Journal of Information Systems and Telecommunication
Vol.11, No.3, July-September 2023, 260-268
http://jist.acecr.org
ISSN 2322-1437 / EISSN:2345-2773

Trip Timing Algorithm for GTFS Data with Redis Model to Improve the Performance

Mustafa Alzaidi[1*], Aniko Vagner[2]

[1].Department of Information Technology Faculty of Informatics University of Debrecen, Hungary
[2].Department of Information Technology Faculty of Informatics University of Debrecen, Hungary

Received: 05 May 2022/ Revised: 04 Dec 2022/ Accepted: 09 Jan 2023

Abstract

Accessing public transport plays an essential role in the daily life productivity of people in urban regions. Therefore, it is necessary to represent the spatiotemporal diversity of transit services to evaluate public transit accessibility appropriately. That can be accomplished by determining the shortest path or shortest travel time trip plan. Many applications like ArcGIS provide tools to estimate the trip time using GTFS data. They can perform well in finding travel time. Still, they can be computationally inefficient and impractical with increasing the data dimensions like searching all day time or in case of huge data. Some research proposed recently provides more computationally efficient algorithms to solve the problem. This paper presents a new algorithm to find the timing information for a trip plan between two start and destination points. Also, we introduce RMH (Range Mapping Hash) as a new approach using Redis NoSQL to find and calculate the accessibility of a trip plan with fixed time complexity of O(2) regardless of the city size (GTFS size). We experimented with the performance of this approach and compared it with the traditional run-time algorithm using GTFS data of Debrecen and Budapest. This Redis model can be applied to similar problems where input can be divided into ranges with the same output.

Keywords: Author Guide; Article; Camera-Ready Format; Paper Specifications; Paper Submission.

1- Introduction

For two key reasons, public transportation accessibility has become a hot topic for scholars and transit organizations. First, improved transit accessibility promotes active transportation (such as walking and bicycling) while decreasing private vehicle use. As a result, it will enhance public health and reduce GHG emissions [1]–[4]. Second, transit-dependent people primarily rely on public transportation to reach key services (e.g., workplace, university, and shopping center). Thus, transit accessibility is crucial to attaining socioeconomic fairness[5], [6]. Also, transit accessibility research may help guide decisions about transportation investment and land use development[7].

GTFS stands for General Transit Feed Specification. It was developed by TriMet and Google in Portland [8]. Google announced transit feed specs in 2007, enabling transit agencies to develop and publish transit data online as open sources using the GTFS format. The feed rapidly became the most extensively used standard for exchanging static transit data in Canada and the United States [9] [10]. Additionally, the transit sector has embraced the GTFS format as a standard for communicating schedule data due to its expanding popularity. Subsequently, software such as OpenTripPlanner, GoogleMaps, and Bing Maps was developed and updated to use GTFS and provide services like, stop locations, timetables, and route planning. The trip (or route) planner is essential to these applications since it examines GTFS data for possible routes between two places, which is the most demanded service.

Handling a user trip planning request requires two actions or steps. First, identifying all feasible pathways or routes between two places as candidate solutions; second, filtering and validating these solutions according to the user's schedule and start time. Route planning is more complex than identifying a path or route in a graph since it considers journeys, directions, and intermediate transfers between bus stops or stations. The shortest path of a graph is a frequent issue; various algorithms have been developed to solve it [11]–[14]. Depending on the factors used to calculate the weight of graph edges, the algorithm may be bi- or multi-criteria. For instance, if the graph edges depict roads, the road weight may be a bi-criterion, taking distance and cost into account, or a multi-criterion,

✉ **Mustafa Alzaidi**
mustafa.alzaidi@inf.unideb.hu

taking other parameters into account. Numerous techniques attempt to solve the shortest path issue by decreasing the considered factors to a single value; these algorithms fall into types like two-phases algorithms [15], kth shortest-path [16], label correction, and setting algorithms[17]–[23], and others [23]–[28]. A subset of these techniques is used in certain studies [29][30] to locate pathways in local transportation networks. We have already introduced a trip planning algorithm variation[31] this algorithm ignores the weight criteria while checking for all possible next transitions from the current and uses a limit for the number of made transitions to find the best route (trip plan). This trip planning algorithm can find the possible trip plans without considering the timing factor. However, some plans may be rejected due to time conflict between the plan trips and the GTFS trips timetable or trip unavailability at the time the user determines to start the journey. Therefore, we must calculate and find the trips time and transit accessibility as a next step.

