Adapting, analyzing and benchmarking transportation routing algorithms for realistic real-time environments

Yoeri Otten
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
y.d.otten@student.utwente.nl

ABSTRACT

There are many algorithms to find paths within a public transportation timetable, often relying upon preprocessing or complex data structures to speed up the process. While these algorithms have undergone many tests, there has not been a review with these algorithms with real-time realistic data. This makes it difficult to choose the most suitable algorithm necessary in most real-world scenarios. We compare several commonly used algorithms adapted for real-time data and test them in static and real-time environments to figure out which problems are encountered, and which algorithm(s) best adapt to those problems.

Keywords

Transit routing, Analysis of algorithms, Graph algorithms, Benchmarking, Real-time data

1. INTRODUCTION

While using public transport, the public often relies on apps and websites, such as 9292[1] (a Dutch public transportation app), to find out which buses or trains to take to get to their destination. With ever-increasing timetables, more integrated networks, and higher demand from travelers. there is a need for algorithms to reliably, accurately, and quickly perform these queries.

One important factor for travelers is that their travel information is up-to-date. Travelers like to be informed early of expected delays, and expect their travel planner to immediately show them the impact schedule changes have on their trip. This adds another layer of complexity to this set of algorithms, since they cannot be reliant on slow preprocessing steps to improve their speeds.

To see how different routing algorithms cope with this extra requirement, we set out a methodology to adapt, analyze, and benchmark a set of different popular algorithms within a pseudo-real-time environment based on actual timetable data, to find which algorithms perform best in accordance with their functionality.

First, we introduce the notation and problems which are used for describing the problems and working of the algorithms. Next, we cover earlier work related to the development and benchmarking of transit routing algorithms.

Finally we lay our methodology and present our results.

[1] https://9292.nl/

34th Twente Student Conference on IT Jan. 29th, 2021, Enschede, The Netherlands.

1.1 Research questions

We have organised our research along the following questions:

1. What are the challenges of working within a real-time environment for transit routing algorithms?
2. How do or can different transit routing algorithms cope with a real-time environment?
3. Which algorithm(s) adapt(s) best to a real-time environment in accordance with their functionality?

In the methodology, we discuss our approach to answering these questions.

2. PRELIMINARIES

In this section, we define the notation and use this notation to define our problems.

2.1 Notation

We base our notation upon the one specified in [6].

We define a timetable to consist of a quadruple $(\mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{F})$ of stops $\mathcal{S}$, connections $\mathcal{C}$, trips $\mathcal{T}$, and footpaths $\mathcal{F}$. A connection consists of a quintuple:

$$(c_{\text{dep\_stop}}, c_{\text{arr\_stop}}, c_{\text{dep\_time}}, c_{\text{arr\_time}}, c_{\text{trip}})$$

Where $c_{\text{dep\_stop}} \in \mathcal{S}$, $c_{\text{arr\_stop}} \in \mathcal{S}$, and $c_{\text{dep\_time}} \leq c_{\text{arr\_time}}$.

A trip is defined as an ordered list of 1 or more connections where for each connection within a trip $c^i_{\text{trip}} = c^{i+1}_{\text{trip}}$, $c^i_{\text{arr\_time}} \leq c^{i+1}_{\text{dep\_time}}$, and $c^i_{\text{arr\_stop}} = c^{i+1}_{\text{dep\_stop}}$.

A footpath is a triple which connects two stops:

$$(f_{\text{dep\_stop}}, f_{\text{arr\_stop}}, f_{\text{dur}})$$

We note that for all stops a, b and c we have:

$$(a, b, x) \in \mathcal{F} \land (b, c, y) \in \mathcal{F} \implies (a, c, z) \in \mathcal{F} \land z \leq x + y$$

Each stop also contains a loop-back to the same stop:

$$\forall_{a \in \mathcal{S}}\ (a, a, x) \in \mathcal{F}$$

We also define the set of journeys $(\mathcal{J})$. A journey (j) is a list of one or more connection pairs $(l^i_{\text{enter}}, l^i_{\text{exit}})$ within the same trip, and footpaths. A journey always starts and ends with a connection pair. For a pair $l^i = (l^i_{\text{enter}}, l^i_{\text{exit}})$ the following holds true:

$$(l^i_{\text{enter}})_{\text{trip}} = (l^i_{\text{exit}})_{\text{trip}} \land (l^i_{\text{enter}})_{\text{dep\_time}} \leq (l^i_{\text{exit}})_{\text{arr\_time}}$$

