Annales Mathematicae et Informaticae
54 (2021) pp. 163–179
DOI: https://doi.org/10.33039/ami.2021.07.001
URL: https://ami.uni-eszterhazy.hu

Route planning on GTFS using Neo4j*

Anikó Vágner

Department of Information Technology,
Faculty of Informatics,
University of Debrecen, Hungary
vagner.aniko@inf.unideb.hu

Submitted: February 1, 2021
Accepted: July 15, 2021
Published online: July 23, 2021

Abstract

GTFS (General Transit Feed Specification) is a standard of Google for public transportation schedules. The specification describes stops, routes, dates, trips, etc. of one or more public transportation company for a city or a country. Examining a GTFS feed it can be considered as a graph. In addition in the last decades new database management systems was born in order to support the big data era and/or help to write program codes. Their collective name is the NoSQL databases, which covers many types of database systems. One type of them is the graph databases, from which the Neo4j is the most widespread. In this paper I try to find the answer for the question how the Neo4j can support the usage of the GTFS. The most obvious usage of the GTFS is the route planning for which the Neo4j offers some algorithms. I built some storage structures on which the tools provided by Neo4j can be effectively used to plan routes on GTFS data.

Keywords: Graph database, GTFS, route planning

AMS Subject Classification: 68-04, 68P20

1. Introduction

Nowadays the smart city concept is very fashionable. Despite the fact that it is not well defined, it can be said that smart city concept appears when some technologies

*The work is supported by the EFOP-3.6.1-16-2016-00022 project. The project is co-nanced by the European Union and the European Social Fund.

are used to develop the life quality of the citizens who inhabit in the cities which populations are increasing [4, 5]. The smart city concept addresses many domains like transport, health, homes, buildings and environment. The focus of this paper is on the public transport of a city, in the narrow sense on the route planning for public transport.

There are a lot of working solutions to support the route planning for public transport in a big town, for example Google Maps Transit[1] for many towns around the world; Traveline[2] for Great Britain; BKK Futár[3] for Budapest, Hungary; Journey Planner of Public Transport Victoria[4] for Victoria, Australia; Rejseplanen[5] for Danmark and Journey Planner of Transport for London[6] for London. Tuaycharoen [18] also introduced one in his paper.

These information can be reached by web applications or mobile applications, but on the back-end side there are comprehensive solutions to store the schedule, run the necessary algorithms and delivers the information about the planned routes. Regarding the well-known 3-tier architecture [7], we can suppose that each of these applications comprises 3 parts: presentation tier, logic tier and data tier. The presentation tier interacts with the users, gets the departure and arrival information, shows the maps, and the resulted routes. The data tier stores the schedule information and provides access to the database. And the logic tier calculates the routes itself from the database and deliver the resulted information to the presentation tier.

The database world has had a big change in the last few years, namely beside the relational database systems a lot of new database management systems have been born to answer the problems of the big data and the application development where the in-memory data structures did not fit to the relational data model [17]. The collective name of these databases is NoSQL and it comprises many types, like key-value, document, graph and column-family.

The well-known and world-wide used source of the public transport schedule is the GTFS databases [10]. Examining a GTFS feed it can be found that it is a graph. My goal is put the content of GTFS sources into a graph database. Considering the database ranking [6] the Neo4j database system is the most popular graph database.

In this paper my goal is to analyse how the Neo4j as a database management system can support the route planning systems for public transport based on GTFS sources. In the paper I didn't consider the presentation tier, only the database and the logic tier. Additioanlly I examine only the tools and opportunities of the Neo4j for both the storage and the algorithm.

[1]`https://www.google.com/transit`
[2]`traveline.info`
[3]`http://futar.bkk.hu`
[4]`https://www.ptv.vic.gov.au/journey`
[5]`https://journeyplanner.dk/`
[6]`https://tfl.gov.uk/plan-a-journey/`

