TIMETABLE-BASED ROUTING IN FIXED SCHEDULE DYNAMIC NETWORKS

Cristian Oswaldo Rodriguez Santiago

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

March 2021

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By:	**Cristian Oswaldo Rodriguez Santiago**
Entitled:	**Timetable-based Routing in Fixed Schedule Dynamic Networks**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

Dr. J. Opatrny	Chair
Dr. L. Opatrny	Examiner
Dr. D. Pankratov	Examiner
------------------	Examiner
Dr. L. Narayanan	Supervisor

Approved Dr. H. Harutyunyan

Chair of Department or Graduate Program Director

March 25th 20 21

Mourad Debbabi, Ph.D., Interim Dean

Faculty of Engineering and Computer Science

Abstract

Timetable-based Routing in Fixed Schedule Dynamic Networks

Cristian Oswaldo Rodriguez Santiago

A fixed schedule dynamic network has a set of nodes (eg. vehicles or satellites) that move using a known schedule and trajectory, so that the connections between nodes in the network appear and disappear in a predictable manner. We study the problem of finding a foremost journey in such a network: given a query time, source and destination node, we find a temporal path that arrives at the *earliest* time at the destination. We give a new approach to the problem that uses a *timetable* sorted in order of the start time of connections, and describe three algorithms using this approach. We prove their correctness and give tight bounds on their worst-case time complexities. We also show extensive experimental results that show that our new algorithms outperforms the previous best algorithm given in [21].

Acknowledgments

I thank my God for providing me the strength and courage to complete this work.

I would like to express a sincere gratitude and appreciation to my supervisor Dr. Lata Narayanan; for her patience, dedication and commitment.

Thanks to all my family, especially my mother Adelina and my sister Camila, for their love and support. I also would like to manifest a deep appreciation to Dr. Luis Felipe Urquiza, a generous and good friend of mine.

To all my brothers and sisters in Christ members of First Filipino Baptist Church of Montréal and Bible Study Fellowship of Montréal, thank you all for your prayers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Advances in wireless technologies, vehicular networking, and automobiles have made it possible to conceive of heterogeneous vehicular networks comprising vehicles, roadside units, as well as pedestrians. Intelligent transportation systems (ITS) consider vehicles as network components responsible for sending, receiving and routing packets within the vehicular ad hoc network (VANET). Each vehicle is equipped with a radio antenna called an *on-board unit* or OBU. In such network, vehicle connectivity is classified as Vehicle-to-Vehicle (V2V), Vehicle-to-Infrastructure (V2I) and Vehicle-to-Pedestrian (V2P). V2V involves data exchange between two or more vehicles while V2I considers the interaction between vehicles and the road environment. V2P includes pedestrians as potential network members [4]. One emerging type of connectivity is V2X known as Vehicle-to-Everything which also includes communication with IT networks and data centers [31].

Among the applications that VANETs support are urban sensing, interactive entertainment, comfort, efficiency and safety [13]. Due to its potential, Transport Canada created the Advance Connectivity and Automation in the Transportation System (ACATS) program whose aim is to prepare Canadian roads for connected and automated vehicle deployments to support V2V, V2I and V2X communication [4]. For instance, Montréal is currently working with EasyMile [17] and Transdev [27] to implement an automated shuttle near the Olympic Stadium area to reduce the number of on-road collisions [4]. Another V2V application developed in collaboration with many truck companies and supported by U.S. Department of Transportation and the European Union is the SARTRE project which estimates to reduce emissions and increase safety and comfort. It forms road trains on highways where the leading car takes the responsibility for the entire road train and every car mimics its actions. This allows the drivers to focus their attention on other things [39].

In vehicular ad hoc networks, V2V communication is challenging because communication links are transient, causing the network topology to keep changing. There is therefore no permanent route between any two nodes in the network. Indeed at any given time step, it may not be possible to guarantee a route between any given source and destination. The situation could be even worse: for a given source-destination pair, it may be that there is *no* time at which there is a route between them. However, by storing packets at intermediate nodes, it may be possible to deliver the packet eventually to the destination. It is for this reason that such networks have been variously called delay or disruption-tolerant, or intermittently-connected networks over the last couple of decades.

This thesis considers a particular type of vehicular network: *Fixed Schedule Dynamic Network (FSDN)* also known as Time Varying Communication Network (TVCN) [29]. Such a network has a fixed set of nodes, which move in the given terrain using a pre-determined schedule and trajectory, i.e. route/path in the terrain. As two nodes in the network move within transmission range, a communication link or *connection* is created. As the nodes move away from each other, the connection is broken, and may or may not re-appear, depending on the trajectories of the nodes. Indeed a single pair of nodes may be part of many different connections at different times; all these connections are said to correspond to the same *edge.* Since the trajectories of the nodes are known, the network is *dynamic* but in a *predictable* manner; for each connection, we know in advance (roughly) when it will start and when it will end.

Examples of FSDN networks include networks created by public transport buses, Low Earth orbiting satellites (LEO), etc. For instance, Unmanned Aerial Vehicles (UAVs) have a predictable circular flight pattern, so they are able to compute routes withing the UAV network at any time [48] [3] [40]. They can support applications such as tactical military applications, humanitarian aid missions, drug blockage enforcement. DakNet [42] provides internet access in developing nations; some links in the Daknet network can be predicted due to the mobility patterns of public transportation vehicles [28]. LEO satellite networks are classified as FSDN as well since the position of each satellite is deterministic due to its trajectory around the earth and links between satellites can be estimated [19]. Another example where the presence of the links can be estimated is Wireless Sensor Networks (WSNs) where connections between sensors appear and disappear based on a sleep schedule due to their energy constraints [20].

FSDNs can be represented by *dynamic graphs*, which have received a lot of attention recently [29]. There are many models of such graphs that have been studied, and we describe these in Chapter 2. For now, we restrict ourselves to saying that a dynamic graph $\mathcal{G}$ is defined as a pair $(G, \mathcal{S}_G)$ where $G = (V, E)$ is a graph with a set of nodes V and a set of edges E and $\mathcal{S}_G = G_0, G_1, \ldots, G_k$ is an ordered sequence of sub graphs G_i. Each G_i has a set of nodes V_i and a set of edges E_i active in the time interval $[t_i, t_i + 1[$. Two nodes are adjacent in $\mathcal{G}$ if they are adjacent in any G_i. A common representation for such graphs is to use the base graph G, and for each edge, store a sorted list of distinct and maximal time intervals in which it is alive. A *temporal path* in a dynamic graph is a sequence of (v_i, t_i), where each $v_i \in V$, so that the pair (v_i, v_{i+1}) is connected at time t_i, and furthermore, the t_i's form a non-decreasing sequence.

1.2 Problem definition

In this thesis, we study the problem of one-to-one routing in an FSDN represented by an evolving graph $\mathcal{G}$. In particular, at a given time, given a source node s and destination node t in $\mathcal{G}$, and a query time t_q, we wish to compute a temporal path in the network that minimizes the arrival time at the destination. Such a path is sometimes called a *foremost journey* in the network.

We assume that time is slotted into discrete time slots, and that the mobility of the nodes/vehicles/satellites is such that the transmission time for a packet is negligible compared to the minimum duration of a connection. Note that we are not guaranteed a source-destination path in the network at any given time; so while several hops in the network may be traversed almost instantaneously, there may be time intervals where it is not possible to make progress towards the destination.

This problem was introduced in [21], and the authors gave an algorithm based on Dijkstra's shortest path algorithm, that used an evolving graph data structure to represent the network. Their algorithm was shown to take $\mathcal{O}(|E|(\log I + \log |V|))$ time where E is the set of edges and I is the maximum number of time intervals in which an edge is present.

1.3 Our results

We give a new timetable-based approach for the foremost journey computation problem, and describe three variants of this approach. The timetable $\mathcal{T}$ stores the connections that appear in the FSDN in order of their start time.

- Our first algorithm, called Pure-Timetable takes time $\mathcal{O}(|V||E| + |\mathcal{T}|)$ and space $\mathcal{O}(|\mathcal{T}|)$. In addition to the timetable of connections, we use an auxiliary timetable to save connections that were not useful when they started but may prove to be useful later.

- Our second algorithm is a hybrid approach that uses both the timetable of connections, and the evolving graph data structure of [21] and takes $\mathcal{O}(|E|\log I + |\mathcal{T}| + |V|)$ time and space $\mathcal{O}(|\mathcal{T}|)$.

- Our final variant called Timetable with Auxiliary Graph uses both the timetable of connections, and an auxiliary graph data structure created and maintained during processing of the timetable $\mathcal{T}$. This last variant finds the route between any source-destination pair in $\mathcal{O}(|E| + |\mathcal{T}| + |V|)$ time and space $\mathcal{O}(|\mathcal{T}|)$.

Note that $|\mathcal{T}| = \mathcal{O}(|E|I)$. It is clear from the above that Hybrid Timetable-Evolving Graph and Timetable with Auxiliary Graph improve over the classic algorithm of Ferreira [21] in some situations. We also do extensive simulations of our algorithms, using random data sets, as well as actual data sets derived from the *Société de transport de Montréal* and satellite positions that validate the superior performance of our algorithms.

1.4 Organization

In Chapter 2, we review the literature concerning dynamic graph models, journey computation and previous algorithms proposed for our problem and related problems. Chapter 3 defines the data structures and the algorithms that we propose describing each timetable-based variant along with Ferreira's algorithm [21]. In Chapter 4 we detail the procedure to obtain the timetables and evolving graphs which are the inputs for our experiments; then, we provide some metrics to characterize each data set. Finally, Chapter 5 discusses the performance of each algorithm and explains the attributes of the computed journeys; e.g., number of hops, journey duration. We conclude with some directions for future research in Chapter 6.

Chapter 2

Related work

This chapter reviews the literature related to the scope of this work. In Section 2.1, we review the mathematical models that are employed to represent dynamic graphs. Then in Section 2.2, we discuss the previously proposed algorithms related to journey calculation in communication networks. Finally, we describe an algorithm designed to find foremost journeys for passengers in transportation networks, called Connection Scan Algorithm in Section 2.3.

2.1 Dynamic graph models

In the literature, there are two types of work on dynamic graph. In the first type, researchers have considered how to support operations that modify the initial state of the graph such as edge insertion, edge deletion, node insertion and node deletion, as soon as they happen. In this work, a dynamic graph is also known as a streaming graph [5] and is modeled with data structures such as STINGER [18], cuSTINGER [23], EvoGraph [47], AIMS [51], Hornet [7]. As an example, Green's algorithm [22] is a well-known algorithm that updates the value of betweenness centrality [1] metric [24] [49] as soon as a change in the graph topology occurs.

The scope of this work focuses on the second type of dynamic graph where the presence of an edge depends on a given time within the timespan of the graph, and can be known or predicted in advance. For instance, Low Earth Orbiting (LEO)

[1]The betweenness centrality of a node v is defined as $C_B(v) = \sum_{s \neq z \neq v} \sigma_{sz}(v)/\sigma_{sz}$ where σ_{sz} is the number of shortest paths between node s and node z while $\sigma_{sz}(v)$ is the number of shortest paths between node s and node z that pass through node v [24].

satellite systems have links in different orbital planes where each endpoint belongs to a different cluster of satellites. At some point, they are out of range due to their relative movement. Such networks are called Fixed Schedule Dynamic Networks (FSDN); the changes to the topology of such a network and can be predicted ahead of time [21].

There have been efforts to formulate a dynamic graph model that successfully captures the dynamic behavior of a network. However, there is a lack of consensus among those models since the approach to formulate each is different. According to [56], dynamic graph models can be grouped in four main categories: sequence of snapshots, whole graph, log file and distributed graph over servers. The following subsections have a brief overview of each dynamic graph model.

2.1.1 Snapshots

Models based on a sequence of snapshots have a static graph per time step. For example, the authors in [54], propose algorithms to capture the Most Frequently Changing Component (MFCC) given a sequence of snapshots. There is a notion introduced in [43] for sequence of static graphs called Evolving Graph Sequence (EGS). They formulate a framework called FVF for efficient query processing in EGSs. Also, a Dynamic Behavioral Mixed-Membership Model (DBMM) is studied in [45] to identify the roles that nodes play in social networks. Normally, every snapshot in the sequence is encoded as a triple (V_i, E_i, t_i) where the set of nodes V_i and the set of edges E_i are present at t_i time step [41].

The downside of this kind of model is that it requires a lot of space to store each snapshot. However, to overcome this problem and improve its memory performance some techniques have been studied. For instance, the authors in [44] improve the FVF framework including a preprocessing step that consist of identifying similar snapshots and group them into clusters, then two representative graphs are extracted. Similar approaches are proposed in [45] and [57]. Authors in [45] define a method to cluster snapshots that have similar query frequency and [57] formulates a procedure that chooses snapshots with lowest variance.

2.1.2 Whole graph

The idea of the whole graph model is to represent a dynamic graph as one single graph. The key aspect is the introduction of time intervals where nodes or edges are considered present (sometimes termed valid, alive, or active). This can be represented by a presence function [9] for both nodes and edges, which given a specific time instant, outputs if they are alive or not.

The formalization of this whole graph approach is given in [21] and [9], also known as FSDN and Time-Varying Graphs (TVGs), respectively. Depending on the application the model may vary but the approach is the same. For example, the authors in [32] use labels for each edge to denote the time interval when they are alive. Another model similar to TVG called Temporally Evolving Graph (TEG) is described in [26] and [25]. Authors in [35] propose a model called Space-Time Varying Graph (STVG); they define adjacency between nodes based on places and events across time.

The authors of [50] propose what they term a unifying model for representing time-varying graphs. Basically, they claim that their model can unify both whole graph and snapshot models. They represent edges as a quadruple (u, t_a, v, t_b): u and v are the source and target nodes, t_a and t_b represent the time interval. Then, they identify four types of edges based on their temporal characteristics. Spatial edges connect two different nodes at the same time $t_a = t_b$ and $u \neq v$. Temporal edges connect the same node but at different time $t_a \neq t_b$ and $u = v$. Mixed edges link two different nodes at different time $t_a \neq t_b$ and $u \neq v$ and Spatial-Temporal edges link the same node at the time $t_a = t_b$ and $u = v$. Afterwards, they prove that any dynamic graph that is modeled with snapshots or whole graph and has those type of edges can be represented with their unifying model.

2.1.3 Log file

Log file models are similar to snapshot models but they follow a materialization procedure [56]. This procedure stores snapshots along with delta log files to store the changes in the topology. It saves memory space since it is not required to store the whole snapshot but it requires an additional processing power to construct the snapshot using the log files.

There are three strategies to materialize snapshots: Time-based, Operation-based and Similarity-based [56] [52]. Time-based materialization fixes the elapsed time between two materialized snapshots. Operation-based materialization establishes a constant number of events to materialize a snapshot. Similarity-based materialization keeps a threshold between two snapshots to be considered similar.

The authors of [52] call their model *snapshot plus log* which is a similarity-based model since they cluster snapshots according to the distribution of historical queries. They assert that the other two materialization strategies may result in a storage or processing power degradation. TGraph [38], ChronoGraph [8] and Immortalgraph [36] are examples of time-based models while DeltaGraph [30] considers the number of events as the main criteria to materialize a snapshot.

2.1.4 Distributed graph over servers

The models described above are organized by time: the dynamic behavior is indexed by the time it takes place. The models that belong to the category of *distributed graph over servers* are called entity-centric as their formulation focuses on nodes and their history during the timespan of the dynamic graph [33]. Nodes are organized in subsets which are handled by a set of servers. Each server maintains the history of a subset of nodes [56]. For instance, HiNode [33] saves the history of each node using Interval Trees and B-Trees. The authors of [34] introduce materialization techniques and parallel complex graph data processing to reduce the memory usage while the model in [55] involves delta log files in its storage in order to reduce the update time.

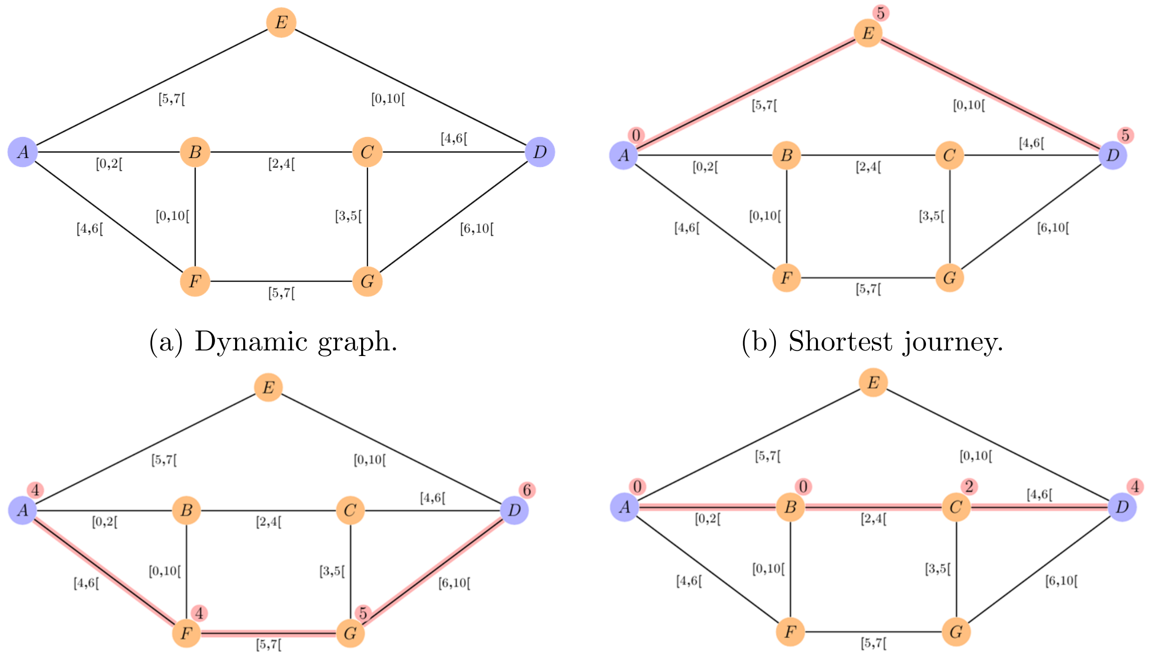
2.2 Journey computation

Paths are to static graphs as journeys are to dynamic graphs. Journeys incorporate the time domain to travel between a source node and a destination node within the network. In static graphs, the shortest path between two nodes is the one that has the minimum cost in terms of number of hops, delay, etc. For dynamic graphs, a journey is undertaken at a given time t. There are three commonly studied notions of least cost [53].

- **Shortest journey:** It ignores the time domain and among all possible journeys, it has the least number of hops between the source node and the destination node, starting from the source node at time no earlier than t.

- **Fastest journey:** It has the minimum difference between arrival time and departure time, starting from the source node no earlier than time t.

- **Foremost journey:** It has the earliest arrival time at the destination, starting from the source node at time no earlier than t.



(a) Dynamic graph. (b) Shortest journey.

(c) Fastest journey. (d) Foremost journey.

Figure 1: Least cost journeys from A to D at time no earlier than 0. The number above any node v on the journey is the cost of the specific least cost journey from A to the node D.

Figure 1 shows a dynamic graph and illustrates the three types of journeys described above. For example, for the dynamic graph shown in Figure 1a, the Figures 1b, 1c, 1d, show the shortest, fastest, and foremost journeys respectively.

We describe some algorithms to compute journeys. The first one and the most expensive one in terms of running time is Matrix Multiplication Algorithm [58] [28]. It uses adjacency matrices for each time step and multiplies them to calculate shortest, fastest and foremost journeys. The authors in [58] show that their approach takes $\mathcal{O}(|V|^4)$ time [2] per time step to compute shortest journeys. The second one is a family

[2]Using faster matrix multiplication algorithms [6], it may be possible to improve the complexity of their approach.

of algorithms that are extensions or modifications of original Dijkstra's algorithm [16]. Finally, the last algorithm, proposed in the context of finding journeys for passengers in scheduled train or subway transportation networks, is Connection Scan Algorithm (CSA) [15]. We describe below further details about Dijkstra's algorithm extensions.

2.2.1 Dijkstra's algorithms extensions

Algorithm 1: Ferreira's Algorithm [21]

```
   Input: $\mathcal{G}$, $source$
1  Make all $d(v) \leftarrow \infty$ but for $d(source) \leftarrow 0$
2  Initialize $min - heap\ Q$
3  Put $(source, key(source) \leftarrow 1)$ as $Q$'s root
4  while $key(root(Q)) \neq \infty$ do
5      $x \leftarrow Q.removeMin()$
6      for each open neighbor $v$ in adjacency list of $x$ do
7          $f_x(v) \leftarrow$ min time $(x, v)$ is available after $key(x)$
8          Insert $v$ in $Q$ if it was not there already.
9          if $f_x(v) < d(v)$ then
10             $d(v) \leftarrow f_x(v)$
11             $key(v) \leftarrow f_x(v) + 1$
12             $parent(v) \leftarrow x$
13             $Q.update(v)$
14     Close $x$ and insert it in the shortest paths tree
```

The first author who introduced an algorithm to compute foremost journeys in Evolving Graphs is Ferreira [21]. This is also our main reference to compare the performance of our proposed algorithms, so we give the pseudocode in Algorithm 1 and describe it in detail here. This algorithm has an evolving graph $\mathcal{G}$ and *source* node as inputs. The following chapter defines $\mathcal{G}$ in detail, but as we can see the algorithm uses a priority queue Q. Every node v in $\mathcal{G}$ has a $d(v)$ attribute which contains the estimate of the earliest time v can be reached. This value is initialized to zero for the source node and infinity for all other nodes. A node is *closed* if the estimate is the final correct value, and it is *open* otherwise. As in Dijkstra's algorithm, we remove the node x with least d value from the priority queue, and relax all its neighbors, after which x can be closed. It is the relaxation step that differs from Dijkstra. For each open neighbor v of x, we need to compute $f_x(v)$, the next time there is a connection between x and v. This calculation involves looking into the schedule of the edge (x, v) to know when v can be reached, and can be performed in $\mathcal{O}(\log I)$ time using binary search, where I is the maximum number of intervals in which an edge is active. Line 11 of Algorithm 1 sets $d(v)$ to be $f_x(v) + 1$. This is because the author considers a delay of one unit of time to forward a packet to its neighbor. One potential bottleneck is $Q.\mathit{update}(v)$, if the value of v changes its priority has to be updated, which takes $\mathcal{O}(\log |V|)$ time.

