations
	query_count	100	The number of queries to generate
	time_initial	0	The initial timestamp
	time_final	24 * 3600000	The final timestamp
	max_time_before_departure	3600000	The maximum time until a query should be started
	hourly_weekday_distribution	... [1]	Chance for each hour to have a connection on a weekday
	hourly_weekend_distribution	... [1]	Chance for each hour to have a connection on a weekend day

Table 2

Configuration parameters for the different sub-generators. Time values are represented in milliseconds. [1] Time distributions are based on public route planning logs [7]. [2] Default delays are based on the Transport Disruption ontology (`https://transportdisruption.github.io/`).

	Belgian railway	Belgian buses	Dutch railway
Real	0.9845	0.9969	0.9862
Synthetic	0.9879	0.9805	0.9870
Difference	0.0034	0.0164	0.0008

Table 3
Coherence values for three gold standards compared to the values for equivalent synthetic variants.

6.2. Distance to Gold Standards

While the coherence metric is useful to compare the level of structuredness between datasets, it does not give any detailed information about how real synthetic datasets are in terms of their *distance* to the real datasets. In this case, we are working with public transit feeds with a known structure, so we can look at the different datasets aspects in more detail. More specifically, we start from real geographical areas with their population distributions, and consider the distance functions between stops, edges, routes and trips for the synthetic and gold standard datasets. In order to check the applicability of PODIGG to different transport types and geographical areas, we compare with the gold standard data of the Belgian railway, the Belgian buses and the Dutch railway. The scripts that were used to derive these gold standards from real-world data can be found on GitHub[22].

In order to construct distance functions for the different generator elements, we consider several helper functions. The function in Equation 5 is used to determine the closest element in a set of elements B to a given element a, given a distance function f. The function in Equation 6 calculates the distance from all elements in A to all elements in B, and the other way around given a distance function f. The computational complexity of χ is $O(\|B\| \cdot \kappa(f))$, where $\kappa(f)$ is the cost for one distance calculation for f. The complexity of Δ then becomes $O(\|A\| \cdot \|B\| \cdot \kappa(f))$.

$$\chi(a, B, f) \coloneqq \arg\min_{b \in B} f(a, b) \tag{5}$$

$$\Delta(A, B, f) \coloneqq \frac{\sum_{a \in A} f(a, \chi(a, B, f)) + \sum_{b \in B} f(b, \chi(b, A, f))}{\|A\| + \|B\|} \tag{6}$$

[22] https://github.com/PoDiGG/population-density-generator

Stops Distance For measuring the distance between two sets of stops S_1 and S_2, we introduce the distance function from Equation 7. This measures the distance between every possible pair of stops using the Euclidian distance function d. Assuming a constant execution time for $\kappa(d)$, the computational complexity for Δ_s is $O(\|S_1\| \cdot \|S_2\|)$.

$$\Delta_s(S_1, S_2) \coloneqq \Delta(S_1, S_2, d) \tag{7}$$

Edges Distance In order to measure the distance between two sets of edges E_1 and E_2, we use the distance function from Equation 8, which measures the distance between all pairs of edges using the distance function d_e. This distance function d_e, which is introduced in Equation 9, measures the Euclidian distance between the start and endpoints of each edge, and between the different edges, weighed by the length of the edges. The computational cost of d_e can be considered as a constant, so the complexity of Δ_e becomes $O(\|E_1\| \cdot \|E_2\|)$.

$$\Delta_e(E_1, E_2) \coloneqq \Delta(E_1, E_2, d_e) \tag{8}$$

$$\begin{aligned} d_e(e_1, e_2) \coloneqq &\min(d(e_1^{\text{from}}, e_2^{\text{from}}) + d(e_1^{\text{to}}, e_2^{\text{to}}), \\ &d(e_1^{\text{from}}, e_2^{\text{to}}) + d(e_1^{\text{to}}, e_2^{\text{from}})) \\ &\cdot (d(e_1^{\text{from}}, e_1^{\text{to}}) - d(e_2^{\text{from}}, e_2^{\text{to}}) + 1) \end{aligned} \tag{9}$$