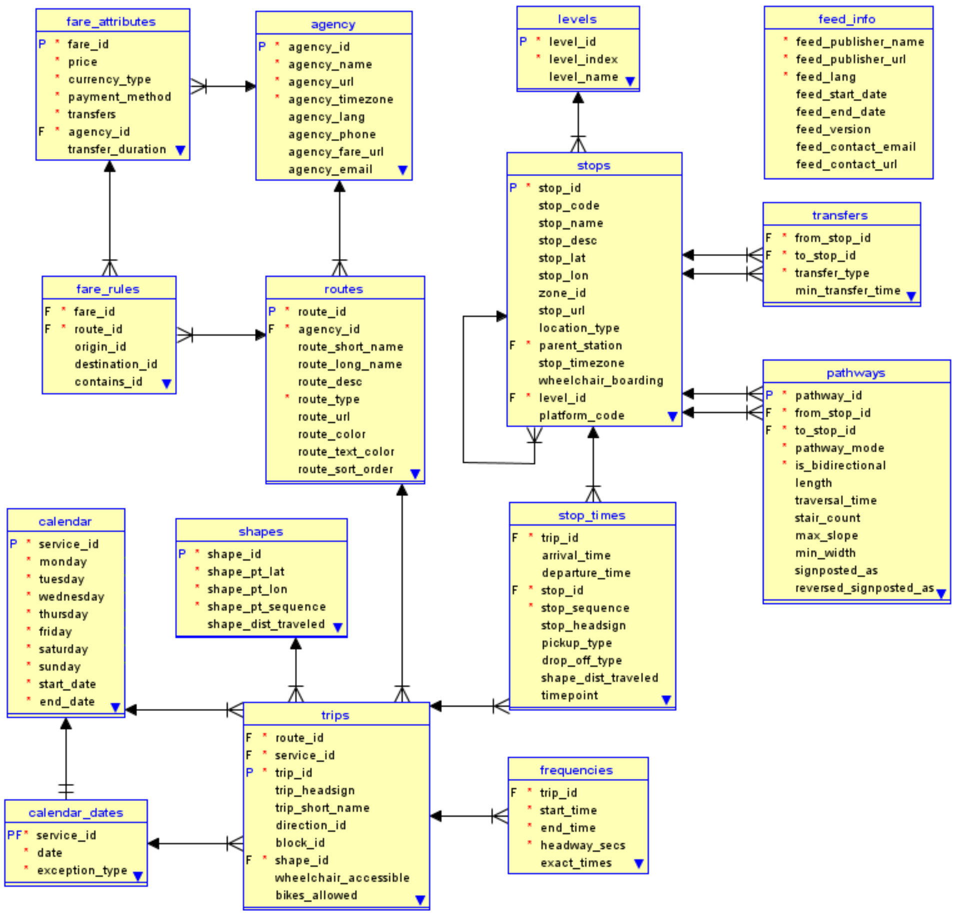

Figure 1. Model of GTFS.

2. GTFS

"The General Transit Feed Specification defines a common format for public transportation schedules and associated geographic information. GTFS 'feeds' let public transit agencies publish their transit data and allow developers write applications that consume the data in an interoperable way." [10] It was introduced by Google in 2005. [9]

The GTFS contains 15 text files in which the fields are separated by commas. The Figure 1 shows a diagram of the GTFS. I modelled it with Oracle SQL Developer Data Modeller, at the same time it is known that the GTFS is not satisfies the relational requirements. [20, 21]

5 files of the 15 are compulsory: agency, routes, trips, stop_times and stops.

Additionally at least one file is required out of the calendar_dates and calendar.

Many GTFS feeds can be downloaded from various websites. I preferred the `https://transitfeeds.com/` website, where the GTFS feeds are organized based on their location. During my work I recognized that I need local knowledge, and as I live in Debrecen I asked the local GTFS feed from Debrecen Regional Transport Association[7].

3. Neo4j

The first version of Neo4j was developed in 2002 [15]. Neo4j is a graph database management system, it can store and manage property graphs, which means that the database contains nodes and directed relationships, additionally each node and relationship can have some properties. Each node can have labels which show the roles of the nodes in the database. The relationships can have type, and each of them connects the start node to the end node. [15]

Beside the structure it is very important what kind of tools a database management system can offer for searching data in the database. The Neo4j documentation [15] states "Neo4j was built to efficiently store, handle, and query highly-connected data in your data model" and "accessing nodes and relationships in a native graph database is an efficient, constant-time operation and allows you to quickly traverse millions of connections per second per core." So I supposed that searching routes in a graph database contained data from a GTFS feed using built-in tools of Neo4j will be very easy.

4. Related work

There are research papers about storing and/or managing GTFS feed data in graph database with more or less success. In the following paragraphs I introduce a few important of them.

Fortin [9] analysed transit networks. They used GTFS feeds as source information and loaded the data into a Neo4j database. They found that their structure didn't support the route planning with Neo4j tools so in their future research they need to change the structure or use other tools for route planning.

Miler [13] stated that "graph database management systems are not routing engines and are not suitable for full graph traversal, which is used in the shortest path calculations". They also said that "if the memory is not an issue, then graph database is the right choice for the shortest path calculation."

Kaltenrieder [12] also worked with Neo4j but they enhanced it with their own program code to realize route planning. Falco [8] used Neo4j to store GTFS data but they didn't apply route planning in their system, they provide only transport information to their users. Similary Abbeyquaye [2] employed Neo4j database to store GTFS data, but he built his route engine in JavaScipt.

[7]`www.derke.hu`

Gao [11] found that their relational approach for graph search queries such as the shortest path discovery is more efficient than the algorithms of Neo4j on large graphs.

All in all, many researchers wanted to store the GTFS data in Neo4j. And as you see, it is not easy to apply the Neo4j tools for route planning on this data.

5. Load GTFS into Neo4j

My initial idea was to load GTFS data into the Neo4j as I can in the easiest way. My goal was to write a common loader which can process any GTFS feeds.

I wrote a Python program to load the data. First I used py2neo[8], but it turned out very early that it doesn't support my work, so I had to change to Neo4j Bolt Driver for Python[9].