To calculate transit accessibility in spatiotemporal dimensions, trip time for station pairs must be calculated at any particular time of day, which is practically impossible with a standard computer as it is time-consuming and needs high computation power[32]Although previous studies [33] introduce algorithms that try to calculate the trip time and transit accessibility while reducing the time complexity and computational power, there is still a need to find an approach to simplify the complexity of such problems solution and reduce the required time and resources, what is the aim of this paper. Algorithms enhancement is a common research topic[34]–[36]. The contribution of this paper has two parts. First, we introduce a new time validation algorithm that can find the timing information for a trip plan or reject the plan if there is a time conflict according to the trips timetable in the GTFS. Second, we go beyond the algorithm enhancement and propose RMH (Range Mapping Hash), which is a new method that can find and extract the timing information for any trip using GTFS data with O(2) time complexity. Our new approach (RMH) eliminates the need for an algorithm to search the GTFS timing records. We use Redis NoSQL Hash to create RMH. Thus we provide a solution by turning the problem of simplifying the existing algorithms into a simple database interaction that can run even on a stander computer. The idea is that for a route going through a station, at any minute between the last going bus and the next bus, the answer for the question "when is the next bus time" will be the time of the next bus. The RMH is applicable not only for the GTFS timing data but also for improving the performance of similar problems, as we describe later. We experiment with the performance of RMH and compare it to run-time search algorithm performance using arbitrary search input for 30 pairs of origin and destination stops using the GTFS data of Debrecen and Budapest. We implement the algorithm and RMH as an open-source project using C# and Redis available on https://github.com/mustafamajid/GTFS-csharp. The project also includes our published trip planning algorithm[31].

Next, we review our route planning algorithm and its output data structure[31]. Then we introduce the time validation algorithm, which will use this output to provide the trip's timing information. Later, we introduce the RMH approach and the Redis implementation. Finally, we list our experiment's results and performance evaluation.

2- GTFS and Trip Planning

The GTFS data is a set of tables, usually in CSV file format. There are three main objects in the GTFS: route, trips, and stop [37]. The routes represent the pathway used by the vehicle, a bus, tram, train, etc., and are usually denoted by the vehicle name. The route visits a set of stops in a specific sequence where the stop can be a bus or tram stop or a train or subway station.

Planning a trip between two locations (the start and destination) requires finding all possible single or combinations of trips that can take the passenger from start to destination points. Finding trip plans can be divided into two steps. First, find all possible routes that can connect the start to the destination point, and these will be the candidate solutions list. Then find the trip's timing information and check for any time conflict in the candidate's plans. To understand the problem, we use Figure 1 as an example of the GTFS data. The figure shows three routes, A, B, and C going through a set of stops denoted by circles with numbers. We consider that the user wants to start the trip from stop 7 at 11:10:00 going to stop 6. Therefore, the candidate solutions will be as follows: first, the user takes route C to stop 9 and then takes route A from stop 9 to stop 6. The second solution is that the user walks from stop 7 to stop 3 and then takes route B to stop 6 if the distance is walkable [38]. The next step is to validate the solution according to the timetable. For the first solution, as in the figure, if the user starts at 11:07:00, 11:17:00, or 11:27:00, he will arrive to stop nine at 11:35:00, 11:35:00, or 11:45:00, respectively. Thus, the user will take the trip at 11:17:00 because it is the earlier trip and then arrive at stop 9 at 11:35:00, where the next trip using route A will be at 11:45:00 and reach the destination at 12:05:00. In the same way, we can find the time information for the second solution. As we mentioned earlier, a solution can be rejected if there is a time conflict; for example, the first solution may be rejected if there is no outgoing trip using route A from stop 9 any time after 11:35:00.

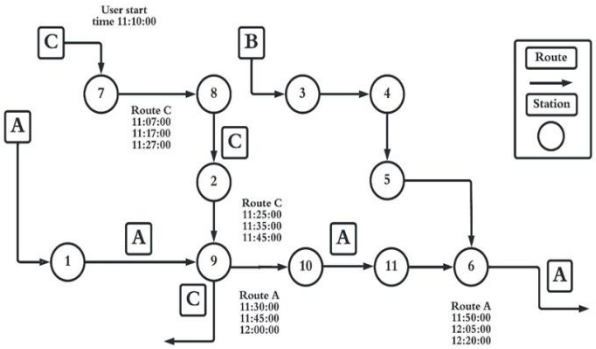

Fig 1. GTFS routes and stops example.

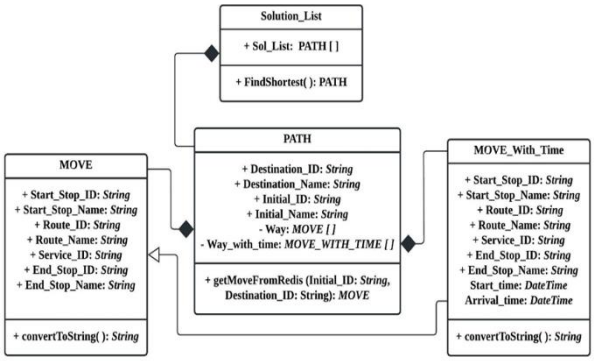

Fig 2. Algorithm data structure.

3- Find Trip Time Information

3-1- Data Structure

Finding trip plans according to the time is a complex problem and needs computation power that is not provided by standard computers [33]. For this work, we use the output structure (the candidate solutions) provided by our previously proposed trip routes planning algorithm [31] as an input to introduce our new trip timing algorithm. Figure 2 shows the UML design of the trip routes planning algorithm output with additional fields to store the time data. The structure contains three main objects Solution_LIST, PATH, and MOVE. Each MOVE represents a single transition from a start-stop to an end-stop using a route (e.g., a bus), and PATH denotes a trip plan or solution containing at least one or more transitions (MOVE) stored in a list called Way. The algorithm's final output is a list of PATH called the Solution_List. The MOVE_WITH_TIME class was inherited from the MOVE class and contained the arrival and departure time fields. Finally, a list of MOVE_WITH_TIME is added to the PATH class called Way_With_Time. The task of the next trip timing algorithm is to validate the PATH by checking every MOVE object in its Way list. If time conflict is found in any MOVE, the whole PATH will be rejected; Otherwise, a new MOVE_WITH_TIME object will be created from the current MOVE by adding the timing fields. The newly created list of MOVE_WITH_TIME objects will form the Way_With_Time list.