As proved in [21] and outlined above, for each edge adjacent to a closed vertex, Algorithm 1 performs $\mathcal{O}(\log I + \log |V|)$, where I is the max number of time intervals in which an edge is present. $|V|$ is the number of nodes. Thus the algorithm takes time $\mathcal{O}(|E|(\log I + \log |V|))$.

Algorithms for shortest and fastest journey computation were introduced in [53]. These algorithms are also modifications or extensions of Dijkstra's algorithm [16]. Some speed-up techniques have been introduced to find shortest journeys faster. For example, the author in [25] extends bidirectional Dijkstra's algorithm and Landmark-based A* to compute shortest journeys in TEG's. Another speed-up technique incorporated in [25] is Contraction Hierarchies (CH). Basically, CH allows to add shortcuts into the original TEG so the algorithm is able to bypass nodes that are not important during shortest journey calculation. The downside of CH incorporation is that it adds a huge pre-processing overhead.

2.3 Connection scan algorithm

Connection Scan Algorithm (CSA) [15] was designed as a route planning algorithm for public transit systems, involving a combination of trains, buses, trams, for example. Users of such algorithms may not only want to find the earliest arrival time to reach a certain destination, but may also have other criteria, such as minimizing the number of hops. Train and bus stops are connected by *footpaths* which incur a certain delay. CSA outputs a sequence of footpaths-vehicles that a traveler has to follow in order to reach a desired destination from a given source position, that optimizes the given criteria.

In CSA, the public transportation system is modeled as a timetable which is a quadruple of stops, connections, trips, and footpaths $(\mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{F})$, respectively. The

Algorithm 2: Connection Scan Algorithm [15]

```
Input  : $source$, $\tau$, $target$, $S$, $C$, $T$, $F$
Output: The minimum arrival time over all journeys that depart after $\tau$ at
        $source$ and arrive at $target$
1  for all stops x do
2      $S[x] \leftarrow \infty$
3  for all trips x do
4      $T[x] \leftarrow false$
5  for all footpaths f from source do
6      $S[f_{arr-stop}] \leftarrow \tau + f_{dur}$
7  Find first connection $c^0$ departing not before $\tau$ using binary search in $C$
8  for all connections c increasing by $c_{dep-time}$ starting at $c^0$ do
9      if $S[target] \leq c_{dep-time}$ then
10         break
11     if $T[c_{trip}]$ is true or $S[c_{dep-stop}] \leq c_{dep-time}$ then
12         $T[c_{trip}] \leftarrow true$
13         if $c_{arr-time} < S[c_{arr-stop}]$ then
14             for all footpaths f from $c_{arr-stop}$ do
15                 $S[f_{arr-stop}] \leftarrow \min\{S[f_{arr-stop}], c_{arr-time} + f_{dur}\}$
16 return $S[target]$
```

footpath graph $\mathcal{F}$ has the stops listed in $\mathcal{S}$ as nodes and the edges are the footpaths that the traveler can walk. The weight for each edge f_{dur} is the time that it takes to walk from one stop to another. Note that each stop has a self-loop whose (non-zero) weight is the time taken to transfer at the stop. $\mathcal{T}$ is the set of trips which are scheduled vehicles. Connections in $\mathcal{C}$ describe how vehicles drive; they are modeled as departure stop, arrival stop, departure time and arrival time $(c_{dep}, c_{arr}, c_{dep-time}, c_{arr-time})$, respectively. A *trip* is described as a sequence of connections driven by the same physical vehicle.

Algorithm 2 shows the earliest arrival Connection Scan Algorithm variant [15]. The inputs are the quadruple timetable, a *source* stop, a *target* stop and the departure time τ. This algorithm assumes that connections are ordered by departure time and the footpath graph is stored as adjacency lists. If we contrast this algorithm with Dijkstra's algorithms extensions then we see that there is no priority queue to keep a tentative arrival time. Instead, it keeps two arrays; $\mathcal{S}$ array maintains the earliest tentative arrival time for each stop and $\mathcal{T}$ array keeps track which trips have been boarded. One connection c is reachable if the traveler is at the departure stop $c_{dep-stop}$ before departure time $c_{dep-time}$ or the trip flag $\mathcal{T}[c_{trip}]$ is set. This flag indicates that a connection that belongs to the same trip was boarded before and the traveler can remain seated. Once the algorithm finds a reachable c connection; the algorithm modifies the tentative arrival time at the stops reachable by foot from the arrival stop $c_{arr-stop}$. This Connection Scan Algorithm variant already includes three optimizations. One called stopping criterion in line 9 which ends the execution after finding a connection such that its departure time is greater than the tentative arrival time at *target*. The second one known as starting criterion in line 7 that finds a connection c^0 whose departure is not before τ using binary search. The last optimization limits the iteration of outgoing footpaths when the time at $\mathcal{S}[c_{arr-stop}]$ is greater than $c_{arr-time}$ in line 13.

CSA journey extraction

Algorithm 2 only outputs the earliest arrival time at the *target* stop but there is no information about the journey. A journey in [15] is an alternating sequence of legs and footpaths that indicates the passenger how to travel within the network; it must start and end with a footpath. In this context, a leg is a pair of connections that belong to the same trip; the first one is the enter connection l_{enter} and the latter is the exit connection l_{exit}. Logically, l_{enter} must appear before l_{exit} in the timetable. There is a footpath between legs, this leg transition is called *transfer*; it must follow the transfer model that the authors in [15] propose. Basically, the transfer model tells the feasibility of boarding a departing connection c' at stop $c'_{dep-stop}$ coming from a connection c arriving at stop $c_{arr-stop}$. In other words, it is possible to board a departing connection that belongs to another trip as long as the passenger is at the departure stop before the departure time. Note that, loops in this model allow the passenger to transfer at the same stop. They call the duration f_{dur} of the loop as change time and every other footpath that is not a loop is called interstop footpath [15].

They propose two ways to extract a journey. The first one uses something that the authors in [15] call *journey pointers* which requires an extension of the current data structures. The second one extracts the journey without *journey pointers* but needs more complex tasks and data structures. Depending on the scenario, one is better than the other in terms of running time. If only one journey from a source stop towards a target stop is needed then the approach without *journey pointers* is faster. But, if many journeys from one source stop towards many target stops are needed then the *journey pointers* approach is faster and easiest to implement.

Journey pointer calculation requires a modification of $\mathcal{T}$ array and an addition of a new $\mathcal{J}$ array that holds the *journey pointers* for each stop. Instead of saving a boolean flag in $\mathcal{T}$ for each trip, it holds the ID of the first reached connection for each trip. $\mathcal{J}$ array keeps triples that consist of the exit connection, the final footpath and the enter connection. Algorithm 2 shows that at line 15 the journey pointer can be easily calculated since the exit connection and the final footpath are right there. But, there is no information about the enter connection. So, to complete the triple the algorithm queries the modified $\mathcal{T}$ array to retrieve the ID of the enter connection. Once the algorithm finishes its execution; it is enough to follow the pointers and back track the journey starting at the target stop until the source stop is found.

Journey extraction without *journey pointers* requires two adjacency matrices. One matches the trips IDs with the journey IDs. So, the algorithm is able to iterate the connections that belong to the same trip fast. The second adjacency matrix encodes the stop ID with a sorted list which contains the IDs of the arriving connections at that stop. The algorithm uses a subroutine to calculate the enter connection, the exit connection and the footpath to form the triple. As described earlier, the enter connection is hard to calculate. So, to overcome this problem this approach builds a set of candidate enter connections for each exit connection. Then, the list of connections that belong to the same trip of a candidate connection is retrieved. A valid enter connection is the x connection within that list such that the time at the departure stop is less than or equal to the x departure time $S[x_{dep-stop}] \leq x_{dep-time}$. Finally, the candidate sets are pruned and the journey is extracted.

Chapter 3

Timetable-Based journey computation

In this chapter, we describe a timetable-based approach to computing foremost journeys in FSDN, that is inspired by the Connection Scan Algorithm described in the previous chapter. First, in Section 3.1 we give some basic definitions and describe the data structures used, starting with the Evolving Graph $\mathcal{G}$ definition introduced in [21]. Then, we specify our definitions of connection, timetable, duplicated timetable and journey.

In Section 3.2 we describe a slight modification in the original Ferreira's algorithm [21] to support the query time t_q. Then Section 3.3, presents what we call Timetable-based Algorithms. So, we propose the following algorithms: Hybrid Timetable-Evolving Graph, Pure-Timetable, Timetable with Auxiliary Graph. The name of each algorithm has to do with the approach to handle non-relaxed connections. Finally, in Section 3.4 a simple journey extraction algorithm is presented.

3.1 Preliminaries

3.1.1 Evolving graph

The authors in [53] define $\mathcal{G}$ as a system $\mathcal{G} = (G, \mathcal{S}_G)$ where $G = (V, E)$ is a graph with a set of nodes V and a set of edges E and $\mathcal{S}_G = G_0, G_1, \ldots, G_s$ is an ordered sequence of subgraphs G_i of G. Each G_i has a set of nodes V_i and a set of edges E_i running in the interval $[t_i, t_i + 1[$. Two nodes are adjacent in $\mathcal{G}$ if they are adjacent in any G_i.

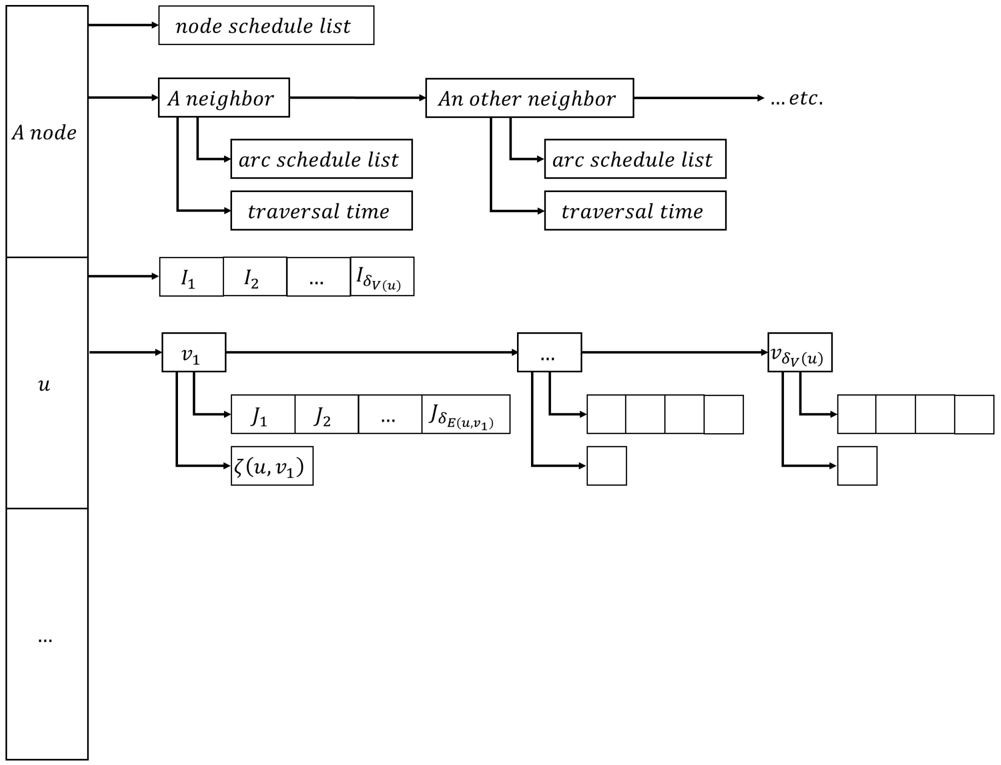


Figure 2: Fixed-schedule Evolving Graph data structure [53].

Alternatively, $\mathcal{G}$ can be defined as $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ as Equations 1 and 2 show. So, each edge e in $E_\mathcal{G}$ has a set of ordered intervals $P_E(e)$ defining the edge schedule which contains the time steps when e is present. In addition, each node v in $V_\mathcal{G}$ has its own ordered schedule $P_V(v)$ indicating the time steps when v is present in $\mathcal{G}$ [53]. Furthermore, the authors provide some notions of node, edge and evolving graph activity: the activity of a vertex v as $\delta_V(v) = |P_V(v)|$, the activity of an edge e as $\delta_E(e) = |P_E(e)|$, the node activity of an evolving graph as $\delta_V = \max\{\delta_V(v), v \in V_\mathcal{G}\}$, the edge activity of an evolving graph as $\delta_E = \max\{\delta_E(e), e \in E_\mathcal{G}\}$ and finally the activity of an evolving graph as $\delta = \max(\delta_V, \delta_E)$.

$$E_\mathcal{G} = \bigcup_{i=0}^{j} E_i : j \leq s \therefore M = |E_\mathcal{G}| \leq |E| = m \quad (1)$$

$$V_\mathcal{G} = \bigcup_{i=0}^{j} V_i : j \leq s \therefore N = |V_\mathcal{G}| \leq |V| = n \quad (2)$$

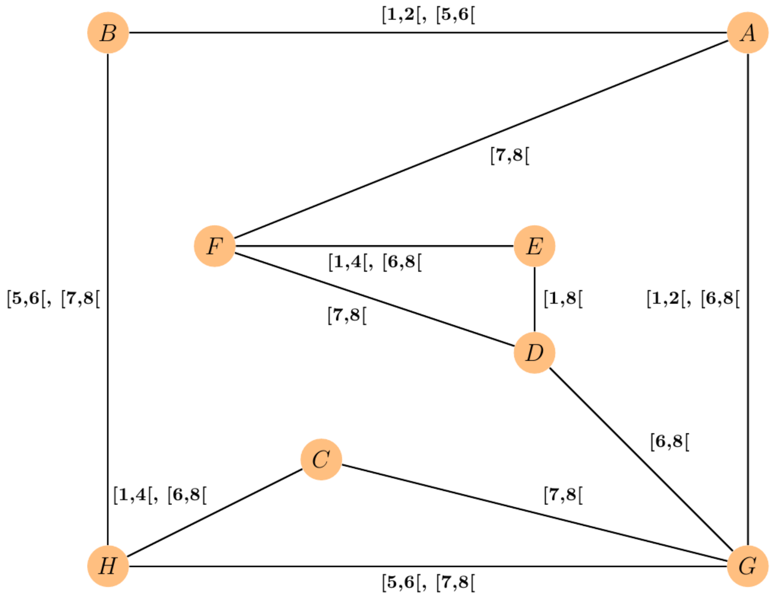


Figure 3: Evolving Graph example.

Similar to weighted static graphs, $\mathcal{G}$ can be weighted too. So, there is a cost to traverse an edge. For instance, the cost can be represented as physical distance, power, hops, delay, etc. If the weight or cost of edge traversal belongs to the time domain then $\mathcal{G}$ defines a function $\zeta(u, v)$ which models the time that it takes to traverse from node u to node v.

Ferrerira's algorithm is first described in [21] and the data structure used $\mathcal{G}$ is described in detail in [53]. As shown in Figure 2, $\mathcal{G}$ is coded as a linked adjacency list. To indicate when an edge is present and how long takes to traverse it, each neighbor v has its own sorted edge schedule $P_E(e)$ given as time steps intervals and $\zeta(u, v)$ attached to it. Also, each node u in $\mathcal{G}$ has a sorted list of intervals $P_V(v)$ to point the time steps when it appears. In other words, $\mathcal{G}$'s model considers that there may be time steps when a node disappears [21]. The authors prove that the size of $\mathcal{G}$ in the worst case is $\mathcal{O}(M + M * \delta_E + N * \delta_V) = \mathcal{O}(M * \delta)$.

For the scope of this work we make three considerations regarding $\mathcal{G}$'s coding and interpretation. First, we consider that every G_i in $\mathcal{S}_G$ has the same set of nodes $V_0 = V_1 = \cdots = V_s$. Therefore the node schedule $P_V(v)$ is not needed since every node is always present and never disappears. Our implementation of $\mathcal{G}$ therefore requires only the edge schedule $P_E(e)$ as time step intervals for every neighbor. For instance, Figure 3 shows an example of an Evolving Graph. The label in each edge is the schedule with the intervals that indicates when the edge is present or alive. Also, Figure 4 shows how Figure 3 would be coded with our considerations for the Evolving Graph data structure.

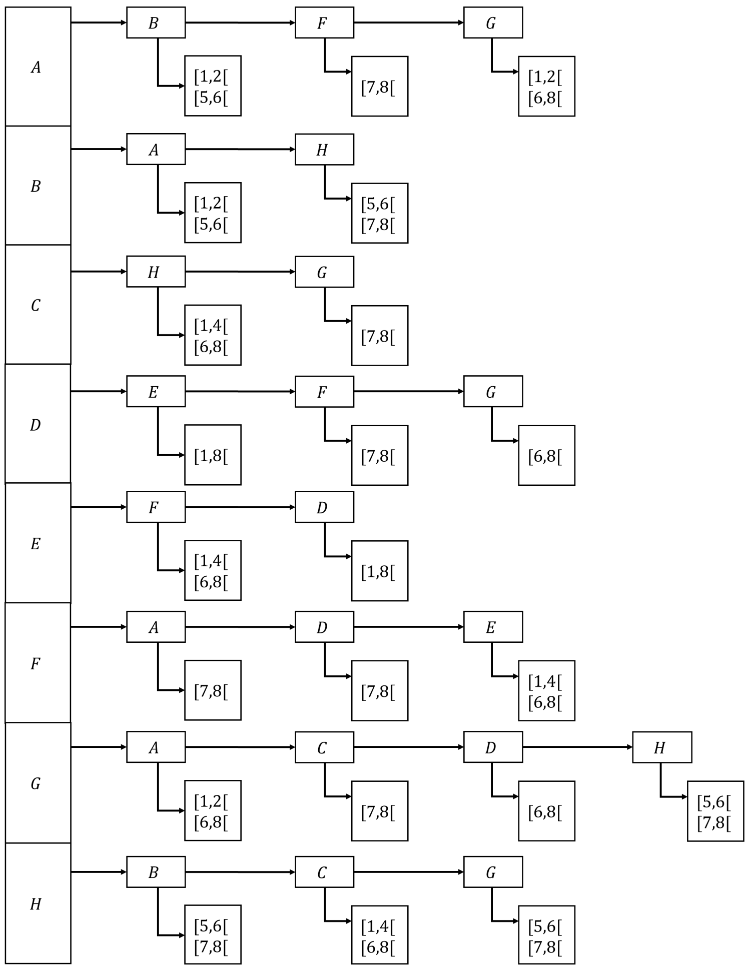


Figure 4: Evolving Graph data structure example.

Second, we assume instantaneous edge traversal. In other words, there is negligible delay or cost for ζ. The assumption of negligible delay is justified for the FSDN that we consider such as bus networks and satellite networks. The two technologies used in vehicular networks are IEEE 802.11p and LTE. The authors of [37] define the mean delay to be the ratio between the summation of all end-to-end delays and the number of received packets. They show that the end-to-end delay with IEEE 802.11p is less than 100ms for 90% of the beacons with 25 vehicles running on the road network while for 50 vehicles; 40% of the beacons have a delay less than 100ms. LTE technology is even better, achieving an end-to-end delay less than 60ms depending on the beacon transmission frequency. Considering that satellite networks have optic links capable of transferring 2,50Gbps over a distance of 2500km [10], and cover only about 3 km in one second, it is reasonable to consider that the duration of connections is longer than end-to-end transmission takes place. A similar calculation holds for buses.

3.1.2 Connection

A connection $c = (c_{source}, c_{target}, c_{start}, c_{end})$ corresponds to a temporal edge in an evolving graph $\mathcal{G}$. Basically, a connection expresses the existence of a link between nodes c_{source} and c_{target} in the time interval $[c_{start}, c_{end}[$. There are two fundamental logical restrictions for this definition regarding the endpoints and time. The first one is $c_{source} \neq c_{target}$ and the second one is $c_{end} > c_{start}$. We say a connection c is *active* at time t if $c_{start} \leq t < c_{end}$.

3.1.3 Timetable

A timetable $\mathcal{T}$ is another way to represent an evolving graph: it is a pair $(V, \mathcal{C})$ of a set of nodes V and an ordered sequence of lists of connections $\mathcal{C} = \{\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_s\}$, where $\mathcal{C}_i$ is the set of the connections that start at time t_i, as Figure 5 shows. We denote the first and last time step in the timetable as $\mathcal{T}_{start} = t_0$ and $\mathcal{T}_{end} = t_s$, respectively.

Figure 6 illustrates an example, and shows the timetable representation of the graph in Figure 3.