The agency, stop, route, trip and stop_times files were load into the database a way that each row of a file become a node in the database. The labels of the nodes showed from which file they come, moreover the values in the rows became the properties of the nodes, where the optional attributes were not loaded into the database. I used the headlines of the files to name the properties of the nodes. I worked the same way on the optional files paying attention that they are optional. The optional files are: level, shape, fare_attributes, fare_rules, frequencies, transfers, pathways and feed_info. In Neo4j I followed the name convention of the Neo4j [15], so I used Camel case, beginning with an upper-case character and I didn't use plural for the nodes.

I created relationships between nodes based on the relationships between the files introduced on Figure 1. So I created the relationship listed in Table 1. When a relationship was made I deleted the appropriate property of the node which stored the connection information. The names of the relationship were followed the name convention of Neo4j, all of them are upper case, using underscore to separate words. I named the relationships in a way that I used the labels of the start and end nodes, supplementing with other information (like to or from) if it is needed.

In the shapes file many lines with the same shape_id describe a shape. So I created a ShapeID node for each shape_id and I created relationships between the Shape nodes and the ShapeID nodes, and between Trip and ShapeID. See Table 2 for the new relationships.

I found a similar problem with block_id in the case of Trip, where more than one trips can have the same block_id. So I created Block nodes, and made relationships between Trip and Block with block_id. See Table 2 for the new relationship.

The last problem was caused by the calendar and calendar_dates. I decided that I use preprocessing for these two files, so I generated a new calendar_tmp file to assign dates to service_ids. First I went through the calendar file and from the start_date to the end_date I generated all adequate dates to the service_id. Then, I went through on the calendar_dates and I deleted or inserted the {service_id,

[8]`https://py2neo.org/v4/`

[9]`https://neo4j.com/docs/api/python-driver/current/`

date pair} from the calendar_tmp if a row of calendar_dates showed it. Finally I created Service and Date labelled nodes based on the calendar_tmp and made the relationships between them.

Then I created the relationship between Trip and Service nodes with the service_ids of the Trip.

Table 1. Additional relationships of nodes in Neo4j.

Relationship name	**From node**	**To node**	**Source GTFS file**	**Column name in GTFS file**
SHAPE_SHAPEID	Shape	ShapeID	shapes	shape_id
TRIP_SHAPEID	Trip	ShapeID	trips	shape_id
TRIP_BLOCK	Trip	Block	trips	block_id
SERVICE_DATE	Service	Date	calendar_tmp	date
TRIP_SERVICE	Trip	Service	trips	service_id

At this point I loaded every information of the GTFS structure to the Neo4j and the loaded data structure follows the logic of the GTFS structure. Now I have a graph structure on which I can try out the route planning tools of the Neo4j. See the model of the loaded data structure drawn in Neo4j at Figure 2.

I tested my data loader with the GTFS feeds for Debrecen[10]; Budapest, Debrecen, Miskolc, Pécs, Tampere and Szeged[11]; and the sample feed provided by the Google[12]. I found that there are many GTFS feeds which don't contain all optional files of the GTFS.

6. Route planning tasks

In route planning the traveller wants to reach another place if they are in a given place additionally they want to leave now, so the time and the date is also important. As my goal was to examine how the Neo4j route planning tools work with GTFS data loaded into the Neo4j, it was simpler if I used stops instead of GPS coordinates. With the Harvesine formula [3] I can easily find the near stops to a given place of which the GPS coordinates are given.

The easiest routing task is to find a route between two stops if it is exists, I mean when the traveller doesn't need to change the line.

A more difficult task is when the stops are not on the same route, so the traveller should change the route. The problem can be easier if the traveller can change the route only in the stop. More difficult solution if the transfers and the pathways of the GTFS should be used. Since the GTFS feeds that I used miss these optional

[10]`derke.hu`
[11]`transitfeeds.com/`
[12]`developers.google.com/transit/gtfs/examples/gtfs-feed`