A journey alternates between pairs and footpaths:

$$l^0, f^1, l^2, f^3 \ldots f^{k-1}, l^k$$

For each triplet l^i, f^{i+1}, l^{i+2} we have:

$$(l^i_{\text{exit}})_{\text{arr\_time}} + f^{i+1}_{\text{dur}} \leq (l^{i+2}_{\text{enter}})_{\text{dep\_time}} \wedge$$
$$(l^i_{\text{exit}})_{\text{arr\_stop}} = f^{i+1}_{\text{dep\_stop}} \wedge f^{i+1}_{\text{arr\_stop}} = (l^{i+2}_{\text{enter}})_{\text{dep\_stop}}$$

Next we define some information of the journey:

$$j_{\text{dep\_time}} = (l^0_{\text{enter}})_{\text{dep\_time}},\ j_{\text{dep\_stop}} = (l^0_{\text{enter}})_{\text{dep\_stop}}$$
$$j_{\text{arr\_time}} = (l^k_{\text{exit}})_{\text{arr\_time}},\ j_{\text{arr\_stop}} = (l^k_{\text{exit}})_{\text{arr\_stop}}$$

Where k denotes the last element in j.

We can now define journey domination, where one journey should always be preferred over another:

$$\forall_{a\in\mathcal{J}}\forall_{b\in\mathcal{J}}\ a_{\text{arr\_time}} \leq b_{\text{arr\_time}} \wedge a_{\text{dep\_stop}} = b_{\text{dep\_stop}}$$
$$\wedge\, a_{\text{arr\_stop}} = b_{\text{arr\_stop}} \iff a \preceq b$$

Finally, we introduce a replacement action: $c_a(c_b)$. This means the mentioned connection a gets replaced by connection b in the timetable. $c_a(\varnothing)$ deletes connection a.

2.2 Problem definitions

In this section, we define the different problems that our algorithms should be able to solve using the above notation.

2.2.1 *Earliest Arrival Problem*

The earliest arrival problem (EAP) is to find a journey a with a set *dep_stop* and *arr_stop* at time t such that it arrives as early as possible: $\forall_{b\in\mathcal{J}}\ b_{\text{dep\_time}} \geq t \implies a \preceq b$

2.2.2 *Earliest Arrival Profile Problem*

The earliest arrival profile problem (EAPP) is to find *all* journeys $\mathcal{A}$ from a preset *dep_stop* and *arr_stop* which depart and arrive between the set times t_{start} and t_{end} and are solutions to the earliest arrival problem.

$$\forall_{a\in\mathcal{J}}\ a_{\text{dep\_time}} \geq t_{\text{start}} \wedge a_{arr\_time} \leq t_{\text{end}} \wedge$$
$$a_{\text{dep\_stop}} = \text{dep\_stop} \wedge a_{\text{arr\_stop}} = \text{arr\_stop} \wedge$$
$$(\forall_{b\in\mathcal{J}}\ b \preceq a \implies b_{\text{dep\_time}} < a_{\text{dep\_time}})$$
$$\implies a \in \mathcal{A}$$

2.2.3 *Pareto-set*

Some algorithms which solve these problems can optimize on multiple criteria. In those cases, the algorithms will return a Pareto-set of results where no journey in the resulting set has criteria *strictly* better than any other journeys in the set. Some examples of criteria to search on are:

- *Earliest arrival*
- Lowest number of transfers
- Lowest fares
- Highest reliability

2.3 Example

We define a timetable with the data found in Table 1. For the purpose of this example, there are no footpaths between stops, except for the stop-loops which will be noted by f_s.

A visualization of these routes can be found in Figure 1.

Solving the EAP on this timetable at $t = 0$ gives us the result: $j_{\text{solution}} = [(c_1, c_4)]$.

After we perform actions $c_4(\varnothing)$ and $c_7(\varnothing)$, a solution would be:

$$j_{\text{solution}} = [(c_1, c_1), f_{s_2}, (c_5, c_6), f_{s_1}, (c_3, c_3)] \vee$$
$$j_{\text{solution}} = [(c_3, c_3)]$$

	$c_{\text{dep\_stop}}$	$c_{\text{arr\_stop}}$	$c_{\text{dep\_time}}$	$c_{\text{arr\_time}}$	c_{trip}
c_1	s_1	s_2	0	5	t_1
c_2	s_1	s_5	10	15	t_2
c_3	s_1	s_3	22	30	t_3
c_4	s_2	s_3	7	15	t_1
c_5	s_2	s_4	10	15	t_4
c_6	s_4	s_1	17	20	t_4
c_7	s_5	s_3	17	20	t_2

Table 1. Example timetable

We note that both solutions would solve the EAP since this problem only asks for a solution at a given start time that arrives the earliest, not one which solves the last possible departure for the first possible arrival. However, most algorithms avoid visiting a stop twice within a journey.