Routes Distance Similarly, the distance between two sets of routes R_1 and R_2 is measured in Equation 10 by applying Δ for the distance function d_r. Equation 11 introduces this distance function d_r between two routes, which is calculated by considering the edges in each route and measuring the distance between those two sets using the distance function Δ_e from Equation 8. By considering the maximum amount of edges per route as $e_{\max}$, the complexity of d_r becomes $O(e_{\max}^2)$ This leads to a complexity of $O(\|R_1\| \cdot \|R_2\| \cdot e_{\max}^2)$ for Δ_r.

$$\Delta_r(R_1, R_2) \coloneqq \Delta(R_1, R_2, d_r) \tag{10}$$

$$d_r(r_1, r_2) \coloneqq \Delta_e(r_1^{\text{edges}}, r_2^{\text{edges}}) \tag{11}$$

Connections Distance Finally, we measure the distance between two sets of connections C_1 and C_2 using the function from Equation 12. The distance between two connections is measured using the function from Equation 13, which is done by considering their respective temporal distance weighed by a constant ϵ [23], and their geospatial distance using the edge distance function d_e. The complexity of time calculation in d_c can be considered being constant, which makes it overall complexity $O(e_{\text{max}}^2)$. For Δ_c, this leads to a complexity of $O(\|C_1\| \cdot \|C_2\| \cdot e_{\text{max}}^2)$.

$$\Delta_c(C_1, C_2) \coloneqq \Delta(C_1, C_2, d_c) \tag{12}$$

$$\begin{aligned} d_c(c_1, c_2) \coloneqq ((&c_1^{\text{departureTime}} - c_2^{\text{departureTime}}) \\ &+ (c_1^{\text{arrivalTime}} - c_2^{\text{arrivalTime}})/\epsilon) \\ &+ d_e(c_1, c_2) \end{aligned} \tag{13}$$

Computability When using the introduced functions for calculating the distance between stops, edges, routes or connections, execution times can become long for a large number of elements because of their large complexity. When applying these distance functions for realistic numbers of stops, edges, routes and connections, several optimizations should be done in order to calculate these distances in a reasonable time. A major contributor for these high complexities is χ for finding the closest element from a set of elements to a given element, as introduced in Equation 5. In practice, we only observed extreme execution times for the respective distance between routes and connections. For routes, we implemented an optimization, with the same worst-case complexity, that indexes routes based on their geospatial position, and performs radial search around each route when the closest one from a set of other routes should be found. For connections, we consider the linear time dimension when performing binary search for finding the closest connection from a set of elements.

[23] When serializing time in milliseconds, we set ϵ to 60000.

Fig. a: Random **Fig. b:** Generated **Fig. c:** Gold standard

Figure 7. Stops for the Belgian railway case.

Metrics In order to measure the realism of each generator phase, we introduce a *realism* factor ρ for each phase. These values are calculated by measuring the distance from randomly generated elements to the gold standard, divided by the distance from the actually generated elements to the gold standard, as shown below for respectively stops, edges, routes and connections.

$$\rho_s(S_{\text{rand}}, S_{\text{gen}}, S_{\text{gs}}) \coloneqq \Delta_s(S_{\text{rand}}, S_{\text{gs}})/\Delta_s(S_{\text{gen}}, S_{\text{gs}})$$
$$\rho_e(E_{\text{rand}}, E_{\text{gen}}, E_{\text{gs}}) \coloneqq \Delta_e(E_{\text{rand}}, E_{\text{gs}})/\Delta_e(E_{\text{gen}}, E_{\text{gs}})$$
$$\rho_r(R_{\text{rand}}, R_{\text{gen}}, R_{\text{gs}}) \coloneqq \Delta_r(R_{\text{rand}}, R_{\text{gs}})/\Delta_r(R_{\text{gen}}, R_{\text{gs}})$$
$$\rho_c(C_{\text{rand}}, C_{\text{gen}}, C_{\text{gs}}) \coloneqq \Delta_c(C_{\text{rand}}, C_{\text{gs}})/\Delta_c(C_{\text{gen}}, C_{\text{gs}})$$

Results We measured these realism values with gold standards for the Belgian railway, the Belgian buses and the Dutch railway. In each case, we used an optimal set of parameters[24] to achieve the most realistic generated output. Table 4 shows the realism values for the three cases, which are visualized in Figures 7 to 10. Each value is larger than 1, showing that the generator at least produces data that is closer to the gold standard, and is therefore more realistic. The realism for edges is in each case very large, showing that our algorithm produces edges that are very similar to actual the edge placement in public transport networks according to our distance function. Next, the realism of stops and routes is lower, but still sufficiently high to consider them as realistic. Finally, the values for connections show that this sub-generator produces output that is closer to the gold standard than the random function according to our distance function. For the Belgian railway case, the connections are only slightly closer, while for the other cases the realism is more significant. The combined realism values show that PODIGG is able to produce realistic data for different regions and different transport types.