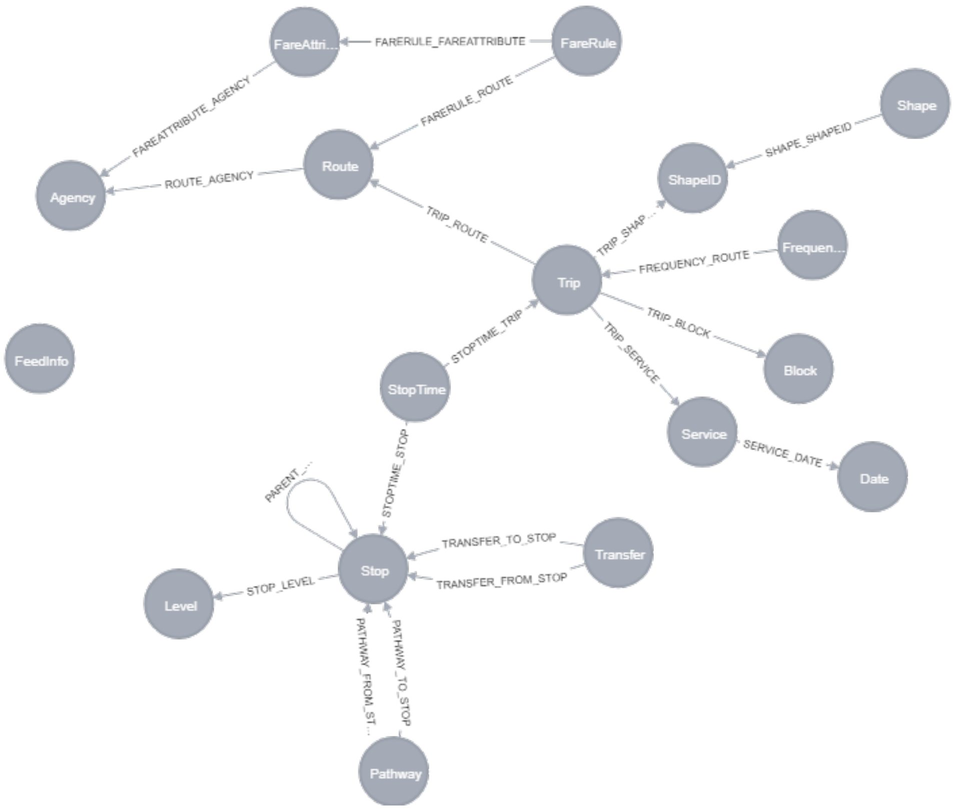

Figure 2. Model of the loaded data strucure drawn in Neo4j.

files, I didn't consider this case. I didn't consider the parent station relationship also to make the problem easier.

The most route planning algorithms offer some modes which set of routes the traveller needs: all, the shortest, the k-shortest and the applications can offer some other modes also. Of course the first that I need is the shortest, but as I use the public transport in Debrecen with the help of an app by Szincsák [19, 20], I see that it is not enough. The buses are late many times, so I prefer choosing a more frequent line with more walking than the shortest route. All routes is also not the good choice, since it will contain the routes which goes first time to the border of the city, than comes back so it takes 2 hours instead of the 10 minutes which is offered by the shortest path. So I prefer the k-shortest path. Another idea to limit the route changes in a way since nobody wants to change routes many times.

7. Introduction Neo4j opportunities for route planning on the loaded data structure

Neo4j uses the Cypher graph query language to query the graph. Its basic tool is the MATCH clause with WHERE and RETURN clauses with which the developer can find nodes and relationships in the graph.

To find the route between two stops without changing the line, the following Cypher statement can be executed:

```
match p= (startStop:Stop)-[:STOPTIME_STOP]-(st1:StopTime)-
  [:STOPTIME_TRIP]-(t:Trip)-[:STOPTIME_TRIP]-(st2:StopTime)-
  [:STOPTIME_STOP]-(endStop:Stop)
where endStop.stop_name='"Laktanya utca"'
  and startStop.stop_name='"Vezér utca"'
  and toInt(st1.stop_sequence)<toInt(st2.stop_sequence)
return p;
```

Figure 2 helps to follow the names of nodes and relationships. In this first Cypher statement I didn't use time and date for the searching, so the result contains many path between the startStop and endStop. Since the direction is important and the traveller cannot travel to the opposite direction than the vehicle goes, I should put the condition about the stop_sequence to the statement.

If I consider the date and the time also and if I want to show the route information, the following Cypher statement can be used:

```
match p= (startStop:Stop)-[:STOPTIME_STOP]-(st1:StopTime) -
  [:STOPTIME_TRIP]-(t:Trip)-[:STOPTIME_TRIP]-(st2:StopTime)-
  [:STOPTIME_STOP]-(endStop:Stop),
p2=(t:Trip)-[:TRIP_ROUTE]-(r:Route),
p3=(t:Trip)-[:TRIP_SERVICE]-(ser:Service)-[:SERVICE_DATE]-(d:Date)
where endStop.stop_name='"Laktanya utca"'
  and startStop.stop_name='"Vezér utca"'
  and toInt(st1.stop_sequence)<toInt(st2.stop_sequence)
  and d.date=" 20190503"
  and st1.departure_time>'10:00:00'
  and st1.departure_time<'11:00:00'
  and st2.arrival_time<'12:00:00'
return p,p2,p3;
```

On the Debrecen data this Cypher query works well.

In the case when the traveller should change the route, and they have this opportunity only in Stops, based on Figure 2 you can see that in the statement we should start from a Stop, than go to a StopTime as in the previous case and we should end again with a StopTime and a Stop. Between the starting StopTime and ending StopTime we should move on the STOPTIME_TRIP or STOPTIME_STOP relationships several times. If I want to get back all solutions, a Cypher statement can be used which is a modified version of the previous one, namely I have put an asterisk to the proper relationship, like this:

```
match p= (startStop:Stop)-[:STOPTIME_STOP]-(st1:StopTime)-
  [:STOPTIME_TRIP]-(t:Trip)-[:STOPTIME_TRIP|:STOPTIME_STOP*]-
  (st2:StopTime) - [:STOPTIME_STOP]-(endStop:Stop)
where endStop.stop_name='"Laktanya utca"'
  and startStop.stop_name='"Gyepusor utca"'
  and toInt(st1.stop_sequence)<toInt(st2.stop_sequence)
  and st1.departure_time>'10:00:00'
  and st1.departure_time<'11:00:00'
  and st2.arrival_time<'12:00:00'
with p, nodes(p) as nodes
where all(x in nodes where not(labels(x)='Trip')
  or exists((x)-[:TRIP_SERVICE]-(:Service)-
           [:SERVICE_DATE]-(:Date{date:" 20190503"}) ))
return p;
```

The query stores the time and date information. I solved the date information a way that every Trip in the path should run on the given day.