A solution to *EAPP* from $t_{\text{start}} = 0$, and $t_{\text{end}} = 22$ would be:

$$\mathcal{A} = \{[(c_1, c_4)], [(c_2, c_7)], [(c_3, c_3)]\}$$

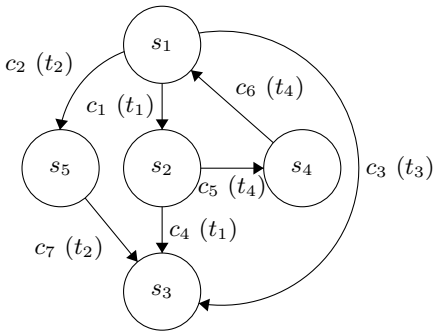

Figure 1. Visualization of our example timetable

3. RELATED WORK

3.1 Algorithms

In the past ten years algorithms which solve these problems have been hot-topic due to interest by large mapping companies [10].

The first ideas for algorithms were mainly graph-based [8]: *Time-expanded* (TE) and *Time-dependent* (TD) graphs. Both approaches work by creating a graph structure of the timetable and finding the shortest path in that graph by using Dijkstra's algorithm [7]. TE formed a graph by making every event (connection, transfer) a node, while TD modeled each route (set of all connections between two stops) as an edge between stops. Currently, only TD is still used in practice for its generally suitable performance in solving simple queries [2]. However, does not perform well in multi-criteria queries.

Many additions and improvements upon these algorithms have been made as discussed in [8]. They often rely extensively on preprocessing of the graph to filter out unnecessary transfers and adding 'speed-up' edges for easier querying.

Recent developments in the field tend to favor non-graph-based solutions [2]. In 2010, the transfer patterns algorithm (TP) was introduced [1], which is based upon the fact that most transfers in a timetable occur at only particular stops. Preprocessing those transfer locations allows for a significant speedup of query times. A derivative of the TP algorithm was introduced [3], which stores connections based on frequency, reducing the number of connec-

tions significantly and accelerating the preprocessing. This acceleration can also be used in graph-based approaches.

In 2012, RAPTOR was introduced [5]. This algorithm approaches the problems by expanding its network each round with all of the directly reachable stops without an interchange. RAPTOR solves for a bicriteria Pareto-set on the number of interchanges and first-arrival. Moreover, the authors introduced two derivatives: McRAPTOR solves multi-criteria queries and rRAPTOR which solves for a range in time and allows for more criteria as well. However, these multi-criteria algorithms, while more useful, are significantly slower than RAPTOR itself.

In 2014, the Connection Scan Algorithms (CSA) was introduced [6]. CSA works by looping through an ordered list of all connections. This approach allows for smaller data structures and has been shown to work very quickly. However, a downside is that the amount of connections increases exponentially with the amount of stops which are in the timetable. Transport for London alone serves over 4,5 million connections each day [6].

More recent work has been looking to increase speed by limiting existing algorithms while still achieving reasonable results. An example of this is described in [4], which introduces *Bounded McRAPTOR*.

4. METHODOLOGY

In this section we discuss how we answered our research questions. To do this we have several steps of building up a benchmarking system which allows us to test a multitude of algorithms. These steps are as follows:

1. Gather timetable data
   - Static data
   - Live data
2. Choose & analyze algorithms
   - Adapt algorithm to live environment
3. Design & implement benchmarking system
4. Implement algorithms
5. Perform benchmarks & interpret results

We elaborate on these steps below.

4.1 Gather timetable data

To perform our benchmarks we need timetable data to execute against. For qualitative and realistic testing there are some requirements the timetable data need to fulfill:

- **Realistic:** realistic data allow our algorithms to be tested in an environment corresponding to that of real-life systems. Realistic data can mostly be found by using publicly available timetable data.
- **Sizeable:** to better differentiate between different (implementations of) the algorithms, we need a suitably sized data set.
- **Detailed:** preferably the timetable should contain detailed information of the surrounding infrastructure, allowing us to realistically simulate footpaths.

For the dataset(s)we also need to find suitable live datafeeds which can be coupled to the static data. This allows us to gather and make changes in our timetable during the benchmarking.

4.2 Choose & analyze algorithms

We need to choose a set of algorithms which solve the problems as described above. Preferably the chosen algorithms are modern, efficient, and used in real-life products and scenarios.

Before we can start implementing the algorithms we first need to analyze them and discuss our approach to making them suitable for real-life updating. For the analysis we discuss the presented data structures used by the authors of the algorithm, then we present strategies to adapt these data structures to be able to cope with live-data, and finally we choose a strategy which is used for testing.