[24] `https://github.com/PoDiGG/podigg-evaluate/blob/master/bin/evaluate.js`

	Belgian railway	Belgian buses	Dutch railway
Stops	5.5490	297.0888	4.0017
Edges	147.4209	1633.4693	318.4131
Routes	2.2420	1.6094	1.3095
Connections	1.0451	1.5006	1.3017
Total	39.0642	483.4170	81.2565

Table 4

Realism values for the three gold standards in case of the different sub-generators, respectively calculated for the stops ρ_s, edges ρ_e, routes ρ_r and connections ρ_c. The last row shows the combined realism with each gold standard, which is the sum of the values for the four sub-generators divided by 4.

Fig. a: Random **Fig. b:** Generated **Fig. c:** Gold standard

Figure 8. Edges for the Belgian railway case.

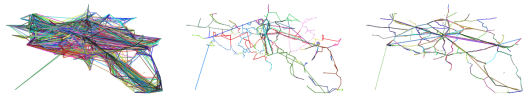

Fig. a: Random **Fig. b:** Generated **Fig. c:** Gold standard

Figure 9. Routes for the Belgian railway case.

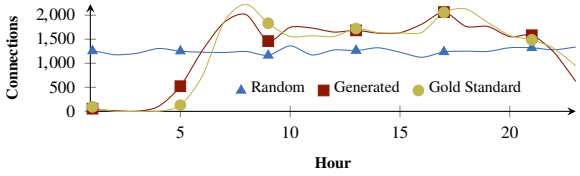

Figure 10. Connections per hour for the Belgian railway case.

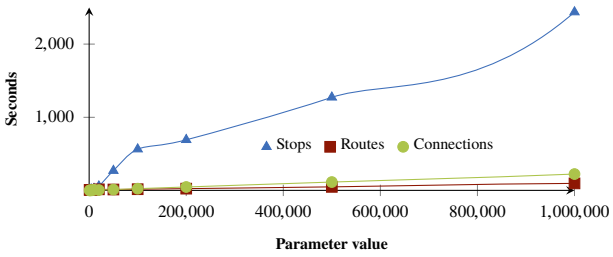

Figure 11. Execution times when increasing the number of stops, routes or connections.

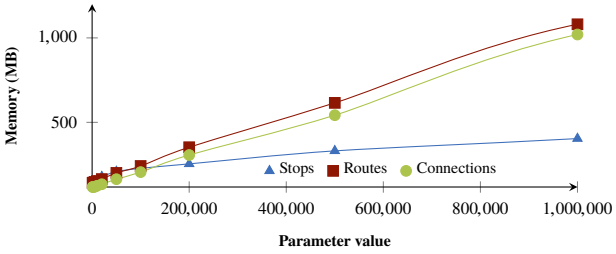

Figure 12. Memory usage when increasing the number of stops, routes or connections.

6.3. Performance

Metrics For validating the efficiency requirement, we measure the impact of different parameters on the execution times of the generator. The three main parameters for increasing the output dataset size are the number of stops, routes and connections. Because the number of edges is implicitly derived from the number of stops in order to reach a connected network, this can not be configured directly. In this section, we start from a set of parameters that produces realistic output data that is similar to the Belgian railway case. We let the value for each of these parameters increase to see the evolution of the execution times and memory usage.

Results Figure 11 shows a linear increase in execution times when increasing the routes or connections. The execution times for stops do however increase much faster, which is caused by the higher complexity of networks that are formed for many stops. The used algorithms for producing this network graph proves to be the main bottleneck when generating large networks. Networks with a limited size can however be generated quickly, for any number of routes and connections. The memory usage results from Figure 12 also show a linear increase, but now the increase for routes and connections is higher than for the connections parameter. These figures show that stops generation is a more CPU intensive process than routes and connections generation. These last two are able to make better usage of the available memory for speeding up the process.