3-2- Algorithm

The stoptimes.txt file list a set of records for each trip; each record contains stop ID, trip ID, trip arrival, and departure time at that stops. The set of trips records is present in the file ordered by trip ID and the arrival time. Thus, if the file starts to list a trip that visits ten stops at row number N, then the row N contains timing data about the first stop, row N + 9 shows the data about the last stop that the trip visits, and row N + 10 will list data for the first stop of another new trip if any. Every PATH must be checked by examining the MOVEs in its WAY list using T's time. The stoptimes.txt file record is checked sequentially to find the trip with the closest time to T. Initially, T is set to the time determined by the user (USER_TIME) to start the trip, and during the next MOVEs check, T is set to the arrival time at the last checked MOVE end stop. The check starts from the first record in the stoptimes.txt until finding the first record (i) with stop_id equal to the MOVE start_stop_id and with the same route used by the MOVE and the departure time is greater than T and one of the next record (i + j) in the same trip with stop_id equal to the MOVE end_stop_id. Where j is the number of intermediate stops, if such records are found, a MOVE_WITH_TIME object is created using the examined MOVE and record (i) departure time as Start_time and record (i + j) arrival_time as arrive time for the new MOVE_WITH_TIME object and as the new T value for the next MOVE check. If no such record is found in the stoptimes.txt file, then the whole PATH is rejected and mentioned as an unacceptable solution. The new resulting MOVE_WITH_TIME objects are used to form a WAY_WITH_TIME list. Figure 3 shows the Trim timing algorithm that validates the MOVE according to the trip's timing information. The algorithm input is the start-stop from which the MOVE starts, the end-stop where the MOVE ends, the route used to make that MOVE, and the user's time. The algorithm output must be the trip on that route with the nearest time to T.

Input: WAY a List of MOVE, USER_TIME.

OutPut: WAY_WITH_TIME as List of MOVE_WITH_TIME objects

Step1: T=USER_TIME , WAY_WITH_TIME = *empty*

Step2: ForEach MOVE M in WAY

Do **Step4** To **Step5**

Step3: Set M_WITH_TIME =*NULL,*

Step4: For (i=0 ; i< Stoptimes.Length-2 ; i++**)**

IF(Stoptimes[i].stop_id == M.start_stop_id **&&** Stoptimes[i].Route == M.route_id **&&** Stoptimes[i].Departure_time >T **) Then:**

For (j=i+1 ; i< Stoptimes.Length-1 ; j++**)**

IF (Stoptimes[j].stop_id == M. end_stop_id)

M_WITH_TIME = MOVE_WITH_TIME (M,Stoptimes[i].Derparture_time Stoptimes[j].Arraival_time).

ADD M_WITH_TIME **to** WAY_WITH_TIME,

T= Stoptimes[j].Arrival_time

Goto **Step2** check the next MOVE

Step5: IF M_WITH_TIME ==NULL **Then:**

Return FALS and Exit **Else**

Step6: Return WAY_WITH_TIME

Fig 3. Trip timing algorithm.

3-3- Time Complexity

Searching the stoptimes.txt file record is a time-consuming process as any linear search, the algorithm performs a linear search for timing information. Let N is the number of records present in the stoptimes file. Then, the best case is if the stop with time greater than T is at the stoptimes file's first record. The worst case is O (N-1), and the average case is O((N-1)/2). Thus, the algorithm time is increased by increasing the number of records. We ignore the number of records between the start and the end stops, as this number is minimal compared to N.

4- Redis

Redis is a high-performance in-memory NoSQL database written with C and worked on most POSIX platforms [39]. Redis is a message broker and session manager that stores data in key-value pairs. An HTML page including its resources may be serialized to a string and saved in a Redis to enable a high-speed page load. Thus, software organizations prefer Redis for its fast performance and scalability. Strings, Lists, Sets, Hashes, and Sorted Sets are the five data structures available in Redis. In this research, we will depend only on the Hash structure. All our object data will be converted to a string and concatenated before being stored in the Redis Hashes. A wide range of programming languages supports Redis. Each language has its libraries and packages for communicating with and manipulating the Redis server. In this project, we utilized StackExchange.Redis, which can be installed using NuGet Package Manager.

5- Range Mapping Hash (RMH)

We propose the RMH as a Redis model to avoid the time-consuming liner data scanning by mapping the input parameters to the output directly without any liner search or scan using the power of hash structure in Redis. For each route between any two stops, we need to map a route and two stops ID and time T to the ID of the trip with the nearest time to T on that route, the trip departure time at the start-stop, and trip arrival time at the end stop. The RMH consists of two structures. Both structures querying results are combined to form the timing answer. We use a Redis Hash structure for the implementation. The Hash structure syntax has three-part, the KEY, which refers to the Hash name, the FIELD that uniquely identifies a row in the hash; and the VALUE. The HGET and HSET commands are used to retrieve and insert data into Redis Hash [39].