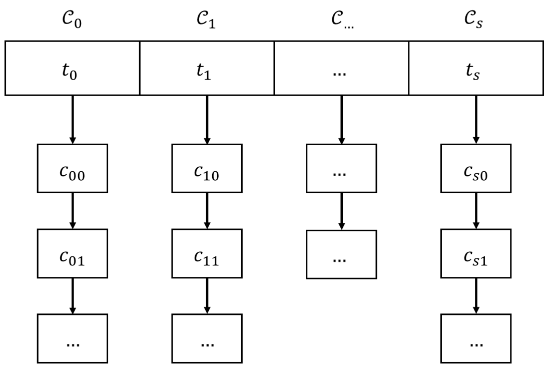


Figure 5: $\mathcal{C}$ data structure.

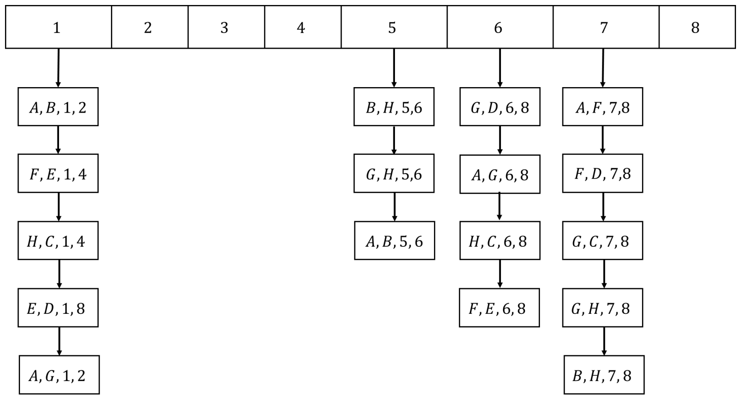


Figure 6: $\mathcal{C}$ data structure example.

3.1.4 Duplicated timetable

We define an operation using $\mathcal{T}$ which returns another timetable $\mathcal{T}_d$ that we call duplicated timetable. This operation $\mathcal{T}_d = duplicate(\mathcal{T})$ is specified in Algorithm 3. The main particularity of $\mathcal{T}_d$ is that each $\mathcal{C}_t$ holds the connections that are *active* at time t. It is clear that the size of the duplicated timetable could be much bigger than the original timetable.

Algorithm 3: Duplication

```
Input  : $\mathcal{T}$
Output: $\mathcal{T}_d$
1 Initialize $\mathcal{T}_d \leftarrow \emptyset$
2 for each connection c increasing by $c_{start}$ in $\mathcal{T}$ starting at $\mathcal{T}_{start}$ do
3     $i \leftarrow c_{start} + 1$
4     while $i < c_{end}$ do
5         $\mathcal{C}_i.insert(c)$
6         $i \leftarrow i + 1$
```

3.1.5 Journey

A journey $\mathcal{J}(source, target)$ in an evolving graph $\mathcal{G}$ is a temporal path from *source* to *target*, that is, it is a sequence of pairs of nodes v_i and time steps t_i:

$$\mathcal{J}(source, target) = \{(v_0, t_0), \ldots, (v_j, t_j)\},\ v_i \in V \tag{3}$$

$$\text{where } v_0 = source\ \wedge\ v_j = target\ \wedge\ t_i \leq t_{i+1}\ \forall\, 0 \leq i < j$$

In terms of the timetable representation, note that in order for a packet to travel from node v_i to v_{i+1} at time t_{i+1} there must be an active connection $c \in \mathcal{C}$ at time t_{i+1} such that:

$$c_{source} = v_i \ \wedge \ c_{target} = v_{i+1} \ \wedge \ c_{start} \leq t_{i+1} < c_{end} \tag{4}$$

We define $dep(\mathcal{J}) = t_0$ and $arr(\mathcal{J}) = t_j$. Informally, we call these the departure time, and arrival time of the journey $\mathcal{J}$.

We say a journey $\mathcal{J}$ is *foremost* if $arr(\mathcal{J})$ is as small as possible. As shown in [53], there exists a foremost journey such that every prefix of the journey is also a foremost journey; such a journey is called a *ubiquitous* foremost journey.

We denote by $\delta(s, v, t_q)$, the earliest time that v can be reached departing from s at time t_q. Then $\delta(s, v, t_q) = t$ if and only if there is a foremost journey $\mathcal{J} = \mathcal{J}(s, v)$ with $dep(\mathcal{J}) = t_q$ and $arr(\mathcal{J}) = t$.

Algorithm 4: Connection Relaxation Algorithm

```
Input: $c$
1 if $d(c_{source}) > c_{start}$ then
2     $d(c_{target}) \leftarrow d(c_{source})$
3 else
4     $d(c_{target}) \leftarrow c_{start}$
5 $parent(c_{target}) \leftarrow c_{source}$
```

We now describe a fundamental primitive used in our algorithms, called *connection relaxation*. Recall that the process of edge relaxation in shortest path algorithms such as Dijkstra and Bellman-Ford for static graphs involves checking if going through an edge (u, v) can lower the cost of reaching the vertex v, and accordingly updating the estimated cost of reaching v. In the evolving graphs that we are investigating, edges have zero weight, but are transient. Therefore, we examine a connection, and if source of the connection has already been reached while the target has not, then we can reset the arrival time at the target appropriately as described in Algorithm 4. The condition $d(c_{source}) > c_{start}$ noted in line 1, tests if c_{source} has been reached *after* the connection c has already started. In this case, $d(c_{target})$ is set to $d(c_{source})$. Otherwise, c_{source} holds the packet until c starts at which time it can pass it to c_{target}. Thus $d(c_{target})$ is set equal to c_{start}. Afterwards, $parent(c_{target})$ is set to indicate c_{source} as the predecessor of c_{target}.

Notice that when Algorithm 4 relaxes a connection, it sets a parent reference from the target of the connection to the source of the connection. While in our description of algorithms, we generally are concerned with the *arrival time* of the foremost journey, it should be clear that by following the references that each node holds in $parent(v)$, we can extract the corresponding journey from *source* to *target*.

3.2 Ferreira's algorithm with t_q support

Algorithm 5: Ferreira's Algorithm with t_q support

```
   Input: $\mathcal{G}$, $source$, $target$, $t_q$
1  Make all $d(v) \leftarrow \infty$ except $d(source) \leftarrow t_q$
2  Initialize $min - heap\ Q$
3  Put $(source, key(source) \leftarrow t_q)$ as $Q$'s root
4  while $key(root(Q)) \neq \infty$ do
5      $x \leftarrow Q.removeMin()$
6      Stop if $x = target$
7      for each open neighbor $v$ of $x$ do
8          $f_x(v) \leftarrow$ min time $(x, v)$ is available after $key(x)$
9          Insert $v$ in $Q$ if it was not there already.
10         if $f_x(v) < d(v)$ then
11             $key(v) \leftarrow d(v) \leftarrow f_x(v)$
12             $parent(v) \leftarrow x$
13             $Q.update(v)$
```

Algorithm 5 shows Ferreira's algorithm [21] that supports t_q as input. The difference between Algorithm 1 and Algorithm 5 is that $d(source)$ is set to t_q instead of zero. This is because the earliest time that *source* can be reached is not zero anymore but t_q. Since, we consider no delay for edge traversing the variables $key(source)$ and i are set to t_q instead of one. Also, the no delay assumption is showed in line 13 once a node is relaxed it sets the value of $d(v)$ and $key(v)$ equal to $f_x(v)$.

Another added feature is the stop statement in line 6. If the node x that is inserted in the shortest path tree is equal to *target* then it means that the algorithm found the foremost journey $\mathcal{J}(source, target)$. So, it is not necessary to continue running the algorithm.

The complexity of this algorithm which is $\mathcal{O}(|E|(\log I + \log |V|))$ remains the same as explained in the previous chapter since its complexity does not depend on t_q but in the total number of processed nodes and table look-up for valid schedule times. Also, its correctness proof is the same since instead of starting at time zero, this version starts at time t_q which is the earliest time during Algorithm 5 running.

3.3 Timetable-Based Algorithms

In this section, we describe a new timetable-based approach to find a foremost journey in an evolving graph. We then describe three new algorithms using this new approach which we call: Pure-Timetable, Hybrid Timetable-Evolving Graph, Timetable with Auxiliary Graph and prove their correctness and tight bounds on their space and time complexities.

The key advantage of our timetable-based approach over Ferreira's algorithm is that we use faster array-based operations on the timetable of connections, rather than priority-queue operations. As in Ferreira's algorithm, we maintain an array $d[]$ containing estimates of the earliest arrival time at every node in the network. The main idea is to process connections in sorted order of the start time of connections. When processing a connection, if one of its endpoints has already been reached, we can relax the connection to obtain the earliest arrival time at the other end of the connection. We will show that as soon as node is reached via a connection for the first time, we will have the earliest arrival time at the node.

The main difficulty that arises is that connections are not eligible to be relaxed when they start, but they could be useful later on. For example, suppose $c_1 = (a, b, t_1, t_1 + 2)$ and $c_2 = (x, a, t_1 + 1, t_1 + 2)$. Then c_1 appears before c_2 in the timetable. It is possible that neither a nor b is reachable at time t_1. However, x may be reachable at time $t_1 + 1$ when connection c_2 is processed. This implies that a is also reachable at time $t_1 + 1$, and via connection c_1 which is still active at time $t_1 + 1$, the node b is also reachable at $t_1 + 1$. Thus we cannot simply discard connection c_1 at time t_1 if it is not useful then; it may be useful later. It is not hard to see that even with connections starting at the same time, their order in the timetable can make a difference to whether or not they can be immediately useful.

A related difficulty arises from the fact that to find a journey starting at time t, we cannot simply start with connections starting at time t, as there may be pre-existing connections that are still alive at time t that can be used.

Our three algorithms utilize different approaches to solve the two problems described above.

Algorithm 6: Pure-Timetable

```
Input: $\mathcal{T}, source, target, t_q$
1  $Initialize(d, location)$
2  $d(source) \leftarrow t_q$
3  $Initialize(\mathcal{T}_{aux})$
4  $RelaxationFlag \leftarrow false$
5  for $t \leftarrow location$ to $s$ do
6      for each connection $c$ in $\mathcal{C}_t$ do
7          Orient $c$ so that $d(c_{source}) \leq d(c_{target})$
8          if $d(c_{source}) < c_{end}$ and $d(c_{target}) = \infty$ then
9              Relax($c$)
10             $RelaxationFlag \leftarrow true$
11             Stop if $c_{target} = target$
12         else if $d(c_{source}) = \infty$ and $d(c_{target}) = \infty$ and $c_{end} > t_q$ then
13             $\mathcal{T}_{aux}.insert(c)$
14     while $RelaxationFlag$ is set do
15         $RelaxationFlag \leftarrow false$
16         for each connection $r$ in $\mathcal{T}_{aux}$ do
17             Orient $r$ so that $d(r_{source}) \leq d(r_{target})$
18             if $d(r_{source}) < r_{end}$ and $d(r_{target}) = \infty$ then
19                 Relax($r$)
20                 Stop if $r_{target} = target$
21                 $\mathcal{T}_{aux}.remove(r)$
22                 $RelaxationFlag \leftarrow true$
23             else if $d(r_{target}) \neq \infty$ or $r_{end} \leq t$ then
24                 $\mathcal{T}_{aux}.remove(r)$
```

3.3.1 Pure-Timetable

The first timetable based algorithm that we describe is Algorithm 6 which we call Pure-Timetable. This approach uses an auxiliary timetable $\mathcal{T}_{aux}$ to store connections that were ineligible to be relaxed when first processed, but might still be eligible to be relaxed later.

The first step is $Initialize(d, location)$. The initialization of d involves setting $d(v)$ to infinity for every node v element of V. The *location* variable is set to the smallest t such that there is a connection starting at t that is still active at time t_q. To do this, we use a the duplicated timetable generated with Algorithm 3, in which connections are duplicated in every time step that they are active; this table is used only for this purpose. Next we set $\mathcal{T}_{aux}$, the auxiliary timetable to empty, and set to false the boolean flag $RelaxationFlag$, which indicates if there was a relaxation of any edge while processing connections that are active at time t. This flag will be used to determine if $\mathcal{T}_{aux}$ is to be scanned again.

In the event of finding a connection c that can not be relaxed, then there are some conditions to meet in order to save a connection in $\mathcal{T}_{aux}$. These conditions are $d(c_{source}) = \infty$, $d(c_{target}) = \infty$ and $c_{end} > t_q$. In other words, one connection is saved only if both ends; c_{source} and c_{target} are not being reached yet and the connection is still *active* after t_q. One thing to recall and notice is that connections in $\mathcal{T}$ are ordered by c_{start} and connections are added to $\mathcal{T}_{aux}$ as $\mathcal{T}$ is scanned. Then, connections in $\mathcal{T}_{aux}$ keep the same order that they have in $\mathcal{T}$.

The proof of correctness of our Pure-Timetable algorithm involves the following lemmas that we prove as follows:

Lemma 3.3.1 *For every t, at the end of the iteration of the outer loop corresponding to time t, the table $\mathcal{T}_{aux}$ contains all connections r satisfying:*

$$d(r_{source}) = d(r_{target}) = \infty \wedge r_{start} \leq t \wedge r_{end} \geq \max(t, t_q)$$

Proof. The conditions to meet in order to save a connection r in $\mathcal{T}_{aux}$ are: $d(r_{source}) = \infty$, $d(r_{target}) = \infty$ and, since $r \in C_t$ we have $r_{end} > t \geq t_q$. In other words, a connection r is saved only if both ends; r_{source} and r_{target} have not been reached yet and the connection is still active. Since connections that start after time t have not yet been processed, it is clear that $r_{start} \leq t$. On the other hand, while scanning $\mathcal{T}_{aux}$, connections that are no longer active are removed. Therefore, at the end of iteration of the outer loop for time t, the connections that $\mathcal{T}_{aux}$ holds are the set of all still-active connections r for which neither endpoint has yet been reached and that are still alive at time t. □

Lemma 3.3.2 *Once $d(v)$ is set to an integer value, it will never change again.*

Proof. Note that the d-value of any node is set either in line 2, or inside a call to a Relax procedure; such a procedure is only called in lines 9 and 19 of the algorithm. In both cases, the connection c is relaxed only if its target v satisfies $d(v) = \infty$. Once $d(v)$ is set to an integer value, no connection with v as target can be relaxed again. □

Lemma 3.3.3 *During the iteration of the outer loop corresponding to time $t \geq t_q$, if we call Relax(c), then $d(c_{target})$ is set to t.*

Proof. First note that when the Relax procedure is called for a connection c in $\mathcal{T}$ (in line 9), we can assume inductively that $d(c_{source}) \leq t = c_{start}$, therefore when c is relaxed, $d(c_{target})$ is set to t. Now consider the case when the Relax procedure is called for a connection r in $\mathcal{T}_{aux}$ in line 19. By Lemma 3.3.1, we know that when we started processing connections in C_t, for all connections in $\mathcal{T}_{aux}$, we must have had $d(r_{source}) = d(r_{target}) = \infty$. Since the connection r is now eligible to be relaxed, it must be that some other connection c in $\mathcal{T}$ with $c_{start} = t$ and $c_{target} = r_{source}$ was relaxed. As argued earlier, in this case, $d(c_{target}) = d(r_{source})$ must have been set to t. Therefore, when r is relaxed, $d(r_{target})$ will be set to t. □

Lemma 3.3.4 *During the iteration of the outer loop corresponding to time $t < t_q$, any call to Relax(c) leads to $d(c_{target}) = t_q$.*

Proof. Consider the first iteration of the outer loop corresponding to a time $t < t_q$. This means there is a connection c that started before t_q but was still alive at time t. Since the only node for which the d value is not ∞ is *source*, if neither endpoint of c is *source*, then the condition for relaxation is not met, and c is added to $\mathcal{T}_{aux}$. Therefore, the first connection c which will be relaxed must have *source* as an endpoint, and since $d(c_{source}) = t_q > t = d(c_{start})$, according to the logic of the relaxation algorithm,

$d(c_{target})$ is set to $d(c_{source}) = t_q$. A similar logic applies to subsequent connections that are relaxed in C_t for $t < t_q$ (even if neither endpoint is *source*).

Now consider the case when the Relax procedure is called for a connection r in $\mathcal{T}_{aux}$ in line 19. By Lemma 3.3.1, we know that when we started processing connections in C_t, for all connections in $\mathcal{T}_{aux}$, we must have had $d(r_{source}) = d(r_{target}) = \infty$. Since the connection r is now eligible to be relaxed, it must be that some other connection c in $\mathcal{T}$ with $c_{start} = t$ and $c_{target} = r_{source}$ was relaxed. As argued earlier, in this case, $d(c_{target}) \leftarrow d(r_{source})$ must have been set to $t_q > t$. Therefore, when r is relaxed, $d(r_{target})$ will be set to t_q.

$\square$

We now show that Algorithm 6 correctly sets $d(v) = \delta(s, v, t_q)$, recall that $\delta(s, v, t_q)$ is the earliest time that v can be reached departing from s at time t_q, for every vertex v in the evolving graph.

To do this, we need some new notation. We define $T(s, t_q, t)$ to be the set of all nodes whose earliest arrival time is t, starting from s at time t_q. That is,

$$T(s, t_q, t) = \{u \mid t = \delta(s, u, t_q)\} \tag{5}$$

Lemma 3.3.5 *For every t with $t_q \le t \le t_s$, at the end of the iteration corresponding to time t in Algorithm 6, for every $u \in T(s, t_q, t)$, we have $d(u) = \delta(s, u, t_q) = t$.*

Proof. We give a proof by contradiction. Let t be the minimum start time such that there exists a $v \in T(s, t_q, t)$ such that after the iteration of the outer loop corresponding to time t, we have $d(v) \neq t$. We claim that then $d(v) = \infty$. First note that the value of $d(v)$ can only be set according to connections being relaxed along a journey from s to v, and therefore, it is impossible that $d(v) < t$. By Lemma 3.3.2, it cannot be that $d(v)$ was set to an integer value $> t$. We conclude that $d(v) = \infty$.

If there are many such vertices, we pick v so that it has a foremost journey $\mathcal{J}$ starting from s at time t_q such that $\mathcal{J}$ has the *minimum number of hops* among all such vertices. Note that there may be many such min-hop foremost journeys, we pick an arbitrary one of them. Now consider the previous vertex u on this journey. It follows from the definition of v as a node with a *min-hop* foremost journey with arrival time t that $d(u)$ must be set correctly to $\delta(s, u, t_q)$. Note that there must be such a previous vertex, as $v \neq source$, since $d(source)$ is correctly set to t_q by the algorithm. We consider the following two cases for u.

1. $\delta(s, u, t_q) = t$: Then there must be a connection c from u to v that is active at time t, in other words, $c_{end} > t$.

   If $c_{start} = t$, then c would be processed while scanning connections in $\mathcal{T}$. If $d(u) = t$ at the time c is processed, then the connection c would be relaxed and $d(v)$ would be set to t, a contradiction. If $d(u) \neq t$ at the time c is processed, then it must be that $d(u) = \infty$. Therefore c would be added to $\mathcal{T}_{aux}$. By assumption $d(u)$ is set to t during some iteration of the while-loop while $\mathcal{T}_{aux}$ is processed. In the next iteration of the while loop, the connection c would be relaxed and $d(v)$ would be set to t, a contradiction.

   If instead $c_{start} < t$ then since $d(u) = d(v) = \infty$ and $c_{end} \geq t$, when the outer loop iteration corresponding to time t starts, c would be part of $\mathcal{T}_{aux}$ by Lemma 3.3.1. As in the previous case, by assumption $d(u)$ is set to t during some iteration of the while loop in which $\mathcal{T}_{aux}$ is processed. In the next iteration of the loop, the connection c is relaxed and $d(v)$ would be set to t.

2. $\delta(s, u, t_q) = t' < t$: By the minimality of t, we infer that the value of $d(u)$ is set to t'. Also, there is an active connection c from u to v at time t. This connection must start at time t, otherwise v would be reachable before time t. Therefore c would be found and relaxed during $\mathcal{C}_t$ scanning since $d(u)$ is already set to t' which is less than t. According to the relaxation logic shown in Algorithm 4 $d(v)$ is set to t, a contradiction.

□

Observing that the size of $\mathcal{T}_{aux}$ never exceeds that of $\mathcal{T}$, we are ready to prove the main theorem of this section:

Theorem 3.3.1 *Algorithm 6 finds the foremost journey from any source node to any target node in an evolving graph $G = (V, E)$ with timetable $\mathcal{T}$ in time $\mathcal{O}(|V||E| + |\mathcal{T}|)$, using space $\mathcal{O}(|\mathcal{T}|)$.*

Proof. The correctness of Algorithm 6 follows from Lemmas 3.3.1 to 3.3.5. We now prove an upper bound on the time complexity. First we note that $\mathcal{T}_{aux}$ can be implemented as a doubly linked list so that insertion and deletion operations can be performed in $\mathcal{O}(1)$ time. Also, we can ensure that its size is always at most $|E|$, by checking for every edge, before inserting a connection corresponding to the edge, if such a connection already exists, and if it does, simply updating with the new connection. It is clear that every instruction in the inner for loop can be implemented in $\mathcal{O}(1)$ time, for a total cost of $\mathcal{O}(|\mathcal{T}|)$.