6.4. Dataset size

An important aspect of dataset generation is its ability to output various dataset sizes. In PODIGG, different options are available for tweaking these sizes. Increasing

the time range parameter within the generator increases the number of connections while the number of stops and routes will remain the same. When enlarging the geographical area over the same period of time, the opposite is true. As a rule of thumb, based on the number of triples per connection, stops and routes, the total number of generated triples per dataset is approximately $7 \cdot \#connections + 6 \cdot \#stops + \#routes$. For the Belgian railway case, containing 30,011 connections over a period of 9 months, with 583 stops and 362 routes, this would theoretically result in 213,937 triples. In practice, we reach 235,700 triples when running with these parameters, which is slightly higher because of the other triples that are not taken into account for this simplified formula, such as the ones for trips, stations and delays.

7. Discussion

In this section, we discuss the adherence to the requirements that were defined, followed by the limitations of this work. Finally, we mention the PODIGG use cases.

7.1. Requirements

Our main research question on how to generate realistic synthetic public transport networks has been answered by the introduction of the mimicking algorithm from Section 4. This is based on the accepted hypothesis that the population distribution of an area is correlated with its transport network design and scheduling. Furthermore, we measured the realism of the generated datasets using the coherence metric and more fine-grained realism metrics for different public transport aspects. The functional and non-functional requirements that were set out in Section 5.1 are achieved by the PODIGG and PODIGG-LC implementations, which will be discussed in this section.

In Functional Requirements 1 and 2 we required realistic mimicking of transit feeds and querysets. These are accomplished by the PODIGG generator which is able to do both, based on commonly used practises in transit network design. Furthermore, PODIGG accepts a wide range of parameters to configure the mimicking algorithm, which confirms Functional Requirement 3. Finally, PODIGG and PODIGG-LC are able to output the mimicked data respectively as GTFS and RDF datasets, together with a visualization of the generated transit network. This accomplishes Functional Requirements 4 and 5.

In Non-Functional Requirement 1, we set out that the generator should be able to be used without requiring any extensive setup or advanced programming skills. This has been accomplished by PODIGG and PODIGG-LC, which are simple command lines tools that can be invoked with a number of optional parameters to configure the generator. In Non-Functional Requirement 2, we aimed for realistic public transport datasets. This is confirmed by the evaluation results from Sections 6.1 and 6.2, which showed that generated datasets are structurally similar to real datasets and are close to real datasets according to distance functions on several levels. In Section 6.3 we showed that PODIGG is able to generate connections at a rate of 16 million per hour, exceeding the Non-Functional Requirement 3 of 10 million per hour. Since datasets are typically generated once and reused multiple times afterwards, the generation speed is in most cases not an issue. By providing PODIGG and PODIGG-LC as a Node module and Docker image, this tool can be used on all major operating systems, and we can therefore confirm Non-Functional Requirement 4. As set out in Non-Functional Requirement 5, the PODIGG suite is available on GitHub under an open license, and is sufficiently documented to promote contributions from the open-source community. The continuous logging by PODIGG accomplishes Non-Functional Requirement 6 about transparancy. The modularity of PODIGG achieves Non-Functional Requirement 7, which allows different sub-generator implementations to be composed together. Finally, Non-Functional Requirement 8 is accomplished by the realism evaluations from Section 6.2, in which we evaluated PODIGG for the bus and train transport type, and the Belgium and Netherlands geospatial regions.

7.2. Limitations and Future Work

In this section, we discuss the limitations of the current mimicking algorithm and its implementation, together with further research opportunities.

The sequential steps in the presented mimicking algorithm require persistence of the intermediary data that is generated in each step. Currently, PODIGG is implemented in such a way that all data is kept in-memory for the duration of the generation, until it is serialized. When large datasets need to be generated, this requires a larger amount of memory to be allocated to the generator. Especially for large amounts of routes or connections, where 100 million connections already require almost 10GB of memory to be allocated. A first

possible solution would be to use a memory-mapped database for intermediary data, so that not all data must remain in memory at all times. An alternative solution would be to modify the mimicking process to a streaming algorithm, so that only small parts of data need to be kept in memory for datasets of any size. Considering the complexity of transit networks, a pure streaming algorithm might not be feasible, because route design requires knowledge of the whole network. The generation of connections however, could be adapted so that it works as a streaming algorithm.