5-1- RMH Trip Departure Time Structure

The first structure is used to map the start-stop ID, route ID, and Time (T) into the next trip's ID and time at this stop using that route. For this structure, we create a Redis hash for each stop route pair to store all trip visiting time at the stop. The Hash KEY part will mention the stop ID and the route ID separated by the "__" string. The FIELD will contain the time to be examined and denote it as T. The hash VALUE part holds the time of the next coming trip according to T, and the trip ID, separated by the "__" string. Figure 4 shows the trip time structure.

All this information is available in the stoptimes file except the time (T). Any possible T value belongs to the set of sharp minutes in the day. Thus a maximum of 1440 entries is needed to cover all the possibilities. For each stop ID, route ID pair from the stoptimes data, and a time T entry, the value field contains the time of the next coming trip and the trip ID separated by "___". For example, in Figure 4, we take route 10 and stop 1100905. Three trips, 208,209, and 210 are listed in the stoptimes file with departure times 11:44:00, 11:54:00, and 12:04:00. At the first hash entry, where the T value is 11:43:00, the answer (the Hash value field) shows the time 11:44:00 and trip ID 208. For any value of T equal to or greater than 11:44:00, the answer will be trip 209 as its time is 11:54:00. When T is greater or equal to 11:54:00, the answer will be trip 210 and its time 12:04:00.

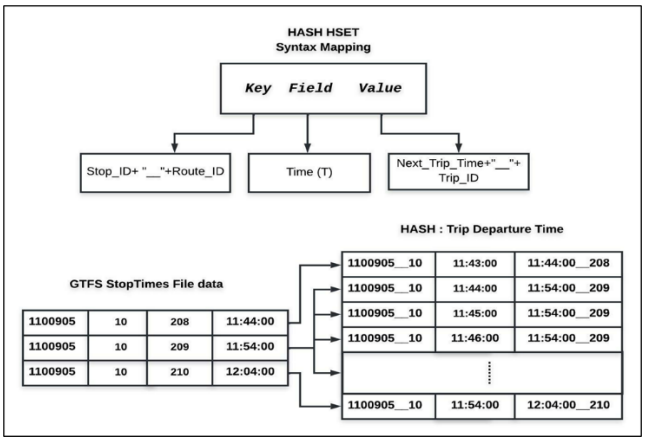

Fig 4. RMH trip departure time structure.

5-2- RHM Arrival Time Structure

After using the trip departure time structure, we have the departure time from the start-stop and the trip ID. Finally, we can find the end stop's arrival time using the end-stop ID and the trip ID using the arrival time structure. Figure 5 shows the arrival time structure, a single Redis Hash for each stop, to list all the trip arrival time combinations. The KEY part of the Redis Hash is used to refer to stop using the stop ID; the FIELD is used to refer to the trip ID, where the VALUE stores the arrival time.

For example, in Figure 5, the trip planning algorithm results in a MOVE with start-stop 1100905, end stop 1002315, and route 10. If the T value is 11:45:00, the timing validation will work as follow: HGET statement using the trip departure time structure is used to retrieve the trip ID and the departure time from the start-stop. The HGET KEY will be "1100905__10" (start-stop ID and the route ID); the FIELD part is "11:45:00" (T). the returned value from this HGET statement will be "11:54:00__209" (the departure time and the trip ID), as shown in figure 8. Now, the trip is known, and the end stop's arrival time is the only missing part. Another HGET statement with KEY is "1002315", and FIELD "209" is used with the arrival time structure; the return value from this statement will be "12:02:00 (the arrival time at the end stop) as shown in Figure 9. If any of the two structures did not return a match for the HGET command, the MOVE is rejected and the whole PATH (path). Thus the total time complexity of RMH formed by two Hash table read operations only. As the time complexity for reading from a Hash structure is O(1)[40], then RMH total complexity is O(2).

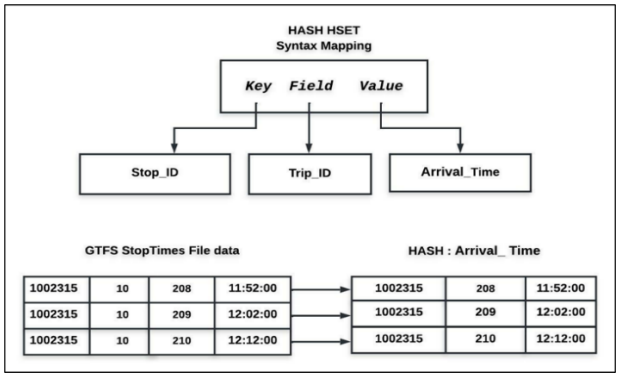

Fig 5. RHM arrival time structure.

6- Experiment and Results

6-1- Experiment Tool

We implement the trip timing algorithm as a C# project. The project is a WinForm application (GUI) containing a set of classes: GTFSData for loading and preprocessing the GTFS data, Algorithm class contains the implementation of our previously published trip planning algorithm, TimeCalculator class contains the implementation of the trip timing algorithm (trip timing), Redis action class include the code for connecting to Redis database load the data to Redis and retrieve the solution and other classes. Figure 6 shows the UML design of the main classes in the project.