It remains to analyze the complexity of the while loop. Notice that the while loop is executed only when an edge is relaxed, that is, when the d value of a node is changed. By Lemma 3.3.2, once the $d(v)$ value is set it will never be set again. So the condition for the while loop to execute can be satisfied at most $|V|-1$ times. Together with the fact that the size of $\mathcal{T}_{aux}$ is at most $|E|$, and each instruction in the for-loop scanning $\mathcal{T}_{aux}$ can be executed in $\mathcal{O}(1)$ time, we conclude that the total time taken to execute the while loop over *all* iterations of the outer loop is $\mathcal{O}(|V||E|) = \mathcal{O}(|V|^3)$. We have shown that the total time taken by Algorithm 6 is $\mathcal{O}(|V||E| + |\mathcal{T}|)$. □

We now give in Figure 7 an example of an evolving graph on which Algorithm 6 takes $\Omega(|V||E| + |\mathcal{T}|)$ demonstrating that the above analysis is tight. The base graph consists of a path of $n/2$ nodes connected to a clique of $n/2$ nodes (i.e. $K_{n/2}$). The schedule of edges is described as follows: the edges in the clique are always alive, while on the path, the connection between node i and node $i+1$ starts only in time step i. In this example we start at time step zero, so at the end of processing connections starting at time step 0, the size of $\mathcal{T}_{aux}$ is equal to number of edges in $K_{n/2}$, that is, there are $\Theta(n^2)$ connections in $\mathcal{T}_{aux}$. In each of the subsequent $n/2$ time steps, exactly one connection is processed and relaxed, which necessitates scanning the entire $\mathcal{T}_{aux}$. However, none of these iterations causes any connection to be removed from $\mathcal{T}_{aux}$, which therefore has size $n/2(n/2-1)$ for $n/2$ steps. The total cost paid is lower bounded by $n/2(n/2)(n/2-1) = \Theta(n^3) = \Theta(|V||E|)$.

In this context the total cost of the previous example is $\Theta(V^3)$.

3.3.2 Hybrid Timetable-Evolving Graph

In this section, we describe a hybrid algorithm that uses both a timetable and the evolving graph representation of a dynamic network. The pseudocode is given in

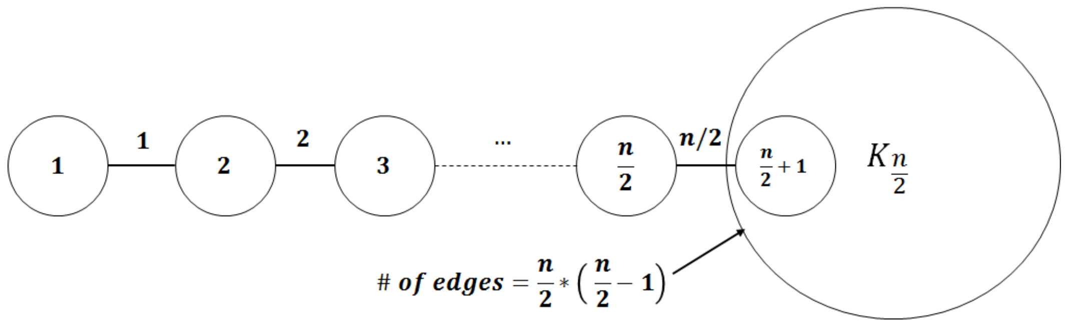


Figure 7: Complexity analysis example.

Algorithm 7. Like Algorithm 6, we process connections in order of start time from a timetable. Instead of storing potentially useful connections in an auxiliary timetable, in this new algorithm, while we process all connections starting at time t, we store in a queue any vertices whose d-values are changed. Subsequently, we explore the *connected components* containing these vertices in the graph created by connections that are alive at time t, by doing a breadth-first search starting at these nodes. To find the connections that are alive at time t, we use the evolving graph data structure of Ferreira [53]. Note that unlike in Pure-Timetable, here we start scanning the timetable at the first time there is a connection starting *after* time t_q.

We proceed to prove the correctness of the algorithm.

Lemma 3.3.6 *Algorithm 7 sets* $d(u) = \delta(s, u, t_q) = t_q$ *for every node* $u \in T(s, t_q, t_q)$ *after the execution of line 6.*

Proof. The only node in Q when *ExploreConnectedComponent* is called in line 6 is *source* where $d(source) = t_q$. It is easy to see that *ExploreConnectedComponent* simply performs a BFS-like procedure, only considering connections that are alive at time t_q. Thus, every node u for which there is a journey from *source* at time t_q is explored, and $d(u)$ is set to t. □

Lemma 3.3.7 *For every* $t > t_q$*, at the end of the iteration of the main loop corresponding to time* t*, Algorithm 7 sets* $d(u) = \delta(s, u, t_q) = t$ *for every node* $u \in T(s, t_q, t)$*.*

Proof. We give a proof by induction. Assume inductively that when we start the iteration of the loop for some $t > t_q$, we have $d(u) = \delta(s, u, t_q) = t'$ for every node

Algorithm 7: Hybrid Timetable-Evolving Graph

```
   Input: $\mathcal{G}$, $\mathcal{T}$, $source$, $target$, $t_q$
1  $Initialize(d)$
2  $location \leftarrow$ beginning of $\mathcal{C}_{t_q+1}$
3  $d(source) \leftarrow t_q$
4  Initialize queue $Q$
5  $Q.enqueue(source)$
6  $ExploreConnectedComponent(Q, target, t_q)$
7  for $t \leftarrow location$ to $s$ do
8      for each connection $c$ in $\mathcal{C}_t$ do
9          Orient $c$ so that $d(c_{source}) \leq d(c_{target})$
10         if $d(c_{source}) \neq \infty$ and $d(c_{target}) = \infty$ then
11             Relax($c$)
12             Stop if $c_{target} = target$
13             $Q.enqueue(c_{target})$
14     $ExploreConnectedComponent(Q, target, t)$

15 Function $ExploreConnectedComponent(Q, target, t)$
16     while $Q \neq \emptyset$ do
17         $v \leftarrow Q.dequeue()$
18         for each edge $(v, u)$ do
19             for each interval in edge $(v, u)$ do
20                 if $d(u) = \infty$ then
21                     if $start(v, u) \leq t < end(v, u)$ then
22                         $d(u) \leftarrow t$
23                         $parent(u) \leftarrow v$
24                         Stop if $u = target$
25                         $Q.enqueue(u)$
```

$u \in T(s, t_q, t')$ for every $t' < t$. Note that the base case follows from Lemma 3.3.6. Observe that Q is empty at this point, since the termination condition of the while loop in *ExploreConnectedComponent* is that Q is empty. Consider a vertex $v \in T(s, t_q, t)$ which has a predecessor $u \in T(s, t_q, t')$ with $t' < t$ in a foremost journey from s. Then there must be a connection c from u to v starting at time t. By the inductive assumption, we have $d(u) = t'$, and after relaxing the connection c, $d(v) = t$, and v will be inserted into the queue. It is clear that any edge relaxed while scanning connections in C_t results in $d(v)$ being set to t for some vertex v. Therefore all vertices in Q have d-value equal to t after all connections in C_t are scanned. When *ExploreConnectedComponent* is called, once again, a BFS is performed using connections that are alive at time t and any vertex u that is reachable from a vertex v in Q will have $d(u) = t$. This finishes the proof. $\square$

Theorem 3.3.2 *Algorithm 7 finds the foremost journey from any source node to any target node in an evolving graph $G = (V, E)$ with timetable $\mathcal{T}$ in time $\mathcal{O}(|E| \log I + |V| + |\mathcal{T}|)$ using space $\mathcal{O}(|\mathcal{T}|)$.*

Proof. The correctness of the algorithm follows from Lemmas 3.3.6 to 3.3.7. We proceed to analyze Algorithm 7 in terms of complexity. The main challenge is to analyze the total time taken by *ExploreConnectedComponent*; it is easy to see that all other operations take at most $\mathcal{O}(|\mathcal{T}|+|V|)$ time. Observe that nodes are enqueued only one time because lines 5, 13, 25 enqueue node v only if $d(v)$ is set. Thus the total time dedicated to queue operations over the entire algorithm is equal to $\mathcal{O}(|V|)$. For each dequeued node, we explore the schedule of each neighbor using binary search. So, in total there are $|E|$ edges and the maximum number of time intervals that an edge is active is I. Thus, the algorithm spends in the worst case $\mathcal{O}(|E| \log I + |V| + |\mathcal{T}|))$. Finally, observe that the evolving data structure uses constant space for every entry in the timetable, and has size $\mathcal{O}(|\mathcal{T}|)$. $\square$

The previous analysis is tight: take an evolving graph whose base graph is a path, which has only one edge active per time step. Then there is one connection that is found in each iteration of the main loop, and only one node whose d-value is set, and which is added to the queue. Afterwards, that node is dequeued and its neighbors have to be explored, which takes $\Omega(|E| \log I)$ time. Clearly we also need $\mathcal{O}(|V| + |\mathcal{T}|)$ time to queue the vertices and scan the timetable.

3.3.3 Timetable with Auxiliary Graph

Our last algorithm, called Timetable with Auxiliary Graph, combines the merits of the previous algorithms. Instead of storing connections that may be useful later in a separate table as in Pure-Timetable, we create and maintain a *graph of currently active connections*, and explore this graph, which is of size $|E|$, rather than exploring

Algorithm 9: Timetable with Auxiliary Graph

```
   Input: $\mathcal{T}$, $\mathcal{T}_d$, $source$, $target$, $t_q$
1  $Initialize(d, G, Q)$
2  $location$ ← beginning of $\mathcal{C}_{t_q+1}$
3  $d(source) \leftarrow t_q$
4  $Q.enqueue(source)$
5  $ExploreGraph(G, Q, target, t_q)$
6  for $t \leftarrow location$ to $s$ do
7      for each connection $c$ in $\mathcal{C}_t$ do
8          Orient $c$ so that $d(c_{source}) \leq d(c_{target})$
9          if $d(c_{source}) \neq \infty$ and $d(c_{target}) = \infty$ then
10             Relax($c$)
11             Stop if $c_{target} = target$
12             $Q.enqueue(c_{target})$
13         else if $d(c_{source}) = \infty$ and $d(c_{target}) = \infty$ then
14             $G.addEdge(c_{source}, c_{target}, c_{start}, c_{end})$
15     $ExploreGraph(G, Q, target, t)$

16 Function $ExploreGraph(G, Q, target, t)$
17     while $Q \neq \emptyset$ do
18         $v \leftarrow Q.dequeue()$
19         for each edge $e = (v, u, e_{start}, e_{end})$ do
20             if $t < e_{end}$ and $d(v) \neq \infty$ and $d(u) = \infty$ then
21                 Relax($connection(e)$)
22                 Stop if $u = target$
23                 $G.remove(e)$
24                 $Q.enqueue(u)$
25             else if $d(u) \neq \infty$ or $e_{end} \leq t$ then
26                 $G.remove(e)$
```

the entire evolving graph as in Hybrid Timetable-Evolving Graph. The pseudocode is given in Algorithm 9. The correctness can be proved in a very similar way to the previous algorithms, and the upper bound on the running time follows from the fact that exploration occurs in a graph of size $|E|$; the complexity analysis is otherwise identical to Algorithm 7. The auxiliary graph uses space $\mathcal{O}(\min(|E|, |\mathcal{T}|))$. We obtain:

Theorem 3.3.3 *Algorithm 9 finds the foremost journey from any source node to any target node in an evolving graph $G = (V, E)$ with timetable $\mathcal{T}$ in time $\mathcal{O}(|E| + |V| + |\mathcal{T}|)$, using space $\mathcal{O}(|\mathcal{T}|)$.*

3.4 Journey extraction

Algorithm 10: Journey Extraction

```
Input  : $target$
Output: $\mathcal{J}$
1 $\mathcal{J} \leftarrow \emptyset$
2 $j \leftarrow target$
3 while $j \neq null$ do
4     $\mathcal{J}.insert(j, d(j))$
5     $j \leftarrow parent(j)$
```

Algorithm 10 shows how we extract a journey after any presented algorithm in this work finds the *target* node. Each node has a reference to its predecessor which is set during the journey calculation. The reference is saved as *parent*, so it is enough to follow those references until there is an *null* value, as line 3 tests. The output is a linked list $\mathcal{J}$ that describes how to reach the *target* node starting from the *source* node.

Chapter 4

Data sets

In this chapter we describe the procedure that we follow to generate evolving graphs and timetables, the data sets we use, and their characteristics. In Section 4.1, we present three different data sets which provide information about the position of nodes, and describe how we generate evolving graphs from them. In Section 4.2 once the data structures are generated for each data set; we describe some of their characteristics such as time slots, number of connections, presence ratio, dynamic ratio.

4.1 Evolving graph and timetable generation

We use three different data sets in our experiments. The first one is SUMO; it stands for Simulation of Urban Mobility [1], and refers to a well-known simulator that simulates the mobility of vehicles in a given road network. The second one is STK which stands for Systems Tool Kit [11]. This provides information about the position of real satellites all over the world. The last one is Société de Transport de Montréal (STM) [14]. It gives the real-time position of buses in the public transportation system of Montréal, Canada. Thus, we have three different type of nodes: vehicles, satellites and buses. Given a set of nodes and their positions, and a transmission range, we calculate edges and connections based on the euclidean distance between each pair of nodes. If this distance is within the transmission range at a given time, then there is a connection between the pair of nodes. The nodes are said to have an edge between them if there is ever a connection between them.

We decided to include these three data sources for the following reasons: mobility nature, type and number of nodes. Regarding the mobility nature, SUMO provides random mobility while STK and STM have real mobility. We consider different numbers of buses in SUMO and STK networks, but the number of buses in STM is fixed; we always use the entire set of buses in our experiments.

We now give more details on each of these data sets.

4.1.1 SUMO

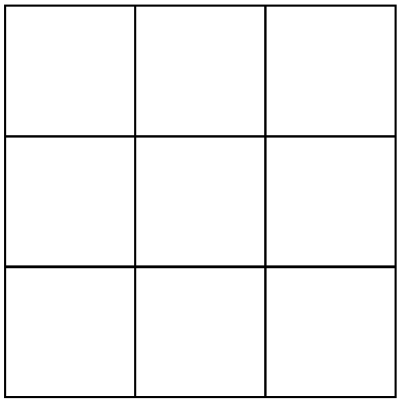


Figure 8: Road network for SUMO simulation.

SUMO is an open source vehicle traffic simulator that includes tools such as *netgenerate*, *jtrrouter*, *randomTrips.py* to model and simulate traffic [1].

Figure 9 shows the schema to obtain the position of each vehicle and parse it into a Evolving Graph and Timetable. The tool *netgenerate* outputs road networks represented as graphs where intersections are nodes and streets are edges. We use *netgenerate* to have the road network shown in Figure 8 saved as a xml file called *net.net.xml*. This road network is a grid, each cell is a square of 2000m length.

A trip is a pair of source and destination edges in the road network. We use *randomTrips.py* which is part of SUMO simulation suite to generate random trips for each vehicle. The trips are saved in file called $flows.xml$. A route in SUMO is a description of the edges that a vehicle has to take to depart from a given source edge and arrive to a given destination. To generate the route for each vehicle we use *jtrrouter* which is a SUMO tool to calculate the routes for each vehicle. The file $jtrrouter.jtrrcfg$ provides configuration parameters like turning ratios, loop allowance and all destinations acceptance. We provide 25/50/25 as turning ratio. It means a vehicle turns right with probability 25%, goes straight with probability 50% and turns left with probability 25%. Also, we allow loops in route calculation. Finally, we have *routes.xml* which encodes the edges that each vehicle has to take during its trip.

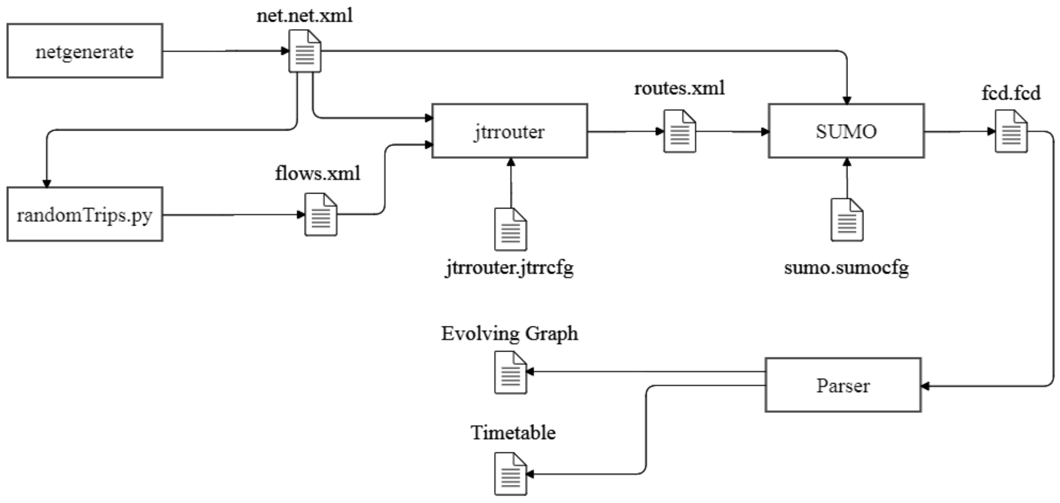
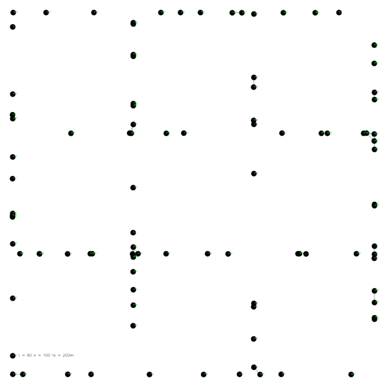


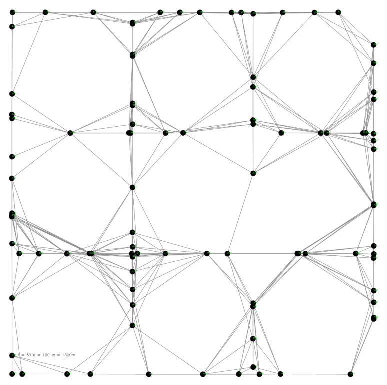
Figure 9: SUMO data parsing.

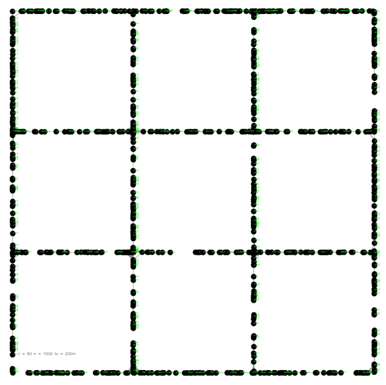
The way we provide the input to SUMO is shown in Figure 9. The *sumo.sumocfg* file is a configuration file where we specify the simulation time. It starts at time 0 and ends at time 120, each time step represents one second in SUMO. The output is *fcd.fcd* which has the position of each vehicle at every time step. So, our *Parser* module calculates the edges between each vehicle given a transmission range at every time step and outputs a Evolving Graph and Timetable files.

We have 10 different densities of nodes and 14 transmission ranges. The first set of nodes has 100 nodes and the last one has 1000 nodes, in between there are set of nodes in 100 steps. The transmission range starts with 200m up to 1500m in 100m steps. Each density of nodes has 100 different random mobility SUMO simulations running for 120 seconds on the road network shown in Figure 8 whose side is 6000m long. For instance, Figure 10 shows different density of nodes with different transmission ranges at time step 60.

As a result, if we combine every density of nodes with every transmission range then we have 14000 different Evolving Graphs and Timetables to run our experiments. Each experiment chooses 100 random pairs of $source - target$ nodes and a random t_q. Therefore, overall we have 1400000 time measurements for each algorithm. An important point to note is that a $source - target$ pair is only included in the calculation of averages if the $target$ is reachable from the $source$ node; if not, it is simply discarded, and another pair is chosen. To determine reachability, we simply run Ferriera's algorithm.


(a) 100 nodes - 200m.


(b) 100 nodes - 1500m.


(c) 1000 nodes - 200m.

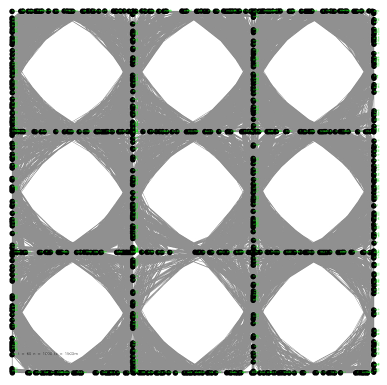

(d) 1000 nodes - 1500m.

Figure 10: SUMO different density of nodes and transmission ranges.

4.1.2 STK

The evolving graph and timetable generation for the STK data source is simple. Figure 11 shows that STK outputs a file called *positions.csv*. It has the position of each satellite at every time step as a three-dimensional coordinate. Then, to generate the evolving graph and timetable we calculate the euclidean distance at every time step between every pair of satellites. If the distance is within a given transmission range then we consider the existence of a connection between them.

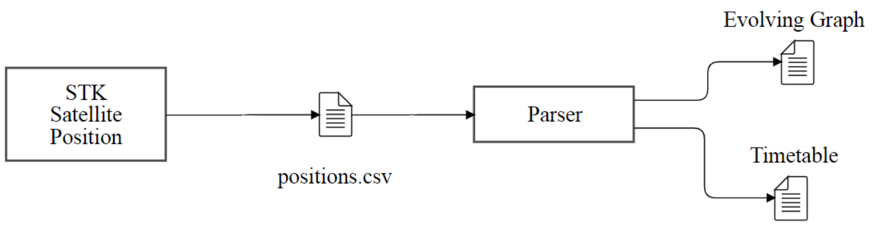


Figure 11: Satellite STK data parsing.