On the Debrecen data this Cypher query doesn't work, it causes out of memory error. I could change the memory size for the Neo4j, but to find all routes with line changes between two stops is so many solutions that it is not worth to search.

If I want to limit the route changes, the previous Cypher statement can be changed a way, that [:STOPTIME_TRIP|:STOPTIME_STOP*] part gets a limit. Following Figure 2 I found that the multiplication number of the relationships can be calculated the following way: (changes+1)*4-3. With 0 changes this number is 1, with 5 changes this number is 21. I tried out with more numbers from 5 to 21:

```
match p= (startStop:Stop)-[:STOPTIME_STOP]-(st1:StopTime)-
  [:STOPTIME_TRIP]-(t:Trip)-[:STOPTIME_TRIP|:STOPTIME_STOP*1..21]-
  (st2:StopTime)-[:STOPTIME_STOP]-(endStop:Stop)
where endStop.stop_name='"Laktanya utca"'
  and startStop.stop_name='"Gyepusor utca"'
  and st1.departure_time>'10:00:00'
  and st1.departure_time<'11:00:00'
  and st2.arrival_time<'12:00:00'
with p, nodes(p) as nodes
where all(x in nodes where not(labels(x)='Trip')
  or exists((x)-[:TRIP_SERVICE]-(:Service)-
     [:SERVICE_DATE]-(:Date{date:" 20190503"}) ))
return p;
```

In all cases the execution time of this query is very long, I don't think that it could be used in an application. Moreover, in the where clause there is a complex condition, and we should use more complex conditions than this to find the routes that we need. I tried to give hints to the execution plan, but it didn't help the query.

Then, I considered the built-in functions of Neo4j which support the route planning. Neo4j offers path finding algorithm, namely Minimum Weight Spanning Tree, Shortest Path, Single Source Shortest Path, All Pairs Shortest Path, A*, Yen's K-shortest paths and Random Walk [15].

In my work the startStop and the endStop are known, so the Minimum Weight Spanning Tree is useless for this problem. Similarly, the the All Pairs Shortest Path is not the solution at this situation since it find the shortest paths between all pairs of nodes. Than as well the Single Source Shortest Path (SSSP) algorithm is not useful in this situation as it calculates the shortest (weighted) path from a node to all other nodes in the graph. The Random Walk provides random paths in a graph, but in my case the route is not random.

The Shortest Path algorithm uses Dijkstra algorithm. To see the nodes of the route between the startStop and endStop its stream version should be used.

```
match p=(startStop:Stop{stop_name:'"Laktanya utca"'}),
        (endStop:Stop{stop_name:'"Gyepusor utca"'} )
call algo.shortestPath.stream(startStop, endStop)
yield nodeId
return algo.getNodeById(nodeId);
```

The function and so the statements works well, and its execution time is also good. However, the travel wants to move in a given date and time, so some restrictions is needed for the statement. The function doesn't offer such conditions in this form.

The next algorithm is the Yen's K-shortest paths algorithm, which computes single-source k-shortest loopless paths for a graph with non-negative relationship weights.

```
match (startStop:Stop{stop_name:'"Laktanya utca"'}),
      (endStop:Stop{stop_name:'"Gyepusor utca"'})
call algo.kShortestPaths.stream(startStop, endStop, 5, 'cost' ,{})
yield index, nodeIds, costs
return algo.getNodesById(nodeIds)
```

Similarly as the shortestPath function, the kShortestPath function and so the statements works well, the execution time is also good, but the problems are also similar as in the case of the shortestPath algorithm, namely some restrictions is needed because of the date and time.

The A* algorithm improves the shortest path algorithm that way that the user can add some information to the algorithm in order that the algorithm could make better choices over which paths to take through the graph. The syntax and the usage of the algorithm can be read in the documentation of the Neo4j [15]. It needs a kind of heuristic, the weight is compulsory, which can be the time between the stops, but we don't have in our data structure. The algorithm also needs longitude and latitude, which are given in the Stop but for the other nodes they are not given, so we cannot use this algorithm in this form on our structure.

To sum up this section the data structure introduced on Figure 2 is very good to store the data but it doesn't support the route planning in the given city. So in the next chapter I modified it to support the route planning.

8. The modified version of the data structure

Pyrga [16] compared the time-expanded and the time-dependent graphs for transport information and they find that the time-dependent graph was more compact and offers a clearly better performance. The time-expanded approach means that every event at a station is modelled as a node in the graph. In my data structure the events are modelled as StopTimes, so my data structure corresponds in some features of the time-expanded approach. The time-dependent approach means that the graph contain only one node per station and there is an edge between two stations if there is an elementary connection from the one station to the other.

Following the ideas of the time-dependent graph approach to build simpler routes in the graph, I should connect the Stops in the graph, and I should equip the relationships with information in order to find the routes easier.