4.3 Design & implement benchmarking system

Next, we need a benchmarking system to perform the benchmark. It is important that our system allows us to record the necessary data. The system should be suitable to implement the algorithms in.

4.4 Implement algorithms

Within the benchmarking system our algorithms need to be implemented. For each algorithm two implementations will be made: one which mimics the presented data structures by the original authors and one implementation which is adapted to support live-data.

For the first implementation the data structures as described by the authors will be used. For the real-time implementation the data structure as discussed is used.

4.5 Perform benchmarks & interpret results

Finally, we perform our benchmarks. Through the results we should be able to compare the different implementations of each algorithm and determine how well they adapt to the real-time environment.

5. RESULTS

5.1 Datasets

For testing we used two different datasets focused on different goals.

Both data sets are relatively small compared to the amount of trips which take place in public transport in the whole of Europe every day. However due to the constraints of this research no larger datasets could be used.

5.1.1 Trainline EU

Trainline EU made timetable data available[2] suitable for the benchmarking of different algorithms. This data is based on the timetable of many Western-European countries in a 48-hour period.

The advantage of this data set is primarily its size. In total, it contains 4,714,403 connections stopping at over a 1067 places. This makes the timetable suitable to differentiate between how the different data structures used for implementation perform in differently sized scenarios.

The data set does not contain any footpath information or live updates. In benchmarks, the interchanges are seen as to be without any delay.

5.1.2 NDOV Loket

The NDOV Loket [9] is a part of OpenGeo which makes available public transportation data of the Netherlands, statically and live. For benchmarking, we have chosen

[2] https://github.com/trainline-eu/csa-challenge/blob/master/bench_data_48h.gz

the IFF database and its corresponding live data sources. This database contains all scheduled train trips which pass through the Netherlands.

These data are primarily focused on the testing in a realistic environment, however, its size is relatively small. For simplicity, we focused our benchmarks on data of the 15th of January 2021. In total this meant around 5,988 trips containing over 58,000 connections stopping at 1,311 stops placed in more than 600 stations. Next to that, around a week of live data was gathered resulting in 1,200,000 messages, which were converted into more than 88,000 updates to the timetable.

5.2 Algorithms

For the comparison of algorithms we have created an implementation the following algorithms which solve the *Earliest Arrival Problem*:

- Time-dependent graph algorithm [5]
- Connection Scan Algorithm [6]
- Round-Based Public Transit Routing [5]

For each array algorithm, we have adhered to the original description of the authors of the algorithm. However, none of the algorithms provided information about how best to adapt them to live environments, for this reason we have had to adapt the data structures as to be generally updateable.

theoretically compared different techniques in the section analysis. In that section, we note that using binary-tree sets was the most suitable which is why we have made an implementation using this structures as well.

For each algorithm examined, we discussed the expected impact of live data on the algorithm based on its constraints.

5.2.1 Static environment

Connection Scan Algorithm.

The Connection Scan Algorithm uses an ordered list of connections which is walked through for every routing request. In a static environment, an array structure can be used for this purpose, since ordering need to only take place once.

This algorithm thus has a complexity of $\mathcal{O}(|\mathcal{C}| + |\mathcal{F}|)$.

Time-dependent graph algorithm.

The time-dependent graph algorithm contains an ordered list of connections for every route stop a to b. In a static environment a double hash-map containing an ordered array structure can be used for this purpose since ordering need only take place once.

The algorithm uses Dijkstra, each node representing a stop and each set of connections between two stops representing the edges. When visiting a stop, a connection to each reachable next node is retrieved using binary search in the connections list of those two nodes.

This gives the algorithm a complexity of:

$$\Theta((\frac{|\mathcal{C}| \log |\mathcal{R}|}{|\mathcal{R}|} + |\mathcal{F}| + |\mathcal{S}|) \log |\mathcal{S}|)$$

Here $\mathcal{R}$ is defined as the set of two-tuples with the stops (s_{dep}, s_{arr}) such that:

$$\forall_{(s_{dep},s_{arr})\in\mathcal{R}} \ (s_{dep}, s_{arr}, x, y, z) \in \mathcal{C}$$

Round-Based Public Transit Routing.

RAPTOR is not primarily based around connections but around routes [5]. A route is a unique set of ordered stops associated with many trips. Each trip's route can be determined by the ordered list of stops of the connections of the trip. Each round, all routes which were discovered in the previous round are relaxed. The number of rounds the algorithm runs for is noted as K.