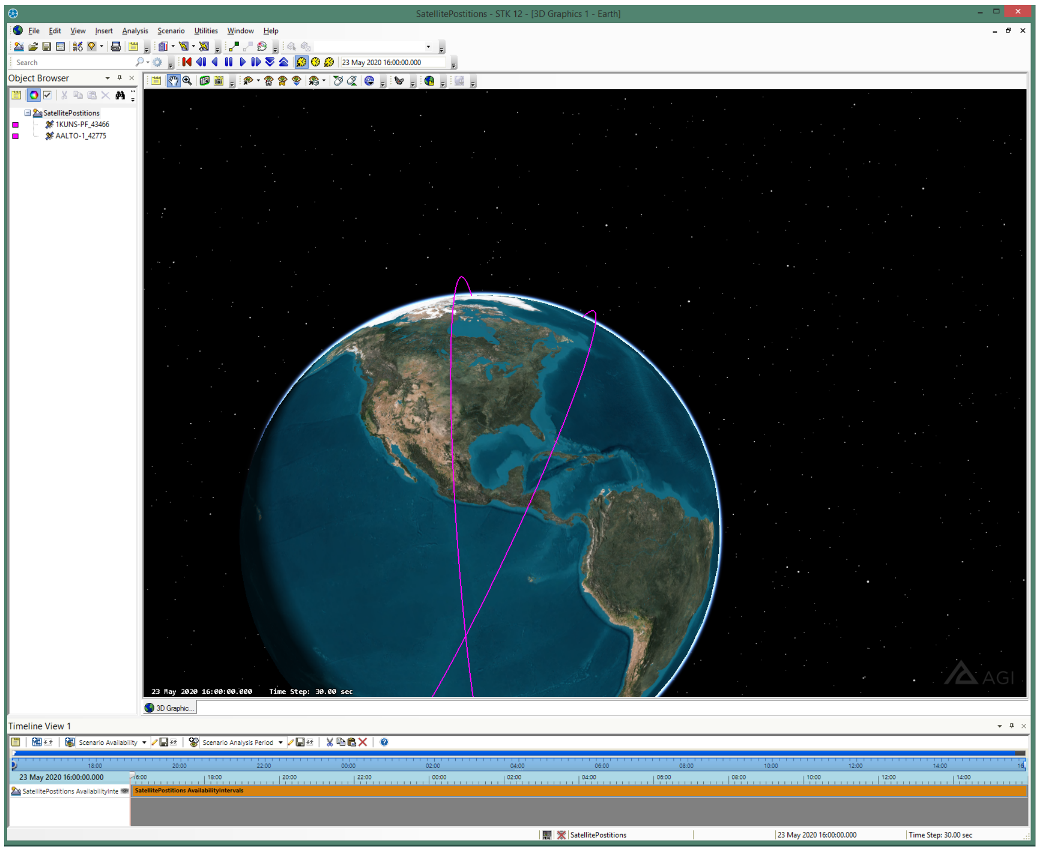

SatellitePositions - STK 12 - [3D Graphics 1 - Earth]
File Edit View Insert Analysis Scenario Utilities Window Help
Object Browser
SatellitePositions
1KUNS-PF_43466
AALTO-1_42775
23 May 2020 16:00:00.000 Time Step: 30.00 sec
AGI
3D Graphic...
Timeline View 1
Scenario Availability
Scenario Analysis Period
SatellitePositions AvailabilityIntervals

Figure 12: STK screenshot.

Figure 12 shows an example of how two satellites and their trajectory around the world look like in STK.

We start with 100 satellites and end up with 1000 satellites in 100 steps. One important feature to highlight of this data set is that it has a built-up approach. In other words, the mobility is the same for every set of satellites except for the 100 new added satellites. Regarding the transmission range, it starts with 500km and ends up with 6000km in 500km steps. If we combine each density of nodes and transmission ranges then we have 120 different Evolving Graphs and Timetables to run our experiments. Each experiment chooses 10000 random pairs of $source - target$ satellites and a random t_q. Thus, they provide 1200000 time measurements for each algorithm. Note that once again, we only include a $(source, target)$ pair, if $target$ is reachable from $source$.

4.1.3 STM

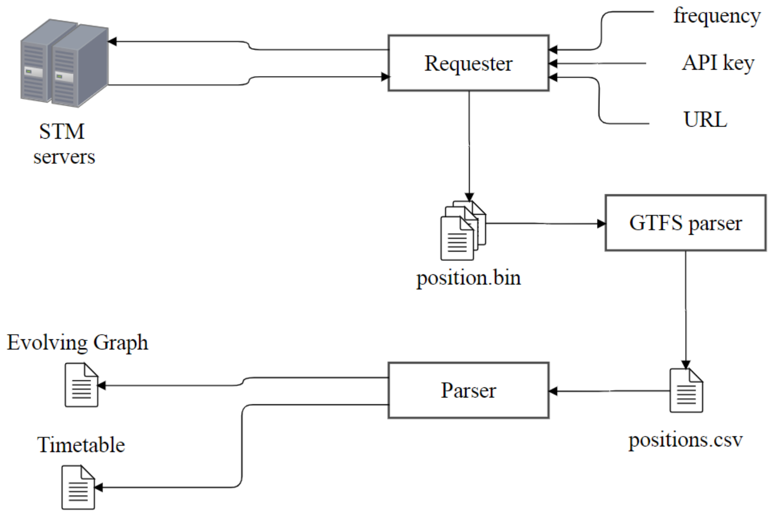


Figure 13: STM data parsing.

STM has a developer API available at `https://api.stm.info` which makes the bus real-time position publicly available. They use the GTFS Realtime [12] specification to define the position of each bus. GTFS Realtime stands for General Transit Feed Specification that allows public transportation systems to provide real-time updates regarding their transit services.

We have developed an automatic *Requester*, as Figure 13 shows, which requires the URL, an API developer key provided by STM, and the frequency. The frequency tells how often the automatic *Requester* requests the position of the buses to the STM servers. Each response is saved as a *position.bin* file. After a period of time many *position.bin* files are collected, we decode the GTFS binary data contained in each file and save all in one single plain-text file called *positions.csv*. Finally, we calculate the euclidean distance between each pair of buses and create a connection between a pair of nodes in a time step if they are within a given transmission range.

The main characteristic of this data source is that the number of nodes is constant equal to 1152 buses, that is, we use all buses. So, the transmission range is the only variable that we can modify. We start with 150m and end up with 2000m in 50m steps. Thus, we have 38 different Evolving Graphs and Timetables to run our algorithms. Similarly as SUMO and STK data set, each experiment chooses 10000 random pairs of $source - target$ buses and a random t_q. Therefore, each algorithm has 380000 samples to compare their performance. We only include a $(source, target)$ pair, if $target$ is reachable from $source$.

4.2 Characteristics of data sets

This section describes some characteristics of our data sets, namely: time slot characteristics, number of connections, presence ratio, and dynamic ratio. The time slot characteristics lets us know how many time slots the evolving graph or the timetable has and how long each time slot lasts in real time. The number of connections as its name suggest tells the size of the timetable. This attribute is useful in terms of order of magnitude.

We define two metrics in the following sections called presence ratio and dynamic ratio. Basically, presence ratio is a comparison between the dynamic graph and its static graph equivalent. Dynamic ratio is a comparison between the actual number of on-off transitions of each edge and its maximum possible value of on-off transitions.

4.2.1 Time slots

Table 1 summarizes the main characteristics regarding time slots. The slot duration means the real elapsed time between each time step. The total duration refers to the

Data Set	Mobility Nature	Max-Min Connections	Number of Time Slots	Slot Duration	Total Duration
SUMO	random	92007-209	120	1 second	120 seconds
STK	real	2278948-1048	194	7.5 minutes	24 hours
STM	real	317302-51300	1684	12.8 seconds	6 hours

Table 1: Time slots.

whole duration of simulation or data collection. In other words, the total duration is equal to slot duration times the number of time slots. STM data set is the one with the greatest number of time steps while STK has the biggest slot duration around 7.5 minutes.

4.2.2 Number of connections

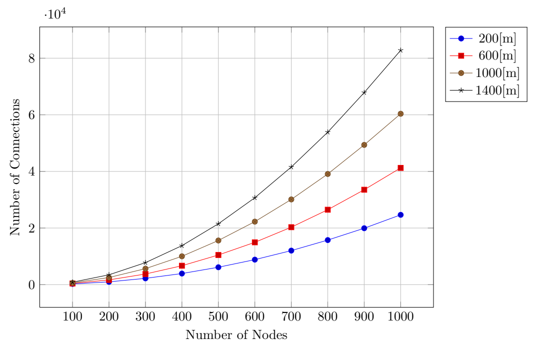
Figures 14, 16 and 17 show the average number of connections for every data set. As expected, the number of connections grows as we increase either the number of nodes or the transmission range.

The slope for SUMO and STM data sets in Figures 17b and 14b is constant as we increase the transmission range, however in Figure 16b the slope of STK data set grows faster at the beginning but then it tends to decrease as the transmission range increases. This is because the transmission range has a limit. In other words, there is a point where the transmission is so big that the network is completely connected. Figure 15 shows an evidence of our previous statement. It is an experiment in SUMO data set which its transmission range limit is around 6500m. As we can see, after 6000m there is no more new connections since every node is connected to every other node in the network.

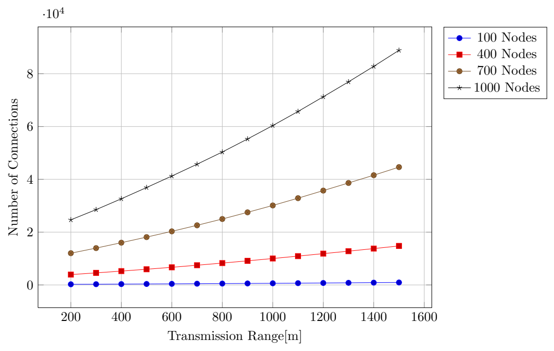
Figures 14a, 16a and 17a show how the number of connections increases as we increase the number of nodes.

4.2.3 Presence ratio

In order to know how present the edges are in an Evolving Graph $\mathcal{G}$ we compare the average degree of every node during the lifespan of $\mathcal{G}$ with the number of neighbors. Equation 6 shows how we calculate the presence ratio; s is the number of time steps,



(a) Varying the number of nodes.



(b) Varying the transmission range

Figure 14: SUMO number of connections.

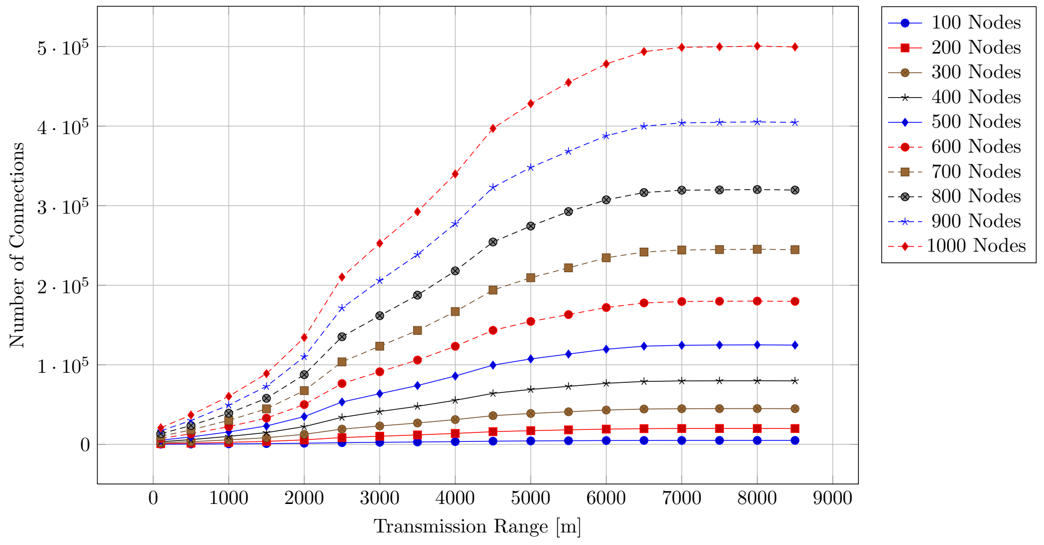
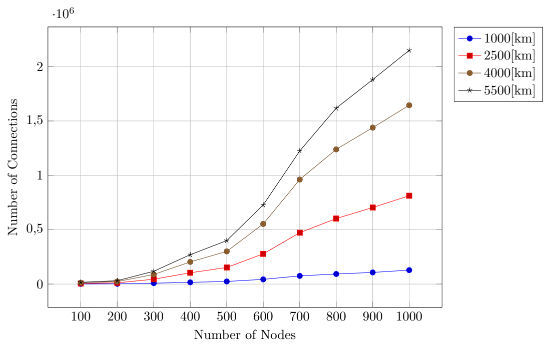
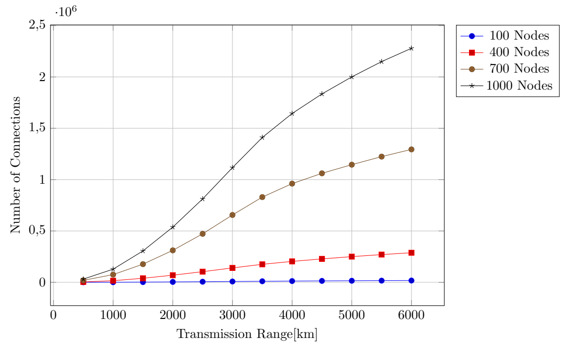


Figure 15: Number of connections in SUMO data set as the transmission range reaches its limit.

$degree(v, t)$ is the degree of node v at time t while $degree(v)$ is the number of neighbors. For example, lets consider the dynamic graph of Figure 18 where every edge is present for two time steps. If we calculate the presence ratio we have Equation 7 which shows that it is equal to 0.20. So in average the edges are present 20% of the time. So, if the presence ratio is equal to one then it means that every edge is present

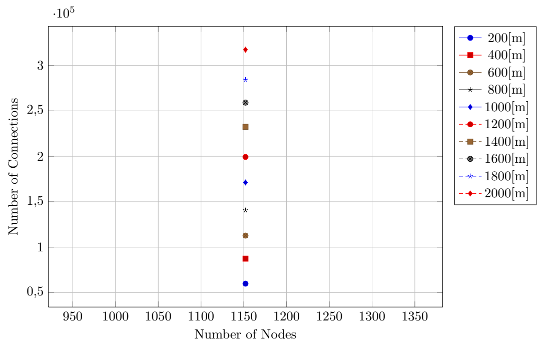


(a) Varying the number of nodes.

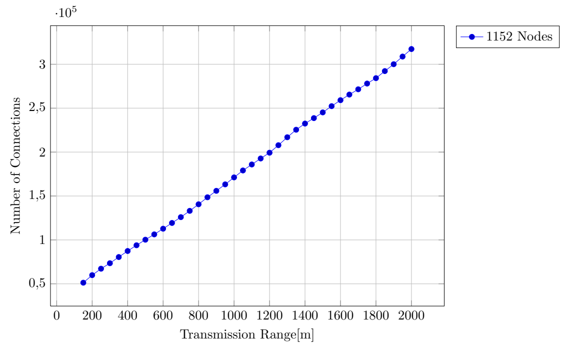


(b) Varying the transmission range

Figure 16: STK number of connections.



(a) 1152 nodes



(b) Varying the transmission range

Figure 17: STM number of connections.

all the time in $\mathcal{G}$.

$$presenceRatio(\mathcal{G}) = \frac{\sum_{v \in V} \sum_{i=0}^{s} degree(v, t_i)}{s * \sum_{v \in V} degree(v)} \tag{6}$$

$$\begin{aligned}
&= \frac{\sum_{i=0}^{10} degree(A,i) + \sum_{i=0}^{10} degree(B,i) + \sum_{i=0}^{10} degree(C,i) + \sum_{i=0}^{10} degree(D,i)}{10 * (degree(A) + degree(B) + degree(C) + degree(D))} \\
&= \frac{4+6+6+4}{10 * (2+3+3+2)} \\
&= \frac{20}{10 * 10} \\
&= \frac{1}{5}
\end{aligned} \tag{7}$$

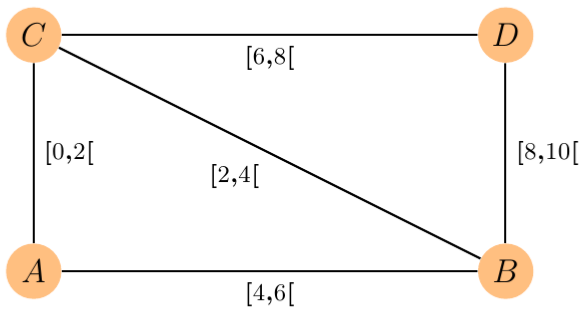
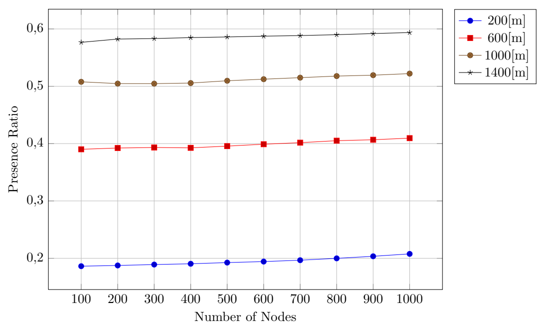
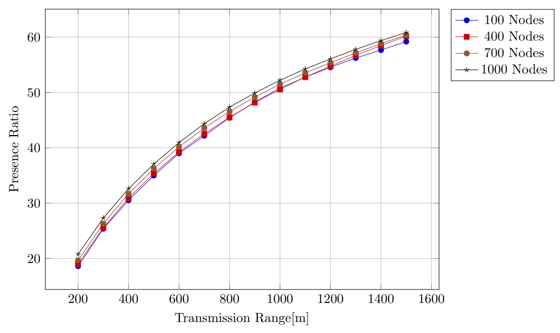


Figure 18: Presence ratio example.



(a) Varying the number of nodes

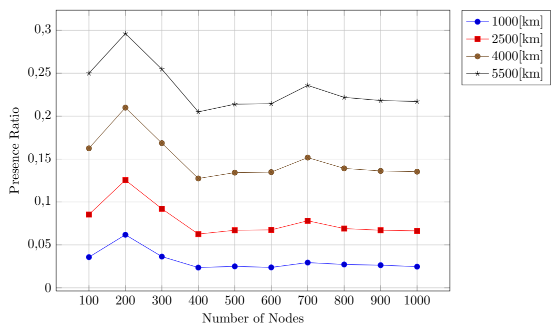


(b) Varying the transmission range

Figure 19: SUMO presence ratio.

Figures 19, 20 and 21 show how the presence ratio varies as we increase the transmission range or the number of nodes. For the SUMO data set, increasing the number of nodes for a given transmission range does not impact the presence ratio as increasing the transmission range for a given density of nodes does. Figure 19b shows that for smaller transmission ranges the slop is high and it tends to decrease as the transmission range increases. This behavior can be described as some sort of saturation since the slope grows slower as we are getting close to presence ratio equal one. The maximum values in Figures 19a and 19b are around 0.6 which means that the edges are present on average sixty percent of the time.

Figure 20a shows that STK data set has the maximum presence ratio at 200 nodes for every transmission range. After the curve reaches this peak; it decreases and then tends to be constant as we increase the number of nodes. If we take a closer look at Figure 20a then we realize that if we increase the number of nodes for a given transmission range then the presence ratio does not change. In Figures 20b and 21b, we see constant slope; as transmission range increases, the presence ratio increases


(a) Varying the number of nodes

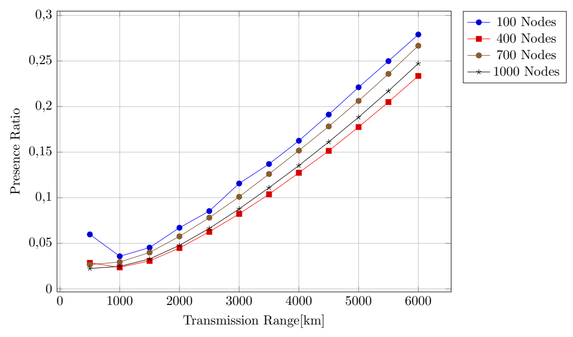
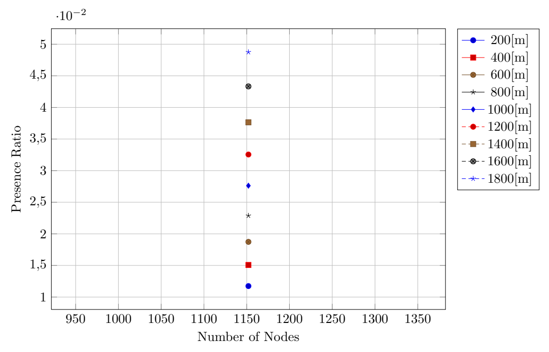

(b) Varying the transmission range

Figure 20: STK presence ratio.