We aimed to produce realistic transit feeds by reusing the methodologies learned in public transit planning. Our results showed that all sub-generators, except for the trips generator, produced output with a high realism value. The trips are still closer to real data than a random generator, but this can be further improved in future work. This can be done by for instance taking into account network capacities [16] on certain edges when instantiating routes as trips, because we currently assume infinite edge capacities, which can result in a large amount of connections over an edge at the same time, which may not be realistic for certain networks. Next to this, in order to improve transfer coordination [16], possible transfers between trips should be taken into account when generating stop times. Limiting the network capacity will also lead to natural congestion of networks [6], which should also be taken into account for improving the realism. Furthermore, the total vehicle fleet size [16] should be considered, because we currently assume an infinite number of available vehicles. It is more realistic to have a limited availability of vehicles in a network, with the last position of each vehicle being of importance when choosing the next trip for that vehicle.

An alternative way of implementing this generator would be to define declarative dependency rules for public transport networks, based on the work by Pengyue et. al. [20]. This would require a semantic extension to the engine so that is aware of the relevant ontologies and that it can serialize to one or more RDF formats. Alternatively, machine learning [13] techniques such as deep learning [19] could be used to automatically learn the structure and characteristics of public transport networks and create similar realistic synthetic datasets. The downside of machine learning techniques is however that it is typically more difficult to tweak parameters of automatically learned models when specific characteristics of the output need to be changed, when compared to a manually implemented algorithm. Sensitivity analysis could help to determine the impact of such parameters in order to understand the learned models better.

Finally, the temporal aspect of public transport networks is useful for the domain of RDF stream processing [10]. Instead of producing single static datasets as output, PODIGG could be adapted to produce RDF streams of connections and delays, where information about stops and routes are part of the background knowledge. Such an extension can become part of a benchmark, such as CityBench [1] and LSBench [18], for assessing the performance of RDF stream processing systems with temporal and geospatial capabilities.

7.3. PODIGG In Use

PODIGG and PODIGG-LC have been developed for usage within the HOBBIT platform. This platform is being developed within the HOBBIT project and aims to provide an environment for benchmarking RDF systems for Big Linked Data. The platform provides several default dataset generators, including PODIGG, which can be used to benchmark systems.

PODIGG, and its generated datasets are being used in the ESWC Mighty Storage Challenge 2017[25]. The first task of this challenge consists of RDF data ingestion into triple stores, and querying over this data. Because of the temporal aspect of public transport data in the form of connections, PODIGG datasets are fragmented by connection departure time, and transformed to a data stream that can be inserted. In task 4 of this challenge, the efficiency of faceted browsing solutions is benchmarked. Because of the geographical property of public transport data, PODIGG datasets are being used for this benchmark.

Finally, PODIGG is being used for creating virtual transit networks of variable size for the purposes of benchmarking route planning frameworks, such as Linked Connections [8].

8. Conclusions

In this paper, we introduced a mimicking algorithm for public transport data, based on steps that are used in real-world transit planning. Our method splits up this process into several sub-generators and uses population distributions of an area as input. As part of this paper, we introduce PODIGG, a reusable framework that accepts

[25] https://project-hobbit.eu/challenges/mighty-storage-challenge/

a wide range of parameters to configure the generation algorithm.

Results show that the structuredness of generated datasets are similar to real public transport datasets. Furthermore, we introduced several functions for measuring the realism of synthetic public transport datasets compared to a gold standard on several levels, based on distance functions. The realism was confirmed for different regions and transport types. Finally, the execution times and memory usages were measured when increasing the most important parameters, which showed a linear increase for each parameter, showing that the generator is able to scale to large dataset outputs.

For these reasons, the public transport mimicking algorithm we introduced, with PODIGG and PODIGG-LC as implementations, provides a useful tool for benchmarking with realistic geospatial and temporal RDF datasets. Flexible configuration allows datasets of any size to be created and various characteristics to be tweaked to achieve highly specialized datasets for testing specific use cases.