I made a REACH relationship from a start node to an end node if there is a route with which a traveller can reach the end stop from the start stop. This means that if a route goes from A Stop to D Stop and between the two stops the route stops in B and C stops, I created the relationships which are introduced in Figure 3, so D can be reached from A, B and C, C can be reached from A and B, and B can be reached from A.

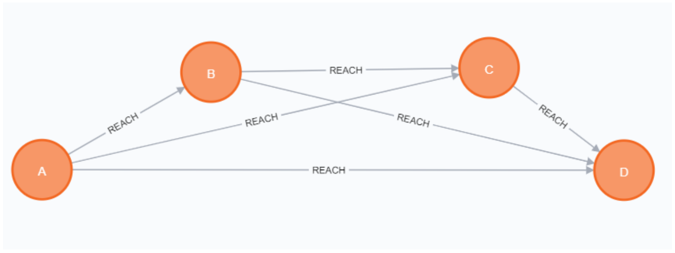

Figure 3. Routes between A and D.

Additionally I stored some information on the relationships: the route_id, the minimum and maximum duration, max_shape_distance_traveled_diff and min_shape_distance_traveled_diff, max_stop_sequence_diff and min_stop_sequence_diff. I wanted to store the trip_id list also, but the Neo4j doesn't allowed it since the list was too long.

I calculated the minimum and maximum duration based on the arrival and departure times of the StopTimes. Similarly the StopTimes stores shape_distance_traveled information, so the shape_distance_traveled_diff is the difference of the shape_distance_traveled of the two Stops. I calculated similarly the stop_sequence_diff from the stop_sequence property of the StopTimes.

A REACH relationship refers one route and not a trip. A StopTime belong to a Trip, but I wanted to make as simple the relationship as I can, so I wanted to store information for a Route. Since a Route covers many Trips, I used the maximum and the minimum values of each calculated properties. There are only a few REACH relationships for Debrecen where the min and the max for shape_distance_traveled_diff is not the same. The duration depends on the time of the day, so there are many REACH relationships where they are not the same. And finally the min and max of stop_sequence_diff-s are equal in the case of all REACH relationships for Debrecen.

With this data structure first I wanted to find the route between two stops without changing the line. It's an easy Cypher statement:

```
match (sf:Stop)-[:REACH]->(st:Stop)
where sf.stop_name='"Laktanya utca"'
  and st.stop_name='"Segner tér"'
return sf, st
```

This statement is to find all route opportunity between the two stops without any limitations, so it would run for a long time, and we know that the result would be so much that it is not worth to work with it. The following statement is for one line change:

```
match p=(sf:Stop)-[:REACH*1..2]->(st:Stop)
where sf.stop_name='"Laktanya utca"'
  and st.stop_name='"Segner tér"'
return p
```

In the result there are many Stops, since between the start and end stop there are many stops where the traveller can change the "line". If you sit on a bus, you don't want to get off and get on again, so we need to filter the result. In the filter I tried to write down that the route_id is distinct during a solution. I tried it a way that the number of the relationships in the path is the same as the number of the distinct route_ids. Since the route_id is a property of a relationship the Neo4j doesn't allow to write such a statement, additionally the count also doesn't work in this conditions. Then I tried the all predictive function in the where condition, but it doesn't allow to put two variable before its where part.

The next opportunity to use the graph algorithms of Neo4j. The first is the shortestPath algorithm. In this case the REACH relationship can be used, and the direction can be given. Moreover it is important to use a weight, since a REACH relationship can mean only 1 sec or even 1 hour. The first time I used the max_duration as weight. My first trying was the following:

```
match p= (startStop:Stop{stop_name:'"Laktanya utca"'}),
(endStop:Stop{stop_name:'"Gyepusor utca"'} )
call algo.shortestPath.stream(  startStop, endStop, 'max_duration',
  {relationshipQuery:'REACH',direction:'OUTGOING'})
yield nodeId
return algo.getNodeById(nodeId)
```

The algorithm works well, but in this form the statement doesn't know anything about the time and date. So the resulted shortest path may not exist in the necessary time interval. The same problem arises if I use the max_shape_dist_traveled as weight.

The next examined algorithm is the A* algorithm:

```
match (startStop:Stop{stop_name:'"Laktanya utca"'}),
(endStop:Stop{stop_name:'"Gyepusor utca"'})
call algo.shortestPath.astar.stream(startStop, endStop,
  'max_duration','flat', 'flon',
  {nodequery:'Stop',relationshipQuery:'REACH',
    direction:'OUTGOING', defaultValue:1.0})
yield nodeId, cost
return algo.getNodeById(nodeId)
```

It works well, but it also gives back only one solution which is not consider the date and the time, so we may not find route in a given time to reach our goal.

The last algorithm is the k shortest path algorithm:

```
match
  (startStop:Stop{stop_name:'"Laktanya utca"',stop_id: '2400205'}),
  (endStop:Stop{stop_name:'"Gyepusor utca"', stop_id: '2300507'})
call algo.kShortestPaths.stream(startStop, endStop,
  10, 'max_duration2',
  {nodequery:'Stop', relationshipQuery:'REACH',
     direction:'OUTGOING'})
yield index, nodeIds, costs
return index, nodeIds, costs
```

As we get a lot of solution, this algorithm can be useful to plan route between two stops.