(a) 1152 nodes

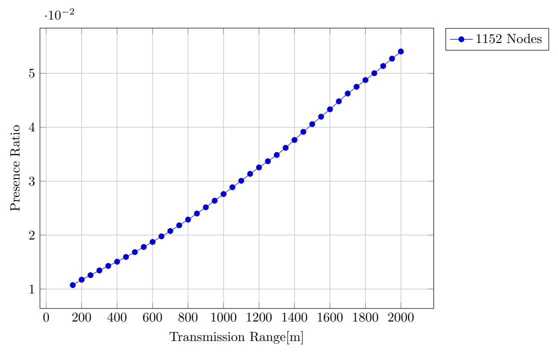

(b) Varying the transmission range

Figure 21: STM presence ratio.

linearly. Since the space where STK and STM are running is big compared to SUMO we do not saturate the presence ratio as fast as we do for SUMO.

4.2.4 Dynamic ratio

The presence ratio metric gives us an overall measure on how present the edges are in $\mathcal{G}$ but there is no information about their dynamicity. In other words, we want to know in average how many intervals the edges have. For example, let's consider an edge between node v and node a in $\mathcal{G}$ with the following schedule: $[0, 1[$, $[2, 3[$, $[4, 5[$. Figure 22 shows the plot of the previous schedule. This type of schedule is considered as highly dynamic because it has the highest frequency switching between active and non-active at every time step. In the other hand, the lowest frequency is shown in Figure 23 which has only one interval.

If we compare the plots of Figures 22 and 24; we find out that they have the same

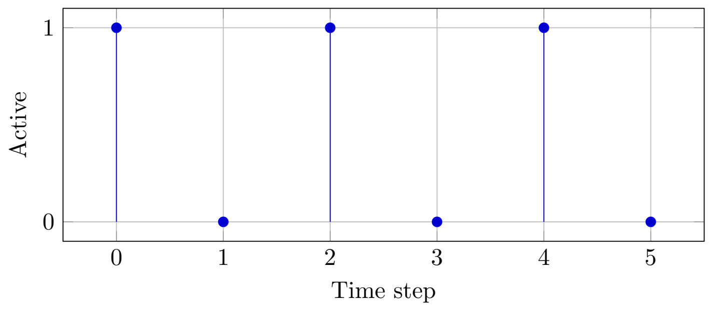


Figure 22: Schedule with intervals switching between active and non-active.

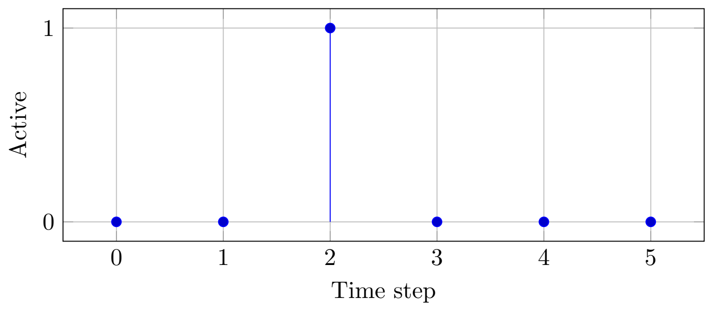


Figure 23: Schedule with only one interval.

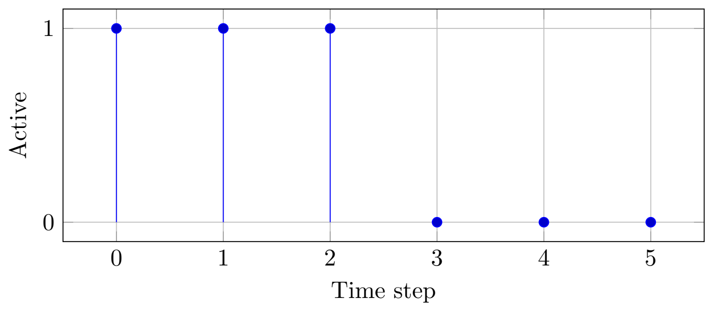
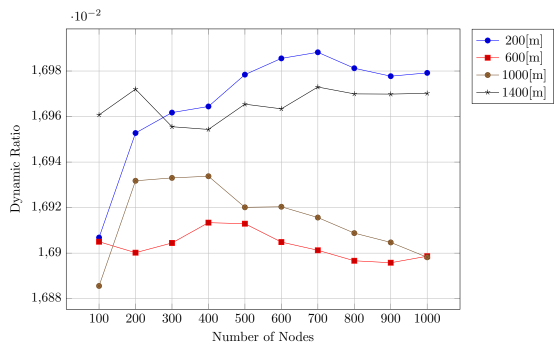
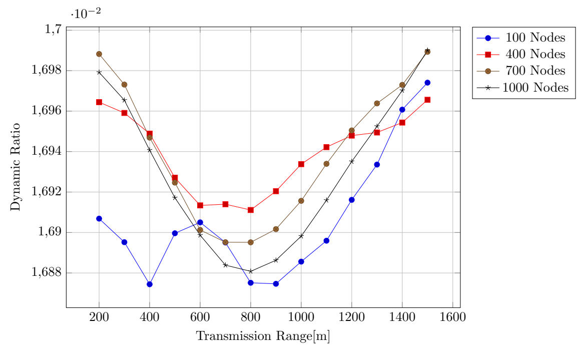


Figure 24: Schedule with one interval with three consecutive time steps.



(a) Varying the number of nodes



(b) Varying the transmission range

Figure 25: SUMO dynamic ratio.

presence ratio equal to 0.5 but Figure 24 is less dynamic because it has only one transition between active and non-active.

$$dynamicRatio(\mathcal{G}) = \frac{\displaystyle\sum_{v \in V} \frac{\displaystyle\sum_{a \in neighbors(v)} scheduleSize(v, a)}{degree(v)}}{|V| * \dfrac{s+1}{2}} = \frac{2 * \displaystyle\sum_{v \in V} \frac{\displaystyle\sum_{a \in neighbors(v)} scheduleSize(v, a)}{degree(v)}}{|V| * (s+1)} \tag{8}$$

Equation 8 shows the summation of the average number of intervals of every node in $\mathcal{G}$ divided by the maximum frequency of every schedule. $scheduleSize(v, a)$ returns the number of intervals in the edge (v, a). $degree(v)$ is the number of neighbors of v. $|V|$ is the number of nodes in V and s is the number of time steps.

The numerator of Equation 8 can be equal to the denominator only if every neighbor of every node has a schedule with $(s+1)/2$ intervals which is the maximum possible frequency. Therefore, $dynamicRatio(\mathcal{G}) = 1$ means that $\mathcal{G}$ has nodes with neighbors highly dynamic switching at every time step.

Figures 25, 27, 28 show how the dynamic ratio behaves according to the variation of the number of nodes or transmission range. SUMO data set displays in Figure 25a very tiny variations as the number of nodes increases for a given transmission range. This behaviour shows that an increment in the number of nodes does not have

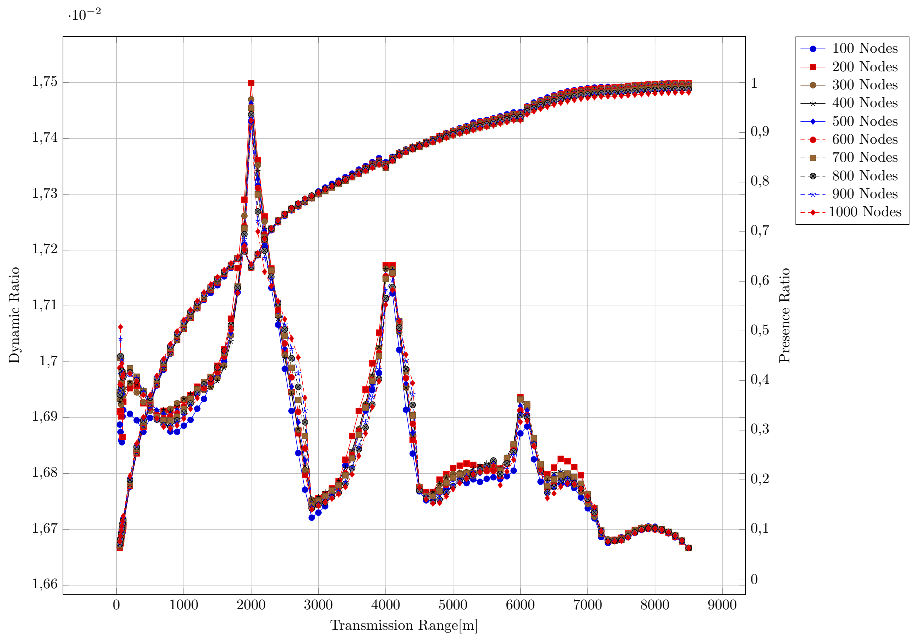
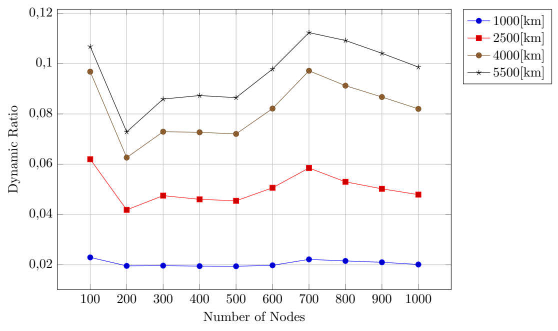


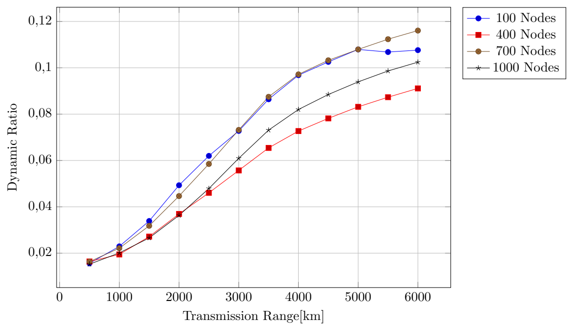
Figure 26: SUMO comparison between presence ratio and dynamic ratio.

a significant impact in the dynamic ratio since it measures the number of intervals in each schedule. In other words, it is not about the number of neighbors but the times they connect and disconnect. The transmission range provides more stable connections if we increment its value as we can see in Figure 25b. But, around 800m, nodes tend to have more intervals again. Thus, an increment in the transmission range causes more stable connections but if the increment is high enough it will add new neighbors that connect and disconnect more often. To show an evidence of our previous reasoning, for SUMO data set we increase the transmission range up to 8500m to see how the dynamic ratio behaves.

Figure 26 shows two plots; the first one is the dynamic ratio at the left axis while the second plot shows the presence ratio at the right axis. The presence ratio seems to have a logarithmic behaviour as we increase the transmission range. We also see the same concave curve that we see in Figure 25b for the first transmission ranges. At 2000m the network experiences a significant increment in its dynamic ratio meaning that there are more neighbors that connect and disconnect more often. Then, at 3000m the edges become stable with less intervals. Further transmission ranges provokes more peaks but less pronounced until it fades reaching its possible minimum value around 0.0166. Notice that at 2000m, 4000m and 6000m when the

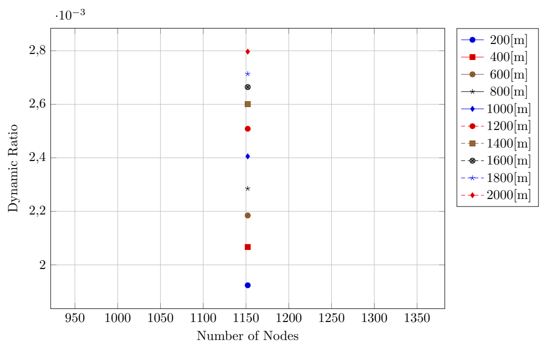


(a) Varying the number of nodes

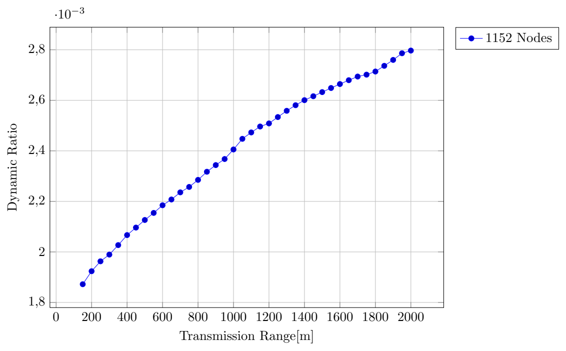


(b) Varying the transmission range

Figure 27: STK dynamic ratio.



(a) 1152 nodes



(b) Varying the transmission range

Figure 28: STM dynamic ratio.

network has a peak in the dynamic ratio the network experiences a small interruption in the presence ratio along its smooth tendency.

Figure 27a shows that for STK data set the dynamic ratio is constant for the first transmission ranges. For larger transmission ranges, the curves tend to have a low peak at 200 nodes and a high peak at 700 nodes. If we look closely to the same density of nodes in Figure 20a; we find out that the network at 200 Nodes is experiencing the same phenomenon that we described earlier for SUMO. For the same density of nodes when there is a peak in the dynamic ratio there is low point in the presence ratio. But, for 700 nodes we have peaks in both ratios. This means that the new added edges because of the increment of the transmission range are not disconnecting that often.

The dynamic ratio for STM data set in Figure 28b has the same shape of presence ratio in Figure 21b. It is showing that an increment in the transmission range provokes more present intervals.

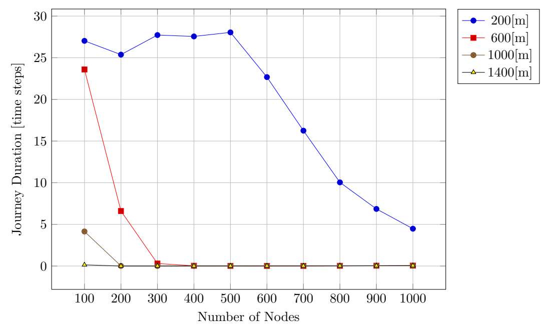
Chapter 5

Experimental results

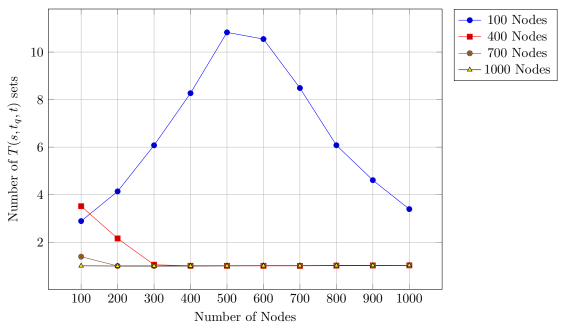
In this chapter we analyse the performance of the algorithms but first in Section 5.1 we analyse the characteristics of the computed journeys considering the average journey duration and the average of distinct journey durations. Finally, in Section 5.2, we examine and compare in terms of running time and number of hops the performance of our timetable-based algorithms against Ferreira's algorithm. We study how the algorithms behave by modifying either the transmission range or the number of nodes. We use the Timer Boost C++ [46] library to measure the running time.

5.1 Computed journey characteristics analysis

We subtract the arrival time or $d(target)$ from the query time t_q to have the travel time or journey duration. If the duration is equal to zero then the *target* node is reachable instantaneously from the *source* node. The number of distinct journey durations/arrival times is computed by finding the number of different $T(s, t_q, t)$ sets. This gives us a more qualitative idea of how much time is spent waiting. For example, suppose there are only two distinct journey durations, 0 and $\mathcal{T}_{end} - t_q$, then clearly at least for one of the possible destinations, a packet has to wait a long time before being transmitted. On the other hand, if there are $\mathcal{T}_{end} - t_q$ distinct journey durations possible, then it is likely that for many journeys, packets don't spend too much time waiting to be transmitted. The previous variables gives us a sense about the temporal properties of the journey but there is no information regarding its physical properties. This is why we consider and plot the number of hops in the next section. For instance,

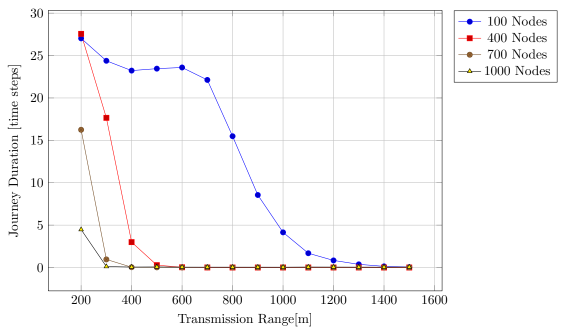


(a) Journey duration.

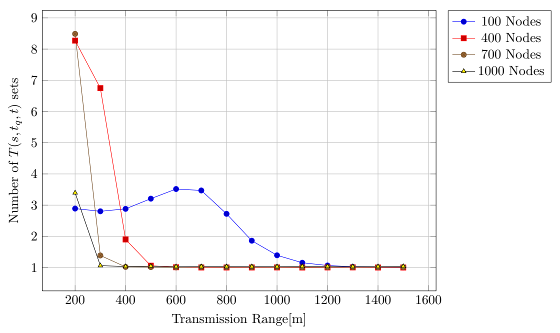


(b) Number of journey durations.

Figure 29: SUMO varying the number of nodes.



(a) Journey duration.



(b) Number of journey durations.

Figure 30: SUMO varying the transmission range.

it may be possible to have an instantaneous journey $d(target) - t_q = 0$ but with $|V| - 1$ hops.

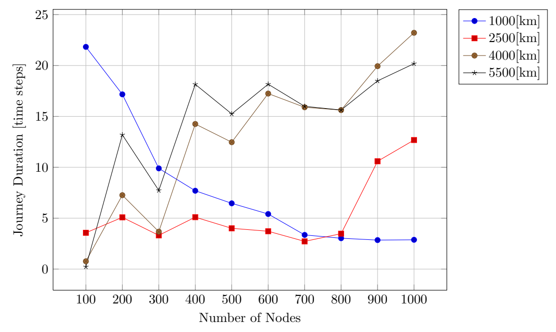
5.1.1 SUMO

We proceed to analyze SUMO journeys data set by fixing the transmission range and modifying the number of nodes. Figures 29a and 29b, show the average journey duration, and number of distinct journey durations respectively. As we can see, there is a maximum value for the average journey duration around 27 time steps for 200m transmission range with 500 nodes. We also have the maximum number of T sets at the same coordinate. We appreciate in Figure 29b the same number of T sets for different density of nodes, for instance 100 and 1000 nodes. But comparing it with Figure 29a then we find out that for 100 nodes the packet is on hold for many time steps while for 100 nodes the journey tends to be instantaneous.

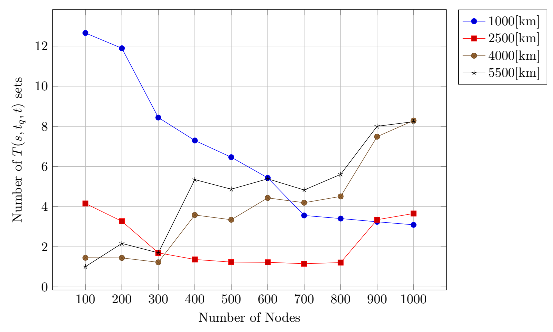
Figures 30a and 30b show the variables described before but fixing the number of nodes and varying the transmission range. There is longer journey duration for lower transmission ranges which means that the packet is on hold at some nodes while for longer transmission ranges the journey tends to be instantaneous. The number of T sets is high for lower transmission ranges and it tends to decrease as the transmission range increases.

Overall, we conclude that for the SUMO data set, the journey duration decreases as the number of nodes (i.e. density of nodes) increases, or as the transmission range (i.e. density of edges) increases.

5.1.2 STK



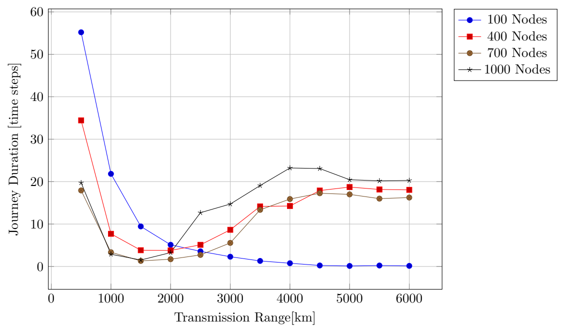
(a) Journey duration.



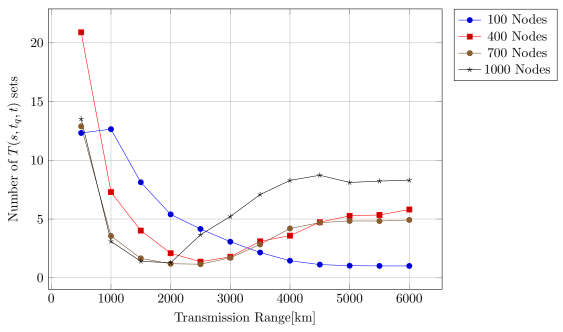
(b) Number of journey durations.

Figure 31: STK varying the number of nodes.

We perform the same analysis for STK data set. If we compare Figures 31a and 31b the we find out that both plots tend to have the same shape. For instance, for a fixed transmission range of 1000km the journey duration and the number of T sets



(a) Journey duration.



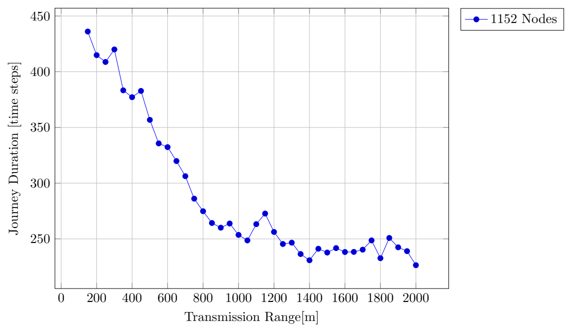
(b) Number of journey durations.

Figure 32: STK varying the transission range.