We used the GetMovesWithTime( ) function from the TimeCalculator class for this experiment, which takes a MOVE list and start-time as a parameter and returns a MOVE_WITH_TIME list. We calculate the execution time for this function and compare it with the execution time for reviving the timing information using the RMH using the GetSolFromRedis( ) function from the RedisAction class.
The GetMovesWithTime() illustrates the implementation of the trip timing algorithm given earlier, whereas the GetSolFromRedis() represents the interaction with the Redis RMH model, which is implemented as mentioned before. This function will retrive the same result returned by the GetMovesWithTime() function (for the same parameters). The Redis RMH serves in a similar way to a data warehouse model where redundant data is stored and utilized to serve the application purpose.

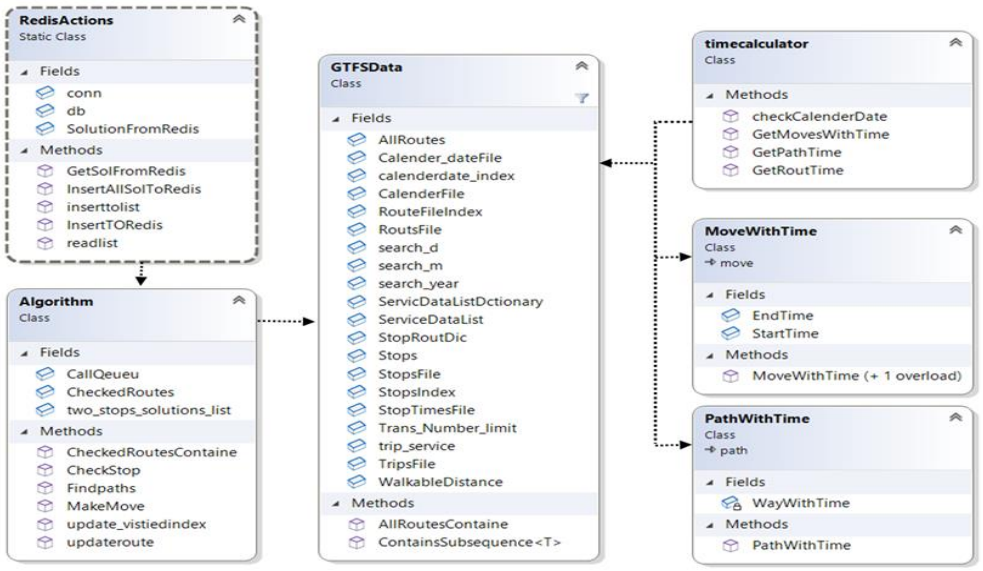

Fig 6.Test tool UML design.

6-2- Result

We experiment using RMH (Redis model) with Budapest and Debrecen cities GTFS data for 30 random start and end stop combinations. Each experiment finds the timing data with and without using Redis (the RMH) and records the time (in milliseconds) that the computer takes to retrieve the result for each combination. Table 1 shows the recorded results.

Table 1. Experiments results.

NO	Budapest Without Redis	Budapest Using Redis	Debrecen Without Redis	Debrecen Using Redis
1	1202.4	8.368	1204.6	8.559
2	1305.4	8.238	1106.7	8.278
3	1105.2	8.38	908.2	8.48
4	1409.3	8.749	907.3	7.237
5	1408.4	8.198	904.1	8.107
6	1206.3	8.558	1008.1	7.848
7	1201.9	7.387	906.7	8.298
8	1106.3	7.938	909.4	6.897
9	1002.6	8.638	907.5	8.318
10	1402.3	8.727	906.3	8.278
11	1404.2	8.648	1009.8	8.67
12	1103.2	7.018	908.7	6.917
13	1107.1	7.449	1105.7	7.53
14	1303.4	7.418	1009.2	7.227
15	1324.1	8.027	1205.4	7.887
16	909.7	8.199	1204.2	8.509
17	1213	8.73	1004.5	7.488
18	908.5	8.319	1003.3	6.809
20	1308.7	7.679	1202.9	8.037
22	1408.4	6.829	1103.2	7.748
23	1203.8	6.839	1208.9	7.05
24	1304.8	8.307	1009.1	7.807
25	1303.5	8.678	907.7	6.909
26	1204.3	8.098	908.5	7.677
27	1305.1	7.099	1101.8	8.3
28	1001.1	7.697	901.8	8.437
29	1003.9	7.158	1001.2	8.047
30	1421.4	7.758	1202.4	6.849
Average	1219.89	7.985	1025.94	7.808

Figure 7 visualize the execution time difference between the time taken for finding the timing information for a trip using the run-time algorithm without Redis and the execution time for retrieving the exact data for the same pair of start and end stops using Redis. We can notice that the run-time performance varies during the experiments, around an average of 1219.89 milliseconds. Conversely, Redis's execution time is more stable. It has ignorable variation during the experiments, with an average of 7.985

milliseconds forming a straight line in the chart close to zero compared to the run-time performance.

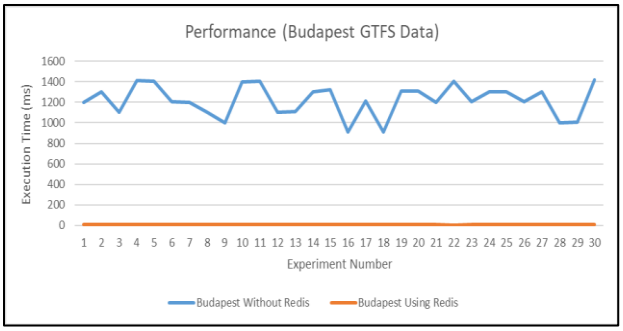

Fig 7.Budapest data experiments result.