I also used the shape_distance_diff as the weight to find the k-shortest path, but the function give back out of memory error. Since the previous statement worked well, at this point I changed the memory size from 0.5 GB to 4GB (the size of the database was 0.5GB), but the result was the same. I tried out with the max_duration as a weight, and it was surprising to me, since with the other weight the algorithm worked well. Additionally this weight would be better for the route planning.

I worked further with the kshortest path using the max_duration, since it works. I examined each routes whether they have appropriate date and time. If yes, it is a solution, if no, we can through this route away.

The following code is an example how we can examine the route. This statement results only a travelling between two stops without change the line. We should examine all the route, each of which are in the route list resulted the kshortest path. Since the statement was slow, I used some hints to make faster, so the speed can be acceptable.

```
match (s1:Stop{stop_id:"2400205"})-[rch:REACH]->
  (s2:Stop{stop_id:"1001605"}),
  (r:Route)-[]-(t:Trip),
  (t)-[]-(st1:StopTime)-[]-(s1),
  (t)-[]-(st2:StopTime)-[]-(s2),
  (t)-[]-(se:Service)-[]-(d:Date)
using join on s1,s2
where id(r)=rch.route_id
  and st1.departure_time>"16:00:00"
  and st1.departure_time<"18:00:00"
  and d.date=" 20190510"
return r,t,d, se, s1,s2,st1,st2
```

So the kShortestPath algorithm on this data structure with post processing to filter the date and time can be used for route planning in Neo4j on GTFS data, but as we see here, it gives error for some circumstances.

9. Other data structures

It is obvious that the stop should be nodes in the Neo4j. Additionally nodes and/or relationships should be put between two stops if the end stop can be reached from the start stop somehow. The modified structure was the one extremity where there is a relationship between two nodes if there is a route between them. The basic structure first was the other extremity, two stops are connected through the stop_times and trips. Between these two extremity I tried more opportunity, I connected the stop_times if there is a trip between them, then I connected the stops if there is a trip between them, I connected only the neighbour stops or stop_times, where the neighbour means that there are no other stops between the neighbour stops in a route or a trip. I found the same problems everywhere: I cannot write statements which can find the routes that I need, or I find something, but it doesn't work since out of memory error, or it is very slow and I cannot tune the query to be good.

10. Conclusion

Several researchers find that Neo4j doesn't support the route planning for transit transport [9, 13]. Others used Neo4j only to store transit data [2, 8, 12]. But Miller [14] states that a graph database is "the best solution if there is a need for a dynamic data model that represents highly connected data”. Abay [1] says that “the graph database model is particularly useful when data connectivity of the data is as important as the data itself”.

In my work I loaded GTFS data into the Neo4j and then I tried to apply the route planning algorithms or statements of Neo4j on it. I used the match-where-return cypher statements, than the shortest path, the k-shortest path and the A* algorithm. I changed the data structure to support the route planning. Finally I found a solution with the kShortestPath built-in function, but this function cause out of memory error in some cases.

Despite these facts I liked to work with Neo4j, it offers a browser, which can effortlesly be used, the cypher statements can easily be understood, the Neo4j Bolt Driver for Python can be used simply. But I found that the documentation doesn’t contain complex examples, I had to browse the internet for many solutions, and as I found it at the kShortestPath algorithm it contains some programming mistakes.

The Neo4j is about 17 years old, and the goal of this database management system was not the route planning. Even so I nearly found a solution for route planning on GTFS. I hope that Neo4j will be improved and after a few years it can be used also for route planning on GTFS data also.

References

[1] N. C. Abay, A. Mutlu, P. Karagoz: *A path-finding based method for concept discovery in graphs*, in: 2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA), 2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA), Corfu, Greece: IEEE, July 2015, pp. 1–6, isbn: 978-1-4673-9311-9, doi: `https://doi.org/10.1109/IISA.2015.7388092`, url: `http://ieeexplore.ieee.org/document/7388092/` (visited on 02/01/2021).

[2] A. Abbeyquaye: *Building a map-based transit planner for the tro-tro system in Accra, Applied Project*, Ashesi University College, 2017.

[3] N. R. Chopde, M. K. Nichat: *Landmark based shortest path detection by using A* Algorithm and Haversine Formula*, International Journal of Innovative Research in Computer and Communication Engineering 1.2 (2013), pp. 298–302.

[4] R. Dameri, A. Cocchia: *Smart city and digital city: Twenty years of terminology evolution*, in: X Conference of the Italian Chapter of AIS, ITAIS2013, 2013, pp. 1–8.

[5] R. P. Dameri: *Searching for Smart City definition: a comprehensive proposal*, INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY 11.5 (Oct. 30, 2013), pp. 2544–2551, issn: 2277-3061, doi: `https://doi.org/10.24297/ijct.v11i5.1142`, url: `https://cirworld.com/index.php/ijct/article/view/1142ijct` (visited on 02/01/2021).