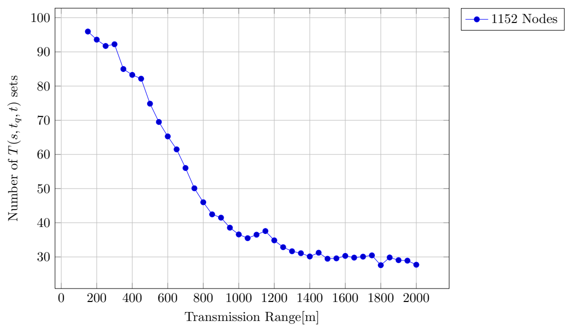
have the same decreasing tendency. This means that the packet is not on hold for many time steps. The same happens for 4000km and 5500km but with the opposite tendency; the packet tends to travel more but without holding much.

Figures 32a, 32b show the effect of varying the transmission range with a constant number of nodes in the journey characteristics. An increment in the transmission range for lower number of nodes provokes less journey duration while for higher number of nodes the journey duration reaches a lower point and tends to increase.

5.1.3 STM



(a) Journey duration.



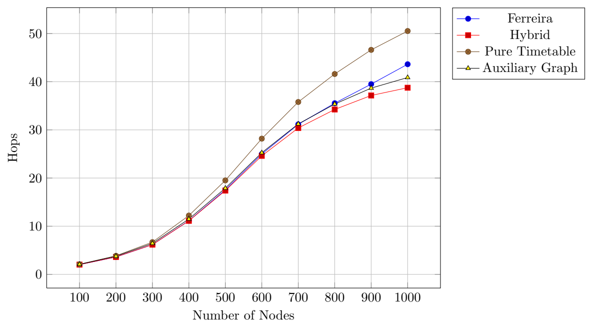
(b) Number of journey durations.

Figure 33: STM varying the transmission range.

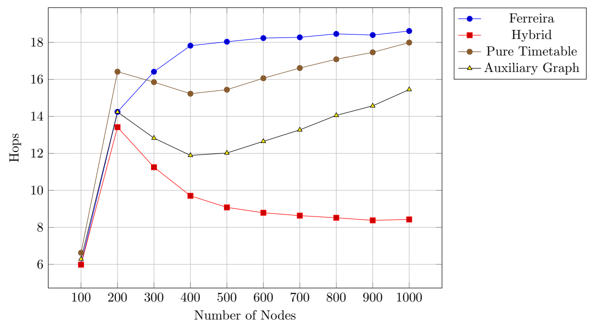
Figures 33a, 33b show the behavior of the journey durations. The range for the transmission range that we have for STM data set is wider. This data set has a lot more time slots than the previous data sets; this is why for the first transmission range 150m we have a journey duration around 450 time steps along with almost one hundred T sets. An increment in the transmission range originates a reduction in both variables.

5.2 Performance analysis

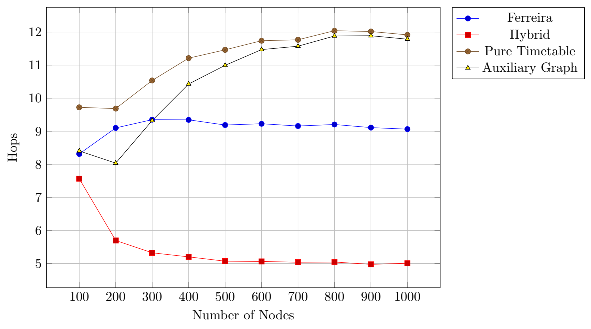
This section studies the performance of our algorithms in terms of running time. Since a sequence of connections can be traversed instantaneously, different journeys with the same duration can have different numbers of hops. So we also compare the number of hops in journeys produced by the different algorithms. As we did previously, we fix either the number of nodes or the transmission range and vary the non-fixed variable to measure the running time.



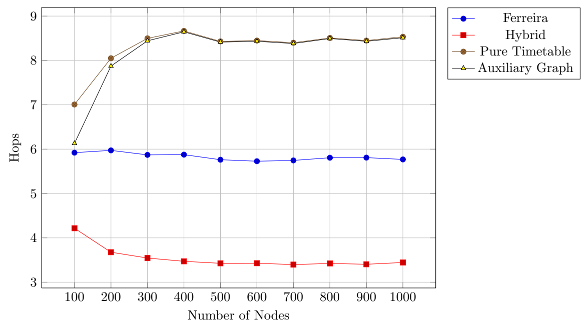
(a) 200[m]



(b) 600[m]



(c) 1000[m]



(d) 1400[m]

Figure 34: SUMO number of hops varying the number of nodes.

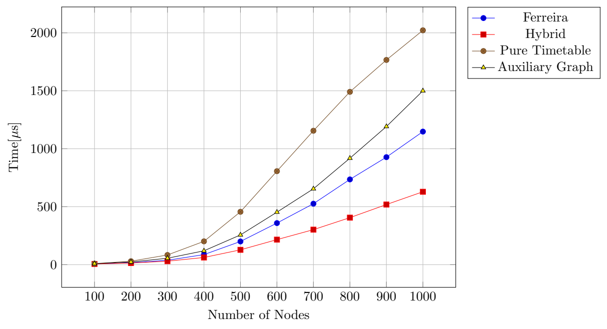
5.2.1 Effect of varying the number of nodes

In this section, we keep the transmission range fixed, and study the effect of varying the number of nodes. As stated earlier, for the STM data set, we only have one number of nodes, so the experiments in this section are only for the SUMO and STK data sets.

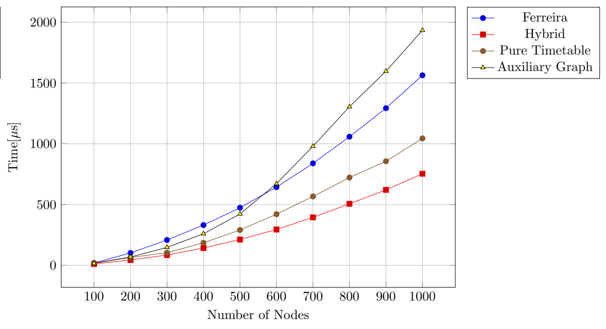
SUMO: Number of hops

Figures 34a to 34d show the effect of varying the number of nodes for fixed transmission range on the number of hops in the foremost journeys produced by the four algorithms we study. There are two things to note here. The first is that even though all algorithms compute journeys with the same arrival time for a given input, their behavior in terms of number of hops in the journey is quite different, especially at higher transmission ranges. We see that Hybrid Timetable-Evolving Graph always finds paths with the fewest number of hops, while Pure-Timetable appears to find the journey with the highest number hops.

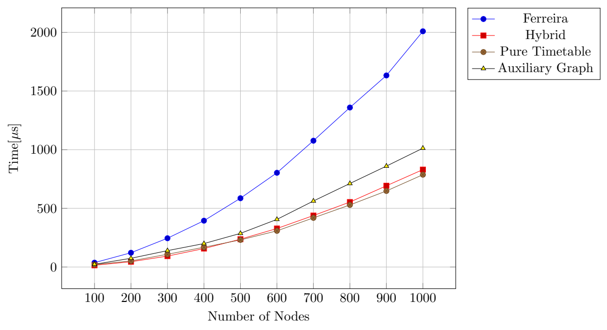
The second thing to note is that for low transmission range, the average number of hops increases with increasing number of nodes for all algorithms, while for higher transmission ranges, the number of hops does not appear to be affected by the number of nodes in the network, once there are enough nodes in the network. A likely explanation is that at low transmission ranges, nodes that are far apart are unreachable, and therefore discarded, so when the number of nodes is small, then they are close by in the network, and the foremost journey has few hops. As the number of nodes grows, we are able to reach nodes at greater distance, and greater number of hops. For higher transmission ranges, as seen in Figure 29, the journey duration is zero or close to zero, and all nodes are essentially reachable immediately, so that the number of hops does not change with increased number of nodes.



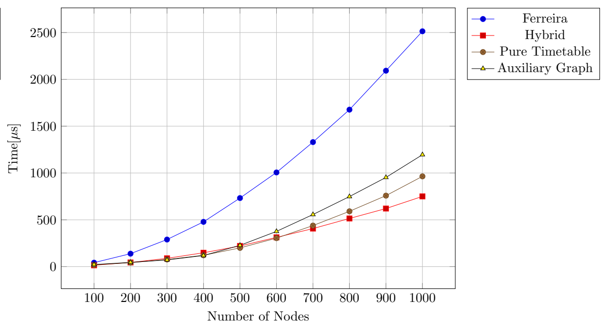
(a) 200[m]



(b) 600[m]



(c) 1000[m]



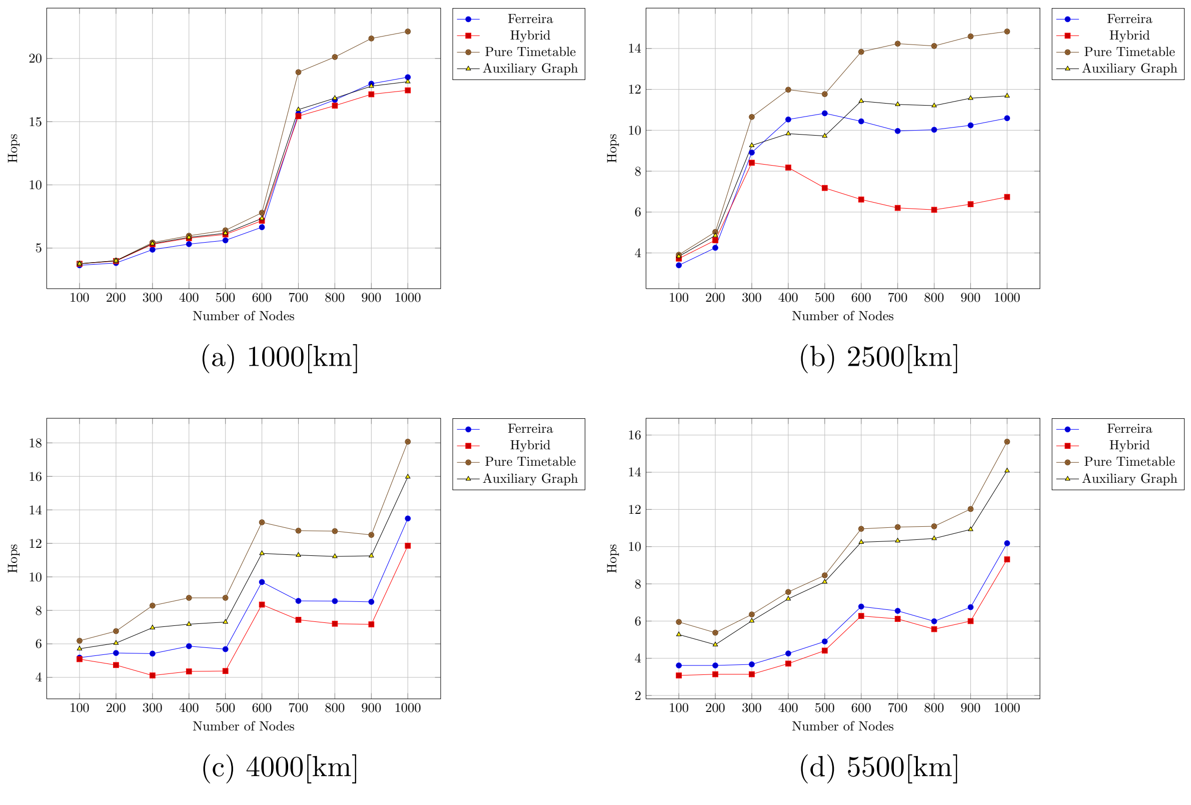
(d) 1400[m]

Figure 35: SUMO running time varying the number of nodes.

SUMO: Running time

Figures 35a to 35d show the running time of all 4 algorithms for networks with transmission range 200m, 600m, 1000m and 1400m. As expected, for each of those transmission ranges, the running time of all the algorithms increases with increasing number of nodes. We see that in all cases, and nearly all densities, Hybrid Timetable-Evolving Graph has the best running time. While for smaller transmission range, the Pure-Timetable and Timetable with Auxiliary Graph algorithms are worse than Ferriera's algorithm, for larger transmission ranges, the running time of Ferreira is the worst, and grows faster with the number of nodes than the other algorithms.

STK: Number of hops



(a) 1000[km]

(b) 2500[km]

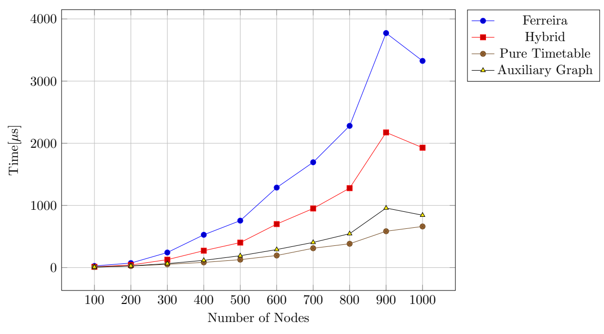
(c) 4000[km]

(d) 5500[km]

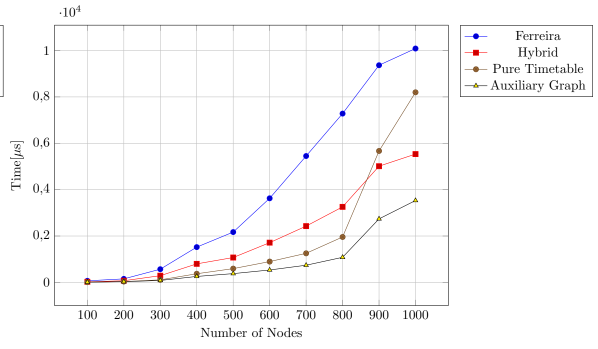
Figure 36: STK number of hops varying the number of nodes.

We see a sudden increment in the number of hops in Figure 36a around 600-700 nodes for the STK data set. Figure 36b also has a sudden increment but it occurs around 200-300 nodes and it is less pronounced. Figure 36c has two sudden changes; Figure 36d has the same shape but its slope tends to be more constant. It seems that the built-up approach for this data set provokes this effect; after a certain number of nodes more nodes are reachable with almost the same number of hops. Note that Hybrid Timetable-Evolving Graph tends to output journeys with fewer hops while Pure-Timetable outputs journeys with more hops.

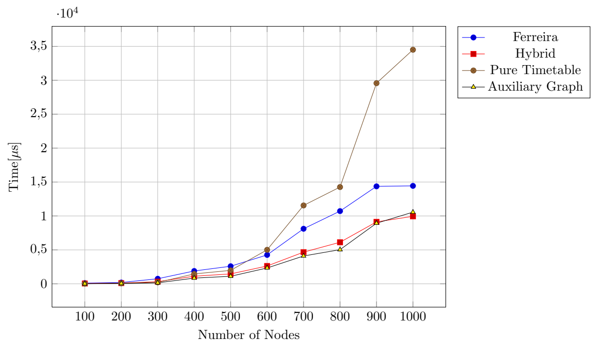
STK: Running time



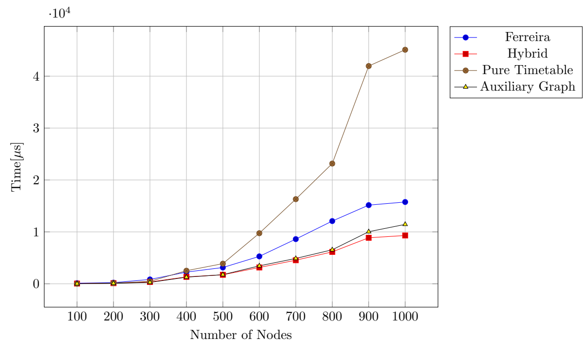
(a) 1000[km]



(b) 2500[km]



(c) 4000[km]



(d) 5500[km]

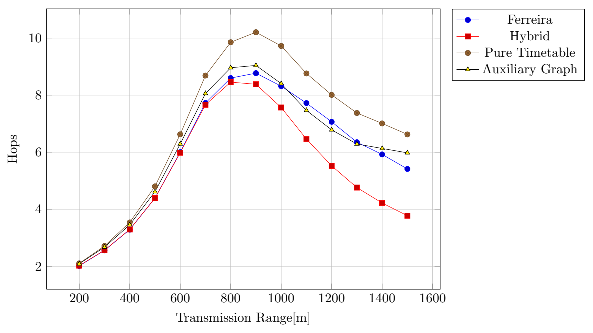
Figure 37: STK running time varying the number of nodes.

Figures 37a to 37d show the running time of all 4 algorithms for networks with transmission range 1000km, 2500km, 4000km and 5500km. We see that in all cases, and nearly all densities, Timetable with Auxiliary Graph has the best or close to the best running time. While for smaller transmission range, Pure-Timetable has the best running time no matter the node density, for larger transmission range, Pure-Timetable becomes the worst algorithm, and its running time increases sharply with the number of nodes.

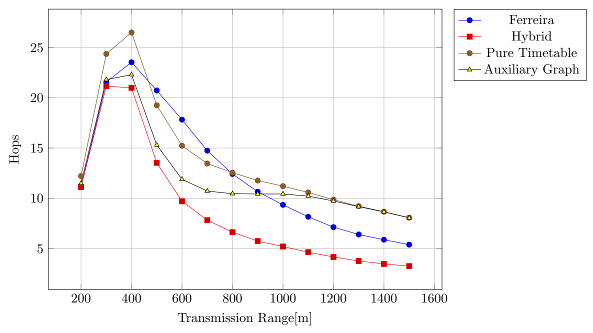
5.2.2 Effect of varying the transmission range

In this section, we keep the number of nodes fixed, and study the effect of varying the transmission range for each of our three data sets.

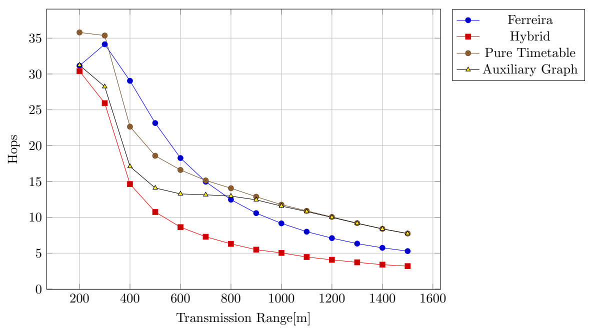
SUMO: Number of hops



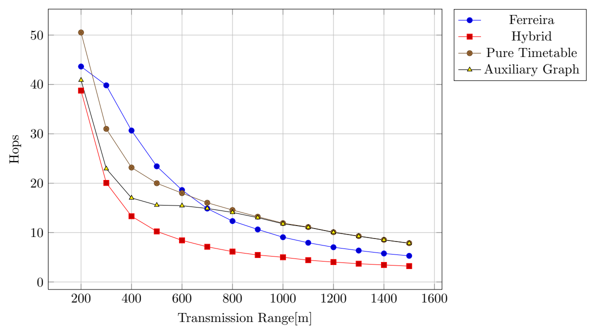
(a) 100 Nodes



(b) 400 Nodes



(c) 700 Nodes

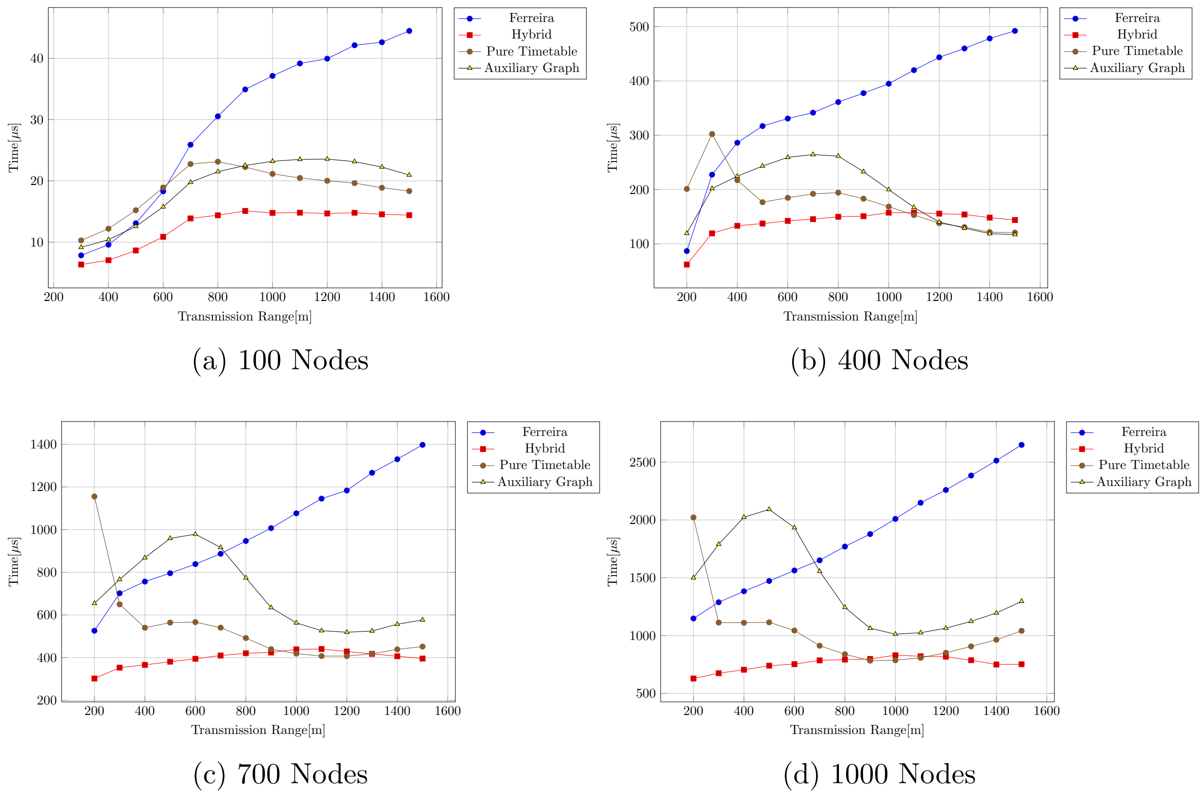


(d) 1000 Nodes

Figure 38: SUMO number of hops varying the transmission range.