Acknowledgements

We wish to thank Henning Petzka for his help with discovering issues and providing useful suggestions for the PODIGG implementation. The described research activities were funded by the H2020 project HOBBIT (#688227).

References

[1] M. I. Ali, F. Gao, and A. Mileo. CityBench: a configurable benchmark to evaluate RSP engines using smart city datasets. In *International Semantic Web Conference*, pages 374–389. Springer, 2015.

[2] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev, and I. Toma. The Linked Data Benchmark Council: a graph and RDF industry benchmarking effort. *ACM SIGMOD Record*, 43(1):27–31, 2014.

[3] H. Bast, M. Hertel, and S. Storandt. Scalable transfer patterns.

[4] T. Berners-Lee. Linked Data, July 2006. `http://www.w3.org/DesignIssues/LinkedData.html`.

[5] C. Bizer and A. Schultz. The Berlin SPARQL benchmark. *International Journal on Semantic Web and Information Systems*, 5(2):1–24, 2009.

[6] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[7] P. Colpaert, A. Chua, R. Verborgh, E. Mannens, R. Van de Walle, and A. Vande Moere. What public transit API logs tell us about travel flows. In *Proceedings of the 6th USEWOD Workshop on Usage Analysis and the Web of Data*, pages 873–878, Apr. 2016.

[8] P. Colpaert, A. Llaves, R. Verborgh, O. Corcho, E. Mannens, and R. Van de Walle. Intermodal public transit routing using Linked Connections. In *Proceedings of the 14th International Semantic Web Conference: Posters and Demos*, 2015.

[9] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1: Concepts and abstract syntax. Recommendation, W3C, Feb. 2014.

[10] E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a streaming world! Reasoning upon rapidly changing information. *Intelligent Systems, IEEE*, 24(6), Nov 2009.

[11] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of large and complex networks*, pages 117–139. Springer, 2009.

[12] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. Intriguingly Simple and Fast Transit Routing. In *Experimental Algorithms*, pages 43–54. Springer, 2013.

[13] P. Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[14] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 145–156. ACM, 2011.

[15] G. Garbis, K. Kyzirakos, and M. Koubarakis. Geographica: A benchmark for geospatial RDF stores. In H. Alani, L. Kagal, A. Fokoue, P. Groth, C. Biemann, J. X. Parreira, L. Aroyo, N. Noy, C. Welty, and K. Janowicz, editors, *Proceedings of the 12th International Semantic Web Conference*, pages 343–359. Springer Berlin Heidelberg, 2013.

[16] V. Guihaire and J.-K. Hao. Transit network design and scheduling: A global review. *Transportation Research Part A: Policy and Practice*, 42(10):1251–1273, 2008.

[17] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[18] D. Le-Phuoc, M. Dao-Tran, M.-D. Pham, P. Boncz, T. Eiter, and M. Fink. Linked stream data processing engines: Facts and figures. In *International Semantic Web Conference*, pages 300–312. Springer, 2012.

[19] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[20] P. J. Lin, B. Samadi, A. Cipolone, D. R. Jeske, S. Cox, C. Rendon, D. Holt, and R. Xiao. Development of a synthetic data set generator for building and testing information discovery systems. In *Third International Conference on Information Technology: New Generations (ITNG'06)*, pages 707–712. IEEE, 2006.

[21] M. A. Nascimento, D. Pfoser, and Y. Theodoridis. Synthetic and real spatiotemporal datasets. *IEEE Data Eng. Bull.*, 26(2):26–32, 2003.

[22] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics (JEA)*, 12:2–4, 2008.

[23] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}bench: a SPARQL performance benchmark. In *2009 IEEE 25th International Conference on Data Engineering*, pages 222–233. IEEE, 2009.

[24] M. Spasić, M. Jovanovik, and A. Prat-Pérez. An RDF dataset generator for the social network benchmark with real-world coherence. In I. Fundulaki, A. Krithara, A.-C. Ngonga Ngomo, and V. Rentoumi, editors, *Proceedings of the Workshop on Benchmarking Linked Data*, volume 1700 of *CEUR Workshop Proceedings*, 2016.