With Debrecen data, the experiment shows a similar performance compared to Budapest experiments. The average execution time is 7.808 milliseconds which is very close to the execution time for Redis with Budapest data. However, again, the experiments show notifiable variation in the performance using the run-time algorithm. Figure 8 shows Debrecen data experiments' performance using Redis and the run-time algorithm (without Redis).

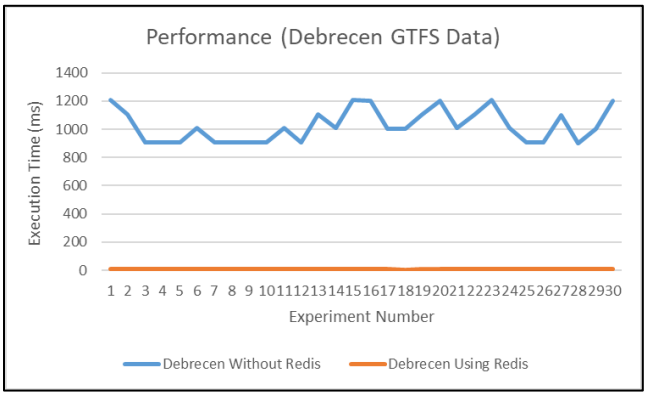

Fig 8.Debrecen experiments result.

The experiments also show that the difference between GTFS data size for the cities (Debrecen 1483 KB and Budapest 42128 KB) affects the performance in the case of run-time algorithm use. This effect can be clear if we compare the average execution time with both cities' data, as shown in Figure 9. We can notice that the average run-time execution time increases with larger cities (in this case, Budapest), while the data size has no effect in the case of using Redis.

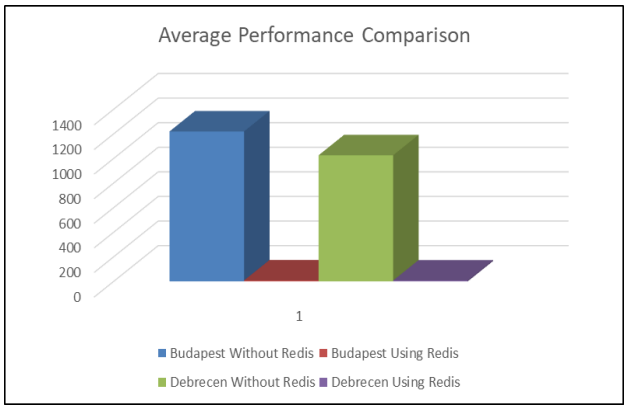

Fig 9.Experiments results average comparison.

7- Conclusions

In contemporary and Smart Cities, sharing transportation data is crucial for a successful transportation system. As a result, the necessity for a uniform format for communicating transportation data has grown. Transportation authority's extensively use GTFS (General Transit Feed Specification) across the globe as a standard format for sharing and publishing data. In addition, trip planning and computing transit accessibility are common topics between researchers and transit organizations as they can affect society's life and productivity. However, computing transit accessibility and finding a trip plan with timing information is complex and requires more computation than a standard computer can provide.

Find a trip plan and transit accessibility consist of two steps. First, find all possible routes that can lead from the start to the destination, mark them as candidate solutions, and then validate them according to the user time to start the trip(start time) and the trip timetable in the GTFS. The first part is done using the trip route planning algorithm, and the second part is accomplished using the trip timing algorithm. Also, they can be combined in one algorithm. This work uses our already published trip planning algorithm output as input to introduce a new trip timing algorithm. The trip planning algorithm output is a set of trip plans; each has one or more transitions. The trip timing algorithm in this paper validates these transitions by searching the trips to find the trip with the closest departure time to the time T (where T initially is specified by the user to start the trip) on the specified route.

Some researchers try to improve the time complexity of trip planning and trip timing algorithms. We introduced the Range Mapping Hash RMH as a Redis model that provides fast access to the timing data and eliminates the need to run the trip timing algorithm as it does the same task with better performance.

The model contains two structures. The first structure can map any route ID, start-stop ID, and the time T; to the next going trip's ID, and the departure time at the start-stop. The idea behind this method is that T can be any time during the day with sharp minutes part. Thus, we have 1440 possible values for T during the day. We use a Redis hash for the implementation. For each stop route combination, we create a hash with 1440 entries such that the key part will mention the stop ID and the route ID separated by the string "__", the hash fields will hold the T possible values, and the value field will hold the next trip_ID and its departure time. If T falls between two trips' departure times, then the answer (the value field) should be the trip with a later departure time. If T is earlier than the departure time of the first trip, then the answer will be the first trip and its departure time. If T is later than the departure time of the last trip in the GTFS data, then no entry will be stored in the hash, and a null value will be returned for such search, leading to rejecting the transition and the plan, and in this case, we will have less than 1440 entries in the hash list. The second structure is a Redis hash with the key part holding the destination stop ID, the field part containing the trip ID goes through that stop, and the value part containing the arrival time. Thus, both structures can form answers for any trip timing request. Both structures are Redis hash, and each can provide the response within O(1) complexity. Thus, the RMH can solve the timing problem with (2) complexity.