We define the set $\mathcal{R}$ be the set of all possible routes. Each trip in $\mathcal{T}$ is part of one unique route, and is stored within the route in order of departure.

Next to the routes, the algorithm also keeps track of a table to lookup which routes depart and arrive at each stop.

RAPTOR has a complexity of $\mathcal{O}(K(\sum_{r\in\mathcal{R}} |r| + |\mathcal{T}| + |\mathcal{F}|))$

5.2.2 Real-time environment

In the real-time environment each algorithm should support the deletion, addition and update of any connection.

Since none of the data structures, as discussed by the original authors of the algorithm, allow for this we have had to adapt these algorithms ourselves. We note that the algorithms need to maintain an ordered list of either the connections or the trips for use in the algorithm.

In Table 2 an overview is given of the different types of collections which can be used for this purpose and the associated complexity. Looking through this we note that while a *hash-set* collection has the lowest complexity in all update categories, it is significantly worse on the iteration speed since its internal state cannot be sorted.

Therefore, the best collection type to use seems to be the binary-tree set. The binary-set stores data in a tree like manner which always stays sorted internally. This allows it to quickly find any item and insert any item while still having a linear iteration complexity.

For each algorithm we have implemented a version with the original data structures (Vec) as described and one which uses binary-trees (BTree) and support the updating mechanism.

Below we describe how we have implemented the updating process for each algorithm. Again, no mechanism for this was described by the original authors.

Connection Scan Algorithm.

Received updates are immediately applied on the internal data structure containing an ordered list of all connections.

Time-dependent graph algorithm.

The Time-dependent algorithm uses an ordered 2d hash-set to store all connections in a binary-tree set for every route in $\mathcal{R}$. To update, the route can be looked up, after which the connection can be updated.

Round-Based Public Transit Routing.

RAPTOR is more difficult to implement in a live system since trips also need to support their route being changed. To make this feasible we introduced a lookup hash-map which can be used to find the position of each route in the route list as described in [5]. With each update, we lookup the old route, remove the trip and later insert the updated trip in the new route.

5.3 Benchmarking

Collection	*Insertion*	*Deletion*	*Update*	*Iteration*
Array	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Linked List	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Hash-set	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$
Binary Tree Set	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Table 2. Comparison of different collection operations on sorted data. n is the number of items in the list

A benchmarking suite was created to create an interface which allowed for the easy implementation and benchmarking of a vast set of different routing algorithms.

The benchmarking program currently has the following functionality:

- Retrieve and interpret IFF timetable data (NDOV)
- Retrieve and interpret InfoPlus RIT live data (NDOV)
- Simple routing of timetable from stop a to b
- Benchmarking with options for a static and real-time environment

The main element is the benchmark of algorithms, during the benchmark the following data is recorded:

- Dataset
- Algorithm
- If performed under live conditions
- Execution duration
- Distance between stops (when available)
- Query answer

Next to the query time we also measure the time it takes to applies updates.

The algorithm to benchmark static algorithms can be found in algorithm 1. The algorithm for our real-time environment can be found in algorithm 2. Our real-time environment tries to simulate the handling of both incoming updates and queries which mimics the events in a real-life system.

Algorithm 1: Benchmark of static algorithms

Data: timetable: $(\mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{F})$, algorithm which can solve the Earliest Arrival Problem
Result: list of query benchmark results
algorithm ← algorithm(timetable);
results ← empty vector;
for $s_1 \in$ *first hundred stops of* $\mathcal{S}$ **do**
  for $s_2 \in$ *next hundred stops of* $\mathcal{S}$ **do**
    start time ← system time now;
    algorithm.earliest_arrival(s_1, s_2, at mid day);
    end time ← system time now;
    results.push((start time - end time, s_1, s_2));
  end
end

5.4 Data retrieved

We do not note down the initialization time for any of the algorithms being tested on since this is dependent on the given data structure. The data structure can be designed in such a way as to give an advantage to certain algorithms which would give an unfair comparison.