Figures 38a to 40d show the effect of increasing the transmission range on the number of hops in journeys produced by the 4 algorithms. We see once again that Hybrid Timetable-Evolving Graph produces journeys with the fewest hops while Pure-Timetable generally produces journeys with more hops. An interesting observation is that for higher number of nodes, the number of hops decreases with transmission range for all algorithms, as one might expect. For 100 nodes and 400 nodes however, the number of hops in the journey first increases with increasing transmission range, and then decreases. This is likely because, for small number of nodes and small transmission range, many pairs of nodes are unreachable, As the transmission range grows, more pairs of nodes become reachable, with multi-hop paths. But as the range grows even more, all nodes become reachable, but the number of hops in the journeys starts decreasing.

SUMO: Running time



(a) 100 Nodes

(b) 400 Nodes

(c) 700 Nodes

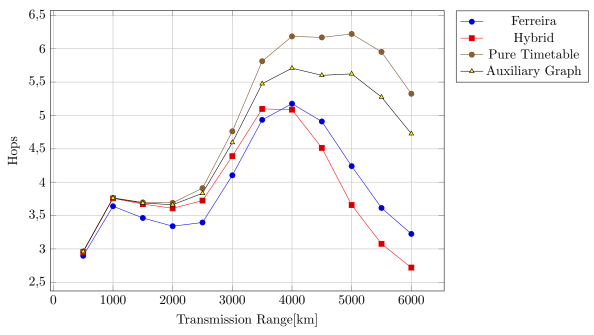
(d) 1000 Nodes

Figure 39: SUMO running time varying the transmission range.

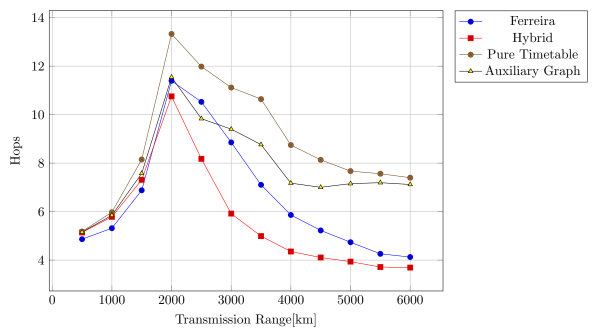
Figures 39a to 39d show the running time of all 4 algorithms for networks of 100, 400, 700, and 1000 nodes respectively. We see that in all cases, as transmission range grows, all our algorithms outperform Ferreira's algorithm. In general, Hybrid Timetable-Evolving Graph has the best time for all transmission ranges and for all numbers of nodes studied, and the running time of all timetable-based algorithms appears to stabilise with transmission range as it increases, while Ferreira's algorithms takes more running time with increasing transmission range.

STK: Number of hops

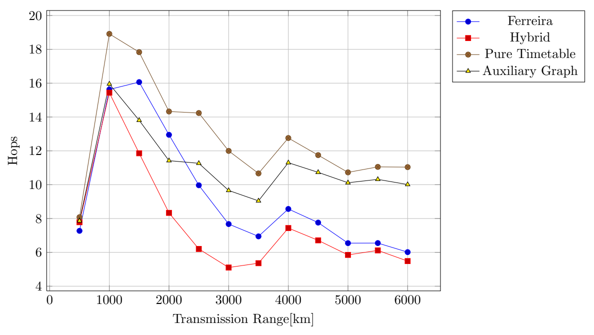
Figure 40 shows the number of hops varying the transmission range in STK data set. As in the case of the SUMO data set, the number of hops first increases as we



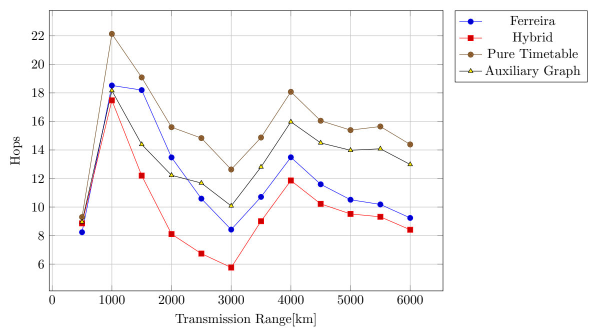
(a) 100 Nodes



(b) 400 Nodes



(c) 700 Nodes



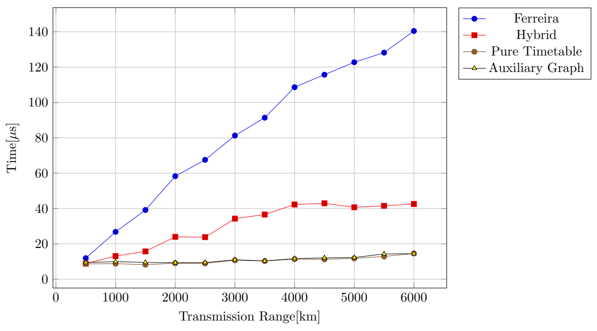
(d) 1000 Nodes

Figure 40: STK number of hops varying the transmission range.

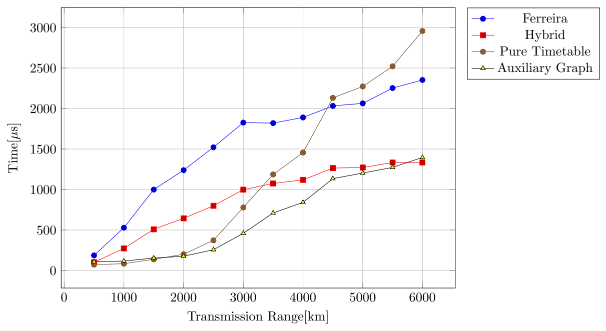
increment the transmission range, and then later generally decreases. The peak is shifted left for higher number of nodes. A similar explanation holds as in the case of the SUMO data set; the curves are less smooth because this is not a random data set. Once again Hybrid Timetable-Evolving Graph produces journeys with the lowest number of hops.

STK: Running time

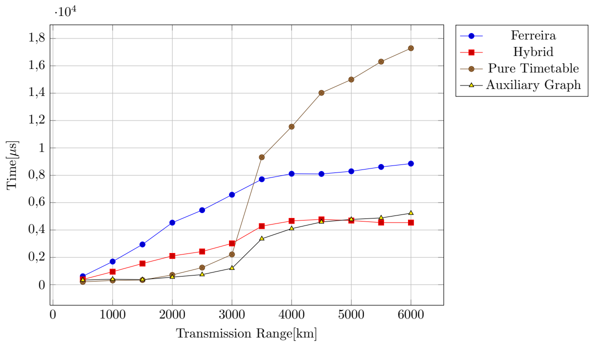
Figures 41a to 41d show the running time of all 4 algorithms for networks of 100, 400, 700, and 1000 nodes. We see that in all cases, and for nearly all transmission ranges, Timetable with Auxiliary Graph has the best running time. While Pure-Timetable works well for 100-node networks, as the number of nodes increases, the running time of Pure-Timetable degrades, especially for higher transmission ranges, becoming worse than Ferriera's approach. Hybrid Timetable-Evolving Graph also consistently has very good running time, becoming close to the time taken by Timetable with Auxiliary Graph for larger numbers of nodes.



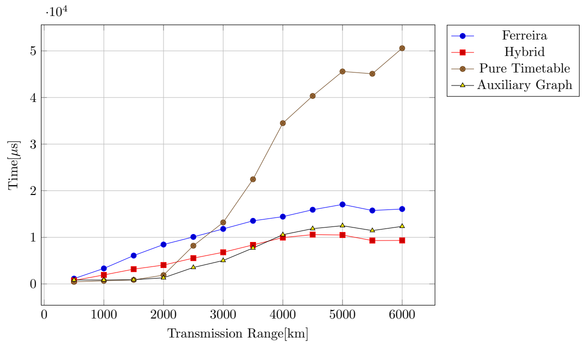
(a) 100 Nodes



(b) 400 Nodes



(c) 700 Nodes



(d) 1000 Nodes

Figure 41: STK running time varying the transmission range.

STM: Number of hops

Figure 42 shows the effect of increasing the transmission range on the number of hops in journeys produced by the different algorithms. As in the SUMO and STK data sets, the number of hops first increases with increasing transmission range, and then decreases. A similar explanation holds. Finally, Hybrid Timetable-Evolving Graph produces journeys with the fewest hops, while Pure-Timetable produces journeys with the most hops.

STM: Running time

As stated earlier, for the STM data set, we only have one number of nodes, ie. 1152 buses. Figure 43 shows the running time obtained or different values of transmission range. It can be seen that for all algorithms, the running time increases as the transmission range increases, which is to be expected, since the number of connections increases as well. For nearly all transmission ranges, all our algorithms perform better

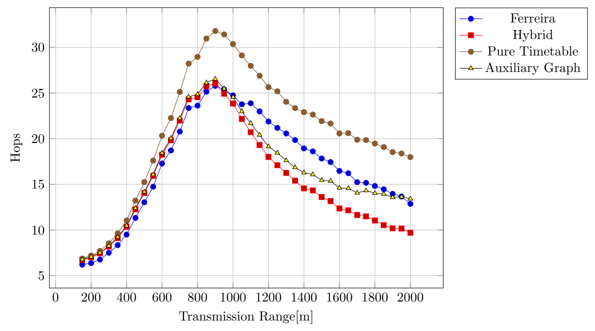


Figure 42: STM number of hops.

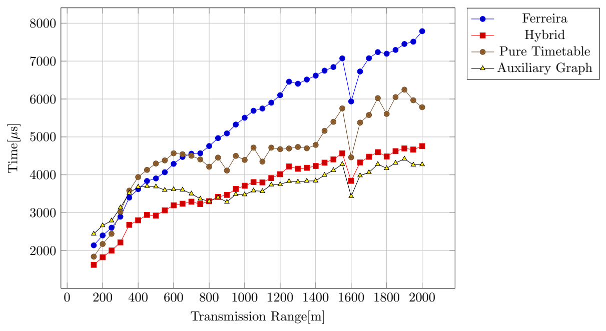


Figure 43: STM running time varying the transmission range.

than Ferreira's algorithm. For lower transmission ranges, Hybrid Timetable-Evolving Graph has the best running time, while for higher transmission ranges, Timetable with Auxiliary Graph has the best running time.

Chapter 6

Conclusions

In this thesis, we studied the problem of finding the foremost journey between a source and target node in a FSDN. We proposed a new timetable-based approach for routing in FSDN, and described three variants of this approach. We proved the correctness of our algorithms and gave tight bounds on their complexity.

Our first algorithm, called Pure-Timetable, uses only a timetable data structure, and takes time $\mathcal{O}(|V||E| + |\mathcal{T}|)$. Our second algorithm, called Hybrid Timetable-Evolving Graph , uses both a timetable data structure and the evolving graph data structure of [21], and takes time $\mathcal{O}(|E| \log I + |\mathcal{T}| + |V|)$. Our last algorithm, called Timetable with Auxiliary Graph, uses both a timetable data structure and a static graph data structure containing all edges *currently* active in the dynamic graph. It takes time $\mathcal{O}(|E| + |\mathcal{T}| + |V|)$. Here $|\mathcal{T}|$ is the total number of connections, and I is the maximum number of connections corresponding to an edge.

We also validated the performance of our algorithms using extensive simulations run on SUMO, the STK satellite data set, and the STM bus network data set. Our results showed that our algorithms, particularly, Hybrid Timetable-Evolving Graph and Timetable with Auxiliary Graph are faster than the previous algorithm given by Ferreira [21] for almost all sizes of network and all transmission ranges. We also showed that Hybrid Timetable-Evolving Graph produces journeys with fewer hops than the other algorithms.

We considered no delay for edge traversal since the minimum duration of a connection is big enough to consider propagation delay negligible in our setting. In future work, it would be interesting to modify our algorithms for the case when a connection takes a certain number of time steps to be traversed. Also, there was no propagation model [2] for the wireless communication between vehicles, buses and satellites. It would be interesting to experiment with the speed and type of the vehicles and pick one propagation model in SUMO simulations to see if they impact the journey characteristics.

Bibliography

[1] P. Alvarez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner. Microscopic traffic simulation using SUMO. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018.

[2] V. S. Anusha, G. Nithya, and S. Rao. A comprehensive survey of electromagnetic propagation models. In *2017 International Conference on Communication and Signal Processing (ICCSP)*, pages 1457–1462. IEEE, 2017.

[3] M. Y. Arafat and S. Moh. Routing protocols for unmanned aerial vehicle networks: A survey. *IEEE Access*, 7:99694–99720, 2019.

[4] S. Bakhtiari. ACATS project final report. Technical report, Canadian Urban Transit Research and Innovation Consortium, 2020.

[5] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler. Practice of streaming and dynamic graphs: Concepts, models, systems, and parallelism. *Computing Research Repository (CoRR)*, 2020.

[6] M. Bläser. Fast matrix multiplication. *Theory of Computing*, pages 1–60, 2013.

[7] F. Busato, O. Green, N. Bombieri, and D. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[8] J. Byun, S. Woo, and D. Kim. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Transactions on Knowledge and Data Engineering*, 32(3):424–437, 2020.

[9] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *Intl. Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.

[10] S. Chaudhary, N. Chaudhary, S. Sharma, and B Choudhary. High speed inter-satellite communication system by incorporating hybrid polarization-wavelength division multiplexing scheme. *Journal of Optical Communications*, 39(1):87–92, 2017.

[11] Ansys Company. Systems tool kit (STK). https://www.agi.com, 2020. Windows version 12.0.

[12] Google Technology Company. GTFS realtime overview. https://developers.google.com/transit/gtfs-realtime, Jul 2019. [Online] August 12, 2020.

[13] F. Cunha, L. Villas, A. Boukerche, G. Maia, A. Viana, R. Mini, and A. Loureiro. Data communication in VANETs: Protocols, applications and challenges. *Ad Hoc Networks*, 44:90–103, 2016.

[14] Société de transport de Montréal. Developers. http://www.stm.info/en/about/developers. [Online] accessed August 12, 2020.

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

[16] E. Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[17] EasyMile. Ez10 passenger shuttle. https://easymile.com/vehicle-solutions/ez10-passenger-shuttle. [Online] accessed March 1, 2021.

[18] D. Ediger, R. McColl, J. Riedy, and D. Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012.

[19] E. Ekici, I. F. Akyildiz, and M. D. Bender. Datagram routing algorithm for LEO satellite networks. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE*

Computer and Communications Societies (Cat. No.00CH37064), volume 2, pages 500–508 vol.2, 2000.

[20] W. Fang, M. Mukherjee, L. Shu, Z. Zhou, and G. P. Hancke. Energy utilization concerned sleep scheduling in wireless powered communication networks. In *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 558–563, 2017.

[21] A. Ferreira. On models and algorithms for dynamic communication networks: The case for evolving graphs. In *Proc. ALGOTEL*, 2002.

[22] O. Green. *High performance computing for irregular algorithms and applications with an emphasis on big data analytics.* PhD thesis, Georgia Institute of Technology, 2014.

[23] O. Green and D. Bader. custinger: Supporting dynamic graph algorithms for GPUs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[24] O. Green, R. McColl, and D. Bader. A fast algorithm for streaming betweenness centrality. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, pages 11–20. IEEE, 2012.

[25] W. Huo. *Query processing on temporally evolving social data.* PhD thesis, UC Riverside, 2013.

[26] W. Huo and V. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, pages 1–4, 2014.

[27] Transdev Canada Inc. Autonomous mobility. https://www.transdev.ca/en/our-solutions/autonomous-mobility/, Jul 2020. [Online] accessed March 1, 2021.

[28] G. Khanna, S. K. Chaturvedi, and S. Soh. On computing the reliability of opportunistic multihop networks with mobile relays. *Quality and Reliability Engineering International*, 35(4):870–888, 2019.

[29] G. Khanna, S. Soh, S. K. Chaturvedi, and K. Chin. On enumeration of spanning arborescences and reliability for network broadcast in fixed-schedule dynamic networks. *IEEE Transactions on Network Science and Engineering*, 2020.

[30] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 997–1008. IEEE, 2013.

[31] K. Kiela, V. Barzdenas, M. Jurgo, V. Macaitis, J. Rafanavicius, A. Vasjanov, L. Kladovscikov, and R. Navickas. Review of V2X–IoT standards and frameworks for its applications. *Applied Sciences*, 10(12):4314, 2020.

[32] G. Koloniari and K. Stefanidis. Social search queries in time. In *Proc. 7th Workshop Personalized Access, Profile Manage., Context Awareness Databases*, pages 1–4, 2013.

[33] A. Kosmatopoulos, A. Gounaris, and K. Tsichlas. Hinode: implementing a vertex-centric modelling approach to maintaining historical graph data. *Computing*, 101(12):1885–1908, 2019.

[34] A. G. Labouseur, J. Birnbaum, P. W. Olsen, S. R. Spillane, J. Vijayan, J. Hwang, and W. Han. The G* graph database: efficiently managing large distributed dynamic graphs. *Distributed and Parallel Databases*, 33(4):479–514, 2015.

[35] I. Maduako and M. Wachowicz. A space-time varying graph for modelling places and events in a network. *International Journal of Geographical Information Science*, 33(10):1915–1935, 2019.

[36] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, 11(3):1–34, 2015.

[37] Z. H. Mir and F. Filali. LTE and IEEE 802.11p for vehicular networking: a performance evaluation. *EURASIP Journal on Wireless Communications and Networking*, 2014(1):1–15, 2014.

[38] V. Z. Moffitt and J. Stoyanovich. Towards sequenced semantics for evolving graphs. In *EDBT*, pages 446–449, 2017.

[39] M. Nasimi, M. A. Habibi, and H. D. Schotten. Platoon–assisted vehicular cloud in VANET: Vision and challenges. *arXiv preprint arXiv:2008.10928*, 2020.

[40] R. A. Nazib and S. Moh. Routing protocols for unmanned aerial vehicle-aided vehicular Ad Hoc networks: A survey. *IEEE Access*, 8:77535–77560, 2020.

[41] P. Ni, M. Hanai, W. J. Tan, and W. Cai. Efficient closeness centrality computation in time-evolving graphs. In *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 378–385, 2019.

[42] A. Pentland, R. Fletcher, and A. Hasson. DakNet: rethinking connectivity in developing nations. *Computer*, 37(1):78–83, 2004.

[43] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment*, 4(11):726–737, 2011.

[44] C. Ren, E. Lo, B. Kao, X. Zhu, R. Cheng, and D. W. Cheung. Efficient processing of shortest path queries in evolving graph sequences. *Information Systems*, 70:18–31, 2017.

[45] R. A. Rossi, B. Gallagher, J. Neville, and K. Henderson. Modeling dynamic behavior in large evolving graphs. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 667–676, 2013.

[46] B. Schäling. *The boost C++ libraries*, chapter 38: Boost.Timer. Boris Schäling, 2011.

[47] D. Sengupta and S. L. Song. Evograph: On-the-fly efficient mining of evolving graphs on GPU. In *International Supercomputing Conference*, pages 97–119. Springer, 2017.

[48] V. R. Syrotiuk and C. J. Colbourn. Routing in mobile aerial networks. In *WiOpt'03: Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks*, pages 9–pages, 2003.

[49] A. Tripathy and O. Green. Scaling betweenness centrality in dynamic graphs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[50] K. Wehmuth, A. Ziviani, and E. Fleury. A unifying model for representing time-varying graphs. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–10. IEEE, 2015.

[51] M. Winter, R. Zayer, and M. Steinberger. Autonomous, independent management of dynamic graphs on GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[52] L. Xiangyu, L. Yingxiao, G. Xiaolin, and Y. Zhenhua. An efficient snapshot strategy for dynamic graph storage systems to support historical queries. *IEEE Access*, 8:90838–90846, 2020.

[53] B. Xuan, A. Ferreira, and A. Jarry. Computing shortest, fastest, and foremost journeys in dynamic networks. *International Journal of Foundations of Computer Science*, 14(02):267–285, 2003.

[54] Y. Yang, J. X. Yu, H. Gao, J. Pei, and J. Li. Mining most frequently changing component in evolving graphs. *World Wide Web*, 17(3):351–376, 2014.

[55] A. Zaki, M. Attia, D. Hegazy, and S. Amin. Efficient distributed dynamic graph system. In *2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS)*, pages 465–471. IEEE, 2015.

[56] A. Zaki, M. Attia, D. Hegazy, and S. Amin. Comprehensive survey on dynamic graph models. *International Journal of Advanced Computer Science and Applications*, 7(2):573–582, 2016.

[57] M. Zameni, M. Moshtaghi, and C. Leckie. Efficient query processing on road traffic network. In *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*, pages 1–6. IEEE, 2016.

[58] Z. Zhang, C. Jiang, S. Guo, Y. Qian, and Y. Ren. Temporal centrality-balanced traffic management for space satellite networks. *IEEE Transactions on Vehicular Technology*, 67(5):4427–4439, 2017.