[6] *DB-Engines Ranking*, 2021, url: `https://db-engines.com/en/ranking`.

[7] R. Elmasri, S. Navathe: *Fundamentals of database systems*, 6th ed, OCLC: ocn586123196, Boston: Addison-Wesley, 2011, 1172 pp., isbn: 978-0-13-608620-8.

[8] E. Falco, I. Malavolta, A. Radzimski, S. Ruberto, L. Iovino, F. Gallo: *Smart City L'Aquila: An Application of the "Infostructure" Approach to Public Urban Mobility in a Post-Disaster Context*, Journal of Urban Technology 25.1 (Jan. 2, 2018), pp. 99–121, issn: 1063-0732, 1466-1853,
doi: https://doi.org/10.1080/10630732.2017.1362901,
url: https://www.tandfonline.com/doi/full/10.1080/10630732.2017.1362901 (visited on 02/01/2021).

[9] P. Fortin, C. Morency, M. Trépanier: *Innovative GTFS Data Application for Transit Network Analysis Using a Graph-Oriented Method*, Journal of Public Transportation 19.4 (Dec. 2016), pp. 18–37, issn: 1077-291X, 2375-0901,
doi: https://doi.org/10.5038/2375-0901.19.4.2,
url: http://scholarcommons.usf.edu/jpt/vol19/iss4/2/ (visited on 02/01/2021).

[10] *GTFS Static Overview*, 2021,
url: https://developers.google.com/transit/gtfs/.

[11] Jun Gao, Jiashuai Zhou, J. X. Yu, Tengjiao Wang: *Shortest Path Computing in Relational DBMSs*, IEEE Transactions on Knowledge and Data Engineering 26.4 (Apr. 2014), pp. 997–1011, issn: 1041-4347,
doi: https://doi.org/10.1109/TKDE.2013.43,
url: http://ieeexplore.ieee.org/document/6475943/ (visited on 02/01/2021).

[12] P. Kaltenrieder, J. Parra, T. Krebs, N. Zurlinden, E. Portmann, T. Myrach: *A Dynamic Route Planning Prototype for Cognitive Cities*, in: Designing Cognitive Cities, ed. by E. Portmann, M. E. Tabacchi, R. Seising, A. Habenstein, vol. 176, Series Title: Studies in Systems, Decision and Control, Cham: Springer International Publishing, 2019, pp. 235–257, isbn: 978-3-030-00316-6 978-3-030-00317-3,
doi: https://doi.org/10.1007/978-3-030-00317-3_10,
url: http://link.springer.com/10.1007/978-3-030-00317-3_10 (visited on 02/01/2021).

[13] M. Miler, D. Medak, D. Odobašić: *The shortest path algorithm performance comparison in graph and relational database on a transportation network*, PROMET - Traffic&Transportation 26.1 (Feb. 28, 2014), pp. 75–82, issn: 1848-4069, 0353-5320,
doi: https://doi.org/10.7307/ptt.v26i1.1268,
url: https://traffic.fpz.hr/index.php/PROMTT/article/view/1268 (visited on 02/01/2021).

[14] J. J. Miller: *Graph Database Applications and Concepts with Neo4j*, in: SAIS 2013 Proceedings, 2013, pp. 1–24.

[15] *Neo4j*, 2021,
url: https://www.neo4j.com.

[16] E. Pyrga, F. Schulz, D. Wagner, C. Zaroliagis: *Efficient models for timetable information in public transportation systems*, ACM Journal of Experimental Algorithmics 12 (June 2008), pp. 1–39, issn: 1084-6654, 1084-6654,
doi: https://doi.org/10.1145/1227161.1227166,
url: https://dl.acm.org/doi/10.1145/1227161.1227166 (visited on 02/01/2021).

[17] P. J. Sadalage, M. Fowler: *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*, Upper Saddle River, NJ: Addison-Wesley, 2013, 164 pp., isbn: 978-0-321-82662-6.

[18] N. Tuaycharoen, A. Sakcharoen, W. Cha-aim: *Bangkok Bus Route Planning API*, Procedia Computer Science 86 (2016), pp. 441–444, issn: 18770509,
doi: https://doi.org/10.1016/j.procs.2016.05.075,
url: https://linkinghub.elsevier.com/retrieve/pii/S1877050916304112 (visited on 02/01/2021).

[19] A. Vágner, T. Szincsák: *Data structure to store GTFS data efficiently on mobile devices*, Journal of Computer Science and Software Application 1.1 (2014), pp. 27–41.

[20] A. Vágner, T. Szincsák: *Public transit schedule and route planner application for mobile devices*, in: Proceedings of the 9th International Conference on Applied Informatics, Volume 2, The 9th International Conference on Applied Informatics, Eger, Hungary: Eszterházy Károly College, 2015, pp. 153–161, isbn: 978-615-5297-19-9,
doi: https://doi.org/10.14794/ICAI.9.2014.2.153,
url: http://icai.ektf.hu/icai2014/papers/ICAI.9.2014.2.153.pdf (visited on 02/01/2021).

[21] J. C. Wong: *Use of the general transit feed specification (GTFS) in transit performance measurement*, Georgia Institute of Technology, 2013.