Algorithm 2: Benchmark of live-algorithms with real-time environment

Data: timetable: $(\mathcal{S}, \mathcal{C}, \mathcal{T}, \mathcal{F})$, algorithm which can solve the Earliest Arrival Problem and receive updates, list of update chunks
Result: list of query benchmark results and update benchmark results
algorithm ← algorithm(timetable);
query results ← empty vector;
update results ← empty vector;
normalizer ← $\min(|\mathcal{S}|, 100)^2$
for $s_1 \in$ *first hundred stops of* $\mathcal{S}$ **do**
  for $s_2 \in$ *next hundred stops of* $\mathcal{S}$ **do**
    start time ← system time now;
    algorithm.earliest_arrival(s_1, s_2, at mid day);
    end time ← system time now;
    query results.push((start time - end time, s_1, s_2));
    for *normalize times* **do**
      for *update* $\in$ *next update chunk* **do**
        start time ← system time now;
        algorithm.update(update);
        end time ← system time now;
        update results.push(start time - end time);
      end
    end
  end
end

5.4.1 Benching information

All benchmarks were performed on a laptop containing an Intel(R) Core(TM) i7-6700HQ processor clocked at 2.60GHz and 16 GiB of memory. No compiler optimizations were used next to the standard release profile of the Rust compiler. The benchmark program was compiled with `rustc` version 1.48.0.

5.5 Benchmark results

All benchmarking results can be found in Figures 2 to 5.

Firstly, we have plotted the box plots of our algorithms for the trainline and NDOV dataset in a static environment in figure 2 and 3. This primarily gives us a look into how the algorithms compare when given more or less connections.

Next, we show two plots containing the execution times in the live environment as described in algorithm 2. Figure 4 shows the execution time of the same queries as performed in the static benchmark, while figure 5 shows the apply time for update per algorithm.

Please note that only data is shown for queries that returned a non-empty response since empty responses usually have a similar execution time.

A summary of the results can be found in table 3.

6. DISCUSSION

There are a few different things to look at, starting with

algorithm	dataset	live	avg. query time (ms)	avg. update time (μs)
CSA with Vec	NDOV	○	0,60	-
CSA with BTree	NDOV	○	0,71	-
CSA with BTree	NDOV	●	0,75	0,90
TD with Vec	NDOV	○	0,20	-
TD with BTree	NDOV	○	0,22	-
TD with BTree	NDOV	●	0,22	0,43
RAPTOR with Vec	NDOV	○	6,86	-
RAPTOR with BTree	NDOV	○	6,98	-
RAPTOR with BTree	NDOV	●	6,99	1,93
CSA with Vec	Trainline	○	26,35	-
CSA with BTree	Trainline	○	15,39	-
TD with Vec	Trainline	○	7,41	-
TD with BTree	Trainline	○	13,31	-
RAPTOR with Vec	Trainline	○	16,42	-
RAPTOR with BTree	Trainline	○	11,29	-

Table 3. Summary of benchmark results for the different algorithms

the differences between the Vec implementation and the BTree implementation. Within our NDOV dataset we see a clear difference between the two. For each algorithm the BTree implementation is slightly slower. This is to be expected as one big downside to the binary-tree set is the CPU optimizability. With the way the data is structured, the structure cannot take use of CPU caching improvements which means that reading from the list can be significantly slower.

However, the inverse is true for the Trainline dataset for the CSA algorithm. In general one should not expect better execution times for the BTree version since the exact same data is being iterated through. One possible explanation is that the BTree implementation is better optimized to find the first connection that needs to be checked, however that does not explain the full difference.

Next, we have kept track of the query times for each algorithm in the static and real-time environment. No real query time differences can be found which indicates that the amount of cache misses due to updates taking in the data is insignificant.

Finally we can compare the updating times. From our testing we found that the Time-dependent algorithm clearly beats all other algorithms in updating time. TD keeps track of connections per stations whereas CSA only keeps track of one big list of connections. This makes it quicker to insert, update or remove data from the connections list. The clear loser is the RAPTOR implementation. RAPTOR keeps track of many routes instead of connections which means that the corresponding route needs to be found which can be an intensive task.

7. CONCLUSION

In this paper we have tested three different algorithms in a static and live environment. With our implementation and analysis steps we have shown the difficulties different algorithms have with coping with real-time environment data and how these could be solved. With our benchmarking we were able to show what impact these features had on each algorithm.

We have found that overall the TD implementation performed best, both in the static as in the live-environment. This is primarily caused by its efficient manner of splitting connections which meant lower updating times.

However, it is difficult to conclude that this algorithm is best in all situations. Each algorithm is focused on solving the same task while keeping different things in mind. For instance RAPTOR which has had better adaptability for multi-criteria queries in mind. Overall, we can conclude that while there is definitely an impact to adapting a public-transport routing algorithm to work in a real-time environment, all algorithms tested in this paper were able to adapt well with minimal impact on their query times.

8. FUTURE WORK

This research created a start to a toolset to benchmark different public transport routing algorithms, however due to the constraints set, we were only able to take a look at a limited number of different algorithms using only a limited set of data. Many of the systems that could be used to retrieve the necessary data are proprietary and baldy documented, which means that every source of data needs its own translation layer, making realistic environments hard to recreate in such a short time.