Using GTFS data from Budapest and Debrecen, we tested the performance of RMH and the normal trip planning algorithm using the same computation hardware and software specifications. Experiments show that RHM can provide better complexity than the time validation algorithm. The experiments also show that RHM provides consistent time independent of data size (city size) in comparison to the run-time algorithm, where the performance is decreased when the data size is increased. In future work, the RMH can be applied to any similar problem where the input can be divided into sets or ranges with identical output. The RMH sacrifices the space to provide better performance.

Acknowledgment

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.

References

[1] T. Litman, “Integrating Public Health Objectives in Transportation Decision-Making,” *American Journal of Health Promotion*, vol. 18, no. 1, pp. 103–108, 2003, doi: 10.4278/0890-1171-18.1.103.

[2] T. Litman, “Exploring the Paradigm Shifts Needed To Reconcile Transportation and Sustainability Objectives,” *Transp Res Rec*, vol. 1670, no. 1, pp. 8–12, Jan. 1999, doi: 10.3141/1670-02.

[3] J. F. Sallis, L. D. Frank, B. E. Saelens, and M. K. Kraft, “Active transportation and physical activity: opportunities for collaboration on transportation and public health research,” *Transp Res Part A Policy Pract*, vol. 38, no. 4, pp. 249–268, 2004, doi: https://doi.org/10.1016/j.tra.2003.11.003.

[4] T. Shannon, B. Giles-Corti, T. Pikora, M. Bulsara, T. Shilton, and F. Bull, “Active commuting in a university setting: Assessing commuting habits and potential for modal change,” *Transp Policy (Oxf)*, vol. 13, no. 3, pp. 240–253, 2006, doi: https://doi.org/10.1016/j.tranpol.2005.11.002.

[5] A. Golub and K. Martens, “Using principles of justice to assess the modal equity of regional transportation plans,” *J Transp Geogr*, vol. 41, pp. 10–20, 2014, doi: https://doi.org/10.1016/j.jtrangeo.2014.07.014.

[6] K. Martens, A. Golub, and G. Robinson, “A justice-theoretic approach to the distribution of transportation benefits: Implications for transportation planning practice in the United States,” *Transp Res Part A Policy Pract*, vol. 46, no. 4, pp. 684–695, 2012, doi: https://doi.org/10.1016/j.tra.2012.01.004.

[7] K. Coffel *et al.*, “Guidelines for Providing Access to Public Transportation Stations,” 2012.

[8] M. Catala, S. Dowling, and D. M. Hayward, “Expanding the Google Transit Feed Specification to Support Operations and Planning,” 2011.

[9] J. Wong, “Leveraging the General Transit Feed Specification for Efficient Transit Analysis,” *Transportation Research Record: Journal of the Transportation Research Board*, vol. 2338, pp. 11–19, Dec. 2013, doi: 10.3141/2338-02.

[10] J. Wong, L. Reed, K. Watkins, and R. Hammond, “Open Transit Data: State of the Practice and Experiences from Participating Agencies in the United States,” 2013.

[11] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numer Math (Heidelb)*, vol. 1, no. 1, pp. 269–271, 1959, doi: 10.1007/BF01386390.

[12] R. Bellman, “On a routing problem,” *Q Appl Math*, vol. 16, no. 1, pp. 87–90, 1958.

[13] R. W. Floyd, “Algorithm 97: Shortest path,” *Commun ACM*, vol. 5, no. 6, p. 345, 1962, doi: http://doi.acm.org/10.1145/367766.368168.

[14] D. B. Johnson, “Efficient Algorithms for Shortest Paths in Sparse Networks,” *J. ACM*, vol. 24, no. 1, pp. 1–13, Jan. 1977, doi: 10.1145/321992.321993.

[15] J. Mote, I. Murthy, and D. L. Olson, “A parametric approach to solving bicriterion shortest path problems,” *Eur J Oper Res*, vol. 53, no. 1, pp. 81–92, 1991, doi: https://doi.org/10.1016/0377-2217(91)90094-C.

[16] J. C. Namorado Climaco and E. Queirós Vieira Martins, “A bicriterion shortest path algorithm,” *Eur J Oper Res*, vol. 11, no. 4, pp. 399–404, 1982, doi: https://doi.org/10.1016/0377-2217(82)90205-3.

[17] E. Q. V. Martins, “On a multicriteria shortest path problem,” *Eur J Oper Res*, vol. 16, no. 2, pp. 236–245, 1984, doi: https://doi.org/10.1016/0377-2217(84)90077-8.

[18] C. Tung Tung and K. Lin Chew, "A multicriteria Pareto-optimal path algorithm," *Eur J Oper Res*, vol. 62, no. 2, pp. 203–209, 1992, doi: https://doi.org/10.1016/0377-2217(92)90248-8.

[19] J. Brumbaugh-Smith and D. Shier, "An empirical investigation of some bicriterion shortest path algorithms," *Eur J Oper Res*, vol. 43, no. 2, pp. 216–224, 1989, doi: https://doi.org/10.1016/0377-2217(89)90215-4.