Another area which could be more focused on is having the benchmarking run with more realistic queries. Currently, all queries are randomly generated, and only generated for a single day of the timetable. By expanding the timetable to cover a greater period, the algorithms need to cope under harsher but more realistic conditions, where correct linkage between timetable days, and efficient memory storage of connections need to be thought of. Next to that, looking at making the queries more realistic would significantly improve the real-world application of the gathered data.

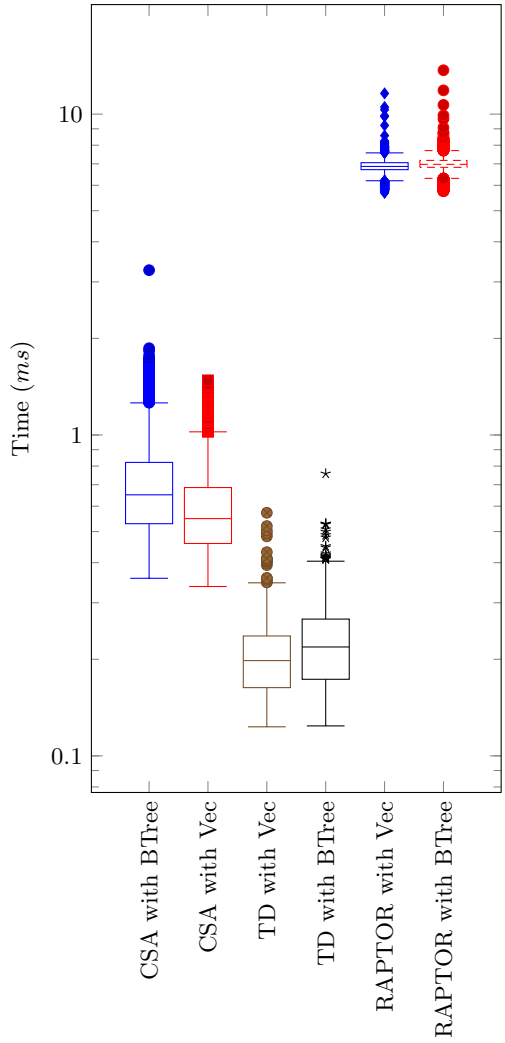

Figure 2. Execution time for queries per algorithm for the NDOV dataset in the static environment

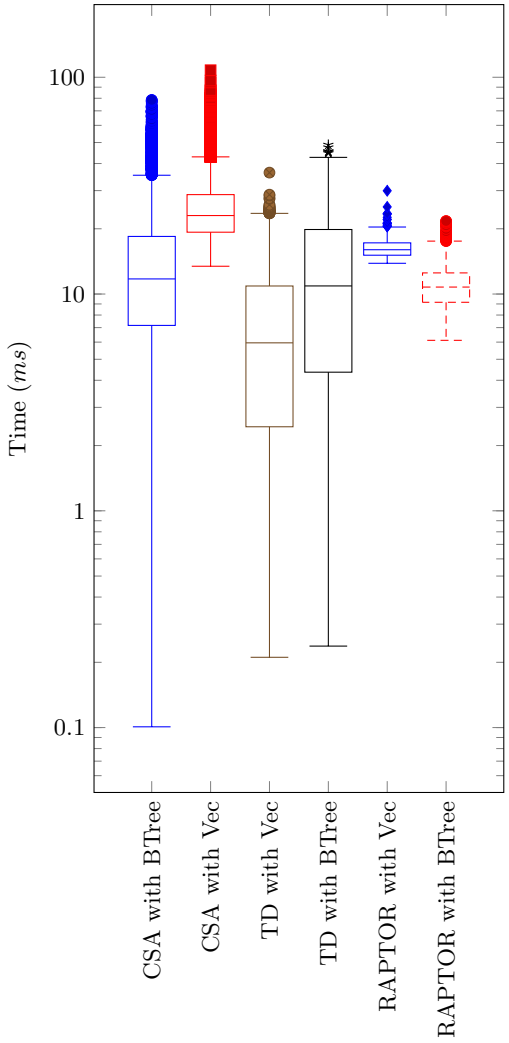

Figure 3. Execution time for queries per algorithm for the Trainline EU dataset in the static environment