[20] H. W. Corley and I. D. Moon, "Shortest Paths in Networks with Vector Weights," *J. Optim. Theory Appl.*, vol. 46, no. 1, pp. 79–86, May 1985, doi: 10.1007/BF00938761.

[21] H. G. Daellenbach and C. A. De Kluyver, "Note on Multiple Objective Dynamic Programming," *Journal of the Operational Research Society*, vol. 31, no. 7, pp. 591–594, Jul. 1980, doi: 10.1057/jors.1980.114.

[22] A. J. V Skriver and K. Andersen, "A label correcting approach for solving bicriterion shortest-path problems," *Comput Oper Res*, vol. 27, pp. 507–524, May 2000, doi: 10.1016/S0305-0548(99)00037-4.

[23] P. Dell'Olmo, M. Gentili, and A. Scozzari, "On Finding Dissimilar Pareto-Optimal Paths," *Eur J Oper Res*, vol. 162, pp. 70–82, Apr. 2005, doi: 10.1016/j.ejor.2003.10.033.

[24] E. Machuca, L. Mandow, and J. Cruz, "An evaluation of heuristic functions for bicriterion shortest path problems," *New Trends in Artificial Intelligence. Proceedings of EPIA'09*, Jan. 2009.

[25] L. Mandow and J. L. de la Cruz, "Frontier Search for Bicriterion Shortest Path Problems," in *Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, NLD: IOS Press, 2008, pp. 480–484.

[26] L. Mandow and J. L. Pérez de la Cruz, "Path recovery in frontier search for multiobjective shortest path problems," *J Intell Manuf*, vol. 21, no. 1, pp. 89–99, 2010, doi: 10.1007/s10845-008-0169-2.

[27] R. Mart\'\i, J. Luis González Velarde, and A. Duarte, "Heuristics for the Bi-Objective Path Dissimilarity Problem," *Comput. Oper. Res.*, vol. 36, no. 11, pp. 2905–2912, Nov. 2009, doi: 10.1016/j.cor.2009.01.003.

[28] A. Raith and M. Ehrgott, "A comparison of solution strategies for biobjective shortest path problems," *Comput Oper Res*, vol. 36, pp. 1299–1331, Apr. 2009, doi: 10.1016/j.cor.2008.02.002.

[29] J. Widuch, "A Label Correcting Algorithm for the Bus Routing Problem," *Fundam Inform*, vol. 118, pp. 305–326, Aug. 2012, doi: 10.3233/FI-2012-716.

[30] C.-L. Liu, T.-W. Pai, C.-T. Chang, and C.-M. Hsieh, "Path-planning algorithms for public transportation systems," in *ITSC 2001. 2001 IEEE Intelligent Transportation Systems. Proceedings (Cat. No.01TH8585)*, 2001, pp. 1061–1066. doi: 10.1109/ITSC.2001.948809.

[31] A. V. MUSTAFA ALZAIDI, "Trip Planning Algorithm For Gtfs Data With Nosql Structure To Improve The Performance," *J Theor Appl Inf Technol*, vol. Vol.99. No, no. 10 31st May 2021, pp. 2290–2300, May 2021.

[32] S. Farber, B. Ritter, and L. Fu, "Space–time mismatch between transit service and observed travel patterns in the Wasatch Front, Utah: A social equity perspective," *Travel Behav Soc*, vol. 4, pp. 40–48, 2016, doi: https://doi.org/10.1016/j.tbs.2016.01.001.

[33] S. K. Fayyaz S., X. C. Liu, and G. Zhang, "An efficient General Transit Feed Specification (GTFS) enabled algorithm for dynamic transit accessibility analysis," *PLoS One*, vol. 12, no. 10, pp. e0185333-, Oct. 2017, [Online]. Available: https://doi.org/10.1371/journal.pone.0185333

[34] S. Motamed, A. Broumandnia, and A. Nourbakhsh, "Multimodal biometric recognition using particle swarm optimization-based selected features," *Journal of Information Systems and Telecommunication*, vol. 1, pp. 79–87, Mar. 2013, doi: 10.7508/jist.2013.02.002.

[35] P. Goli and M. M. R. KARAMI, "Speech Intelligibility Improvement in Noisy Environments for Near-End Listening Enhancement," 2016.

[36] G. M. Saeed, H. B. N. Babak, and L. Mojtaba, "Achieving Better Performance of S-MMA Algorithm in the OFDM Modulation," 2013.

[37] Q. Zervaas, *The Definitive Guide to GTFS (Consuming open public transportation data with the General Transit Feed Specifcation)*, First Edit. 2014. [Online]. Available: http://gtfsbook.com/gtfs-book-sample.pdf

[38] N. Amirah, D. Mohamad, and A. Hilmy, *Acceptable walking distance accessible to the nearest bus stop considering the service coverage*. 2021. doi: 10.1109/ICOTEN52080.2021.9493435.

[39] "Introduction to Redis – Redis." https://redis.io/topics/introduction (accessed Jan. 18, 2021).

[40] S. Tapia-Fernández, D. García-García, and P. García-Hernandez, "Key Concepts, Weakness and Benchmark on Hash Table Data Structures," *Algorithms*, vol. 15, no. 3, 2022, doi: 10.3390/a15030100.