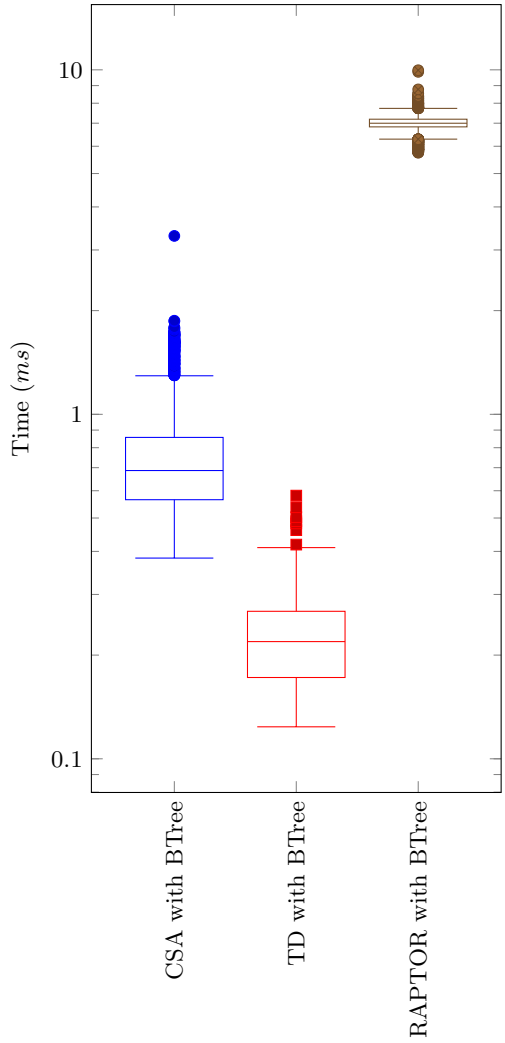

Figure 4. Execution time for queries per algorithm for the NDOV dataset in the live environment

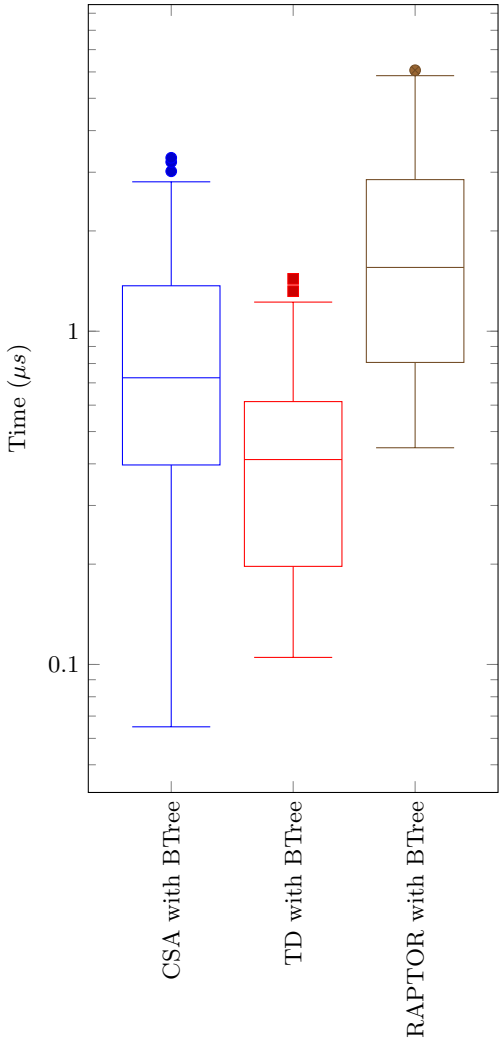

Figure 5. Execution time for updates performed per algorithm

9. REFERENCES

[1] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger. Fast routing in very large public transportation networks using transfer patterns. In M. de Berg and U. Meyer, editors, *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.

[2] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015.

[3] H. Bast and S. Storandt. Frequency-based search for public transit. In Y. Huang, M. Schneider, M. Gertz, J. Krumm, and J. Sankaranarayanan, editors, *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, pages 13–22. ACM, 2014.

[4] D. Delling, J. Dibbelt, and T. Pajor. Fast and exact public transit routing with restricted pareto sets. In S. G. Kobourov and H. Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 54–65. SIAM, 2019.

[5] D. Delling, T. Pajor, and R. F. F. Werneck. Round-based public transit routing. In D. A. Bader and P. Mutzel, editors, *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012, The Westin Miyako, Kyoto, Japan, January 16, 2012*, pages 130–140. SIAM / Omnipress, 2012.

[6] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. Connection scan algorithm. *ACM J. Exp. Algorithmics*, 23, 2018.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] E. Pyrga, F. Schulz, D. Wagner, and C. D. Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM J. Exp. Algorithmics*, 12:2.4:1–2.4:39, 2007.

[9] Stichting OpenGeo. Dutch real-time transit data, 2013-2019.

[10] Universität Freiburg. Google focused research award on next-generation route planning, 2015.