ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

# TRIP PLANNING ALGORITHM FOR GTFS DATA WITH NOSQL STRUCTURE TO IMPROVE THE PERFORMANCE

**[1]MUSTAFA ALZAIDI, [2]ANIKO VAGNER**

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

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

E-mail: [1] mustafa.alzaidi@inf.unideb.hu,
[2] vagner.aniko@inf.unideb.hu

## ABSTRACT

Nowadays, GTFS (General Transit Feed Specification) data is used by many transport agencies as a standard format for publishing their data. Trip planning applications where a user can plan a trip between two locations are widely used. This paper will introduce an algorithm to found all possible trip plans between any two locations using GTFS data. The algorithm uses both stop and route level search to found the possible transition to be made. We introduce a technique that reduces the server overhead by defining and implementing a Redis NoSQL data structure to store all possible search results. Trip planning request at the server will be served by querying the Redis structure instead of run the algorithm with each request. We experiment, measure, and list the server performance using this technique with two different GTFS data sets and compare performance with and without using it.

Keywords: *GTFS, Trip Planning, Redis, NoSQL, Smart City***.**

## 1 INTRODUCTION

In 2005, Google started using the Google-Maps web application to create a transit trip planner. TriMet and Google in Portland formulated the General Transit Feed Specification (GTFS) [1]. In 2007, Google published transit feed specifications and encouraging transit agencies to use and depend on the GTFS format to create and post transit data on the web for public use as open sources. Later the feed becomes the most commonly used standard for static transit data exchange in the United States [2]. Because of its rising and popularity, the transit industry adopted the GTFS format as a standard for sharing their schedule data. At that time, over 170 transit agencies in the United States and Canada generate and publish their schedules as GTFS [3]. Later, applications that use GTFS like CarFreeAtoZ, Hopstop, and MapQuest were created, providing on-map stops locations, bus timetables, and trip-planners feature. The essential part of such applications is trip-planner, where the server searches GTFS data to found a possible route between two locations. Considering trips, routes, and transit between trips in stops (bus stop or transport station), planning a trip is more complicated than finding a path in a graph.

Finding the shortest path in a graph is a common problem, and there are several algorithms like Dijkstra [4], Bellman-Ford [5][6], Floyd-Warshall [7], Johnson [8]. The algorithm may be a bi-criterion or multi-criterion, where the criterion here is how we consider the weight for the graph edges. For example, if the graph represents a road network, the road weight may be a bi-criterion by considering the distance and the cost or maybe a multi-criterion if we consider more factors. Many algorithms try to solve the shortest path problem by reducing the weight value to a single value; these algorithms fall into categories like k-th shortest path algorithms [6], two phases algorithms [9], label setting algorithms [10][11][12], label correcting algorithms[13]–[16], and others[17]–[22]. Some research [23][24] uses a variation of these algorithms to introduce an algorithm to find paths in local transport networks. The algorithm's performance is affected by the considered weight criteria and the data structure used for the implementation. This paper will introduce a new algorithm as a new variation, where the performance is enhanced by using a data structure that reduces the time required for search the data. The algorithm ignores the weight criteria using the number of transitions made to evaluate results instead of calculating and examing weight to choose the best possible next transition every time.

In trip-planning applications, every time a user runs a trip planner, the server runs the route searching or trip planning algorithm, which is a time-consuming process. This paper also proposes a new technique to enhance the server response time and reduce the server overhead by introducing a Redis NoSQL data structure to store all possible plans between any two stops, eliminating the need to run the algorithm with every user request. As any search request can be served using the data in the Redis structure. We experiment with this strategy, the GTFS data of Debrecen and Budapest cities, from [25][26] and measure and compare the performance with and without using Redis. The generated data can increase the ability to analyze the transportation service and help the city provide better public transportation using a Smart City paradigm[27]. We implement the algorithm and the strategy as an open-source project using C#. The project code is available at https://github.com/mustafamajid/GTFS.git.

## 2 OVERVIEW OF GTFS DATA

Google develops General Transit Feed Specification (GTFS) as a format to define public transport systems data. Its main objective was to allow public transit agencies to upload their data to Google Transit schedules so that Google Maps users could easily decide which bus, train, or other vehicles to take for transport between two specific locations. GTFS can describe the public transportation schedules and associated geographic information and make this information easy to access and used standardly by users and transport application developers.

### 2.1 GTFS Files and Structure

GTFS is a set of text files that contain data about local transport trips, routes, stops, and time table data in a CSV format. Each file represents a database table. GTFS includes three types of tables Required, Optionally Required, and Optional tables [28]. This categorization gives the GTFS data the flexibility to include more or less data according to the agency or the application needs. The required file must be present in all of the GTFS. The optionally required files must be present depending on some other GTFS data. Next, we will describe the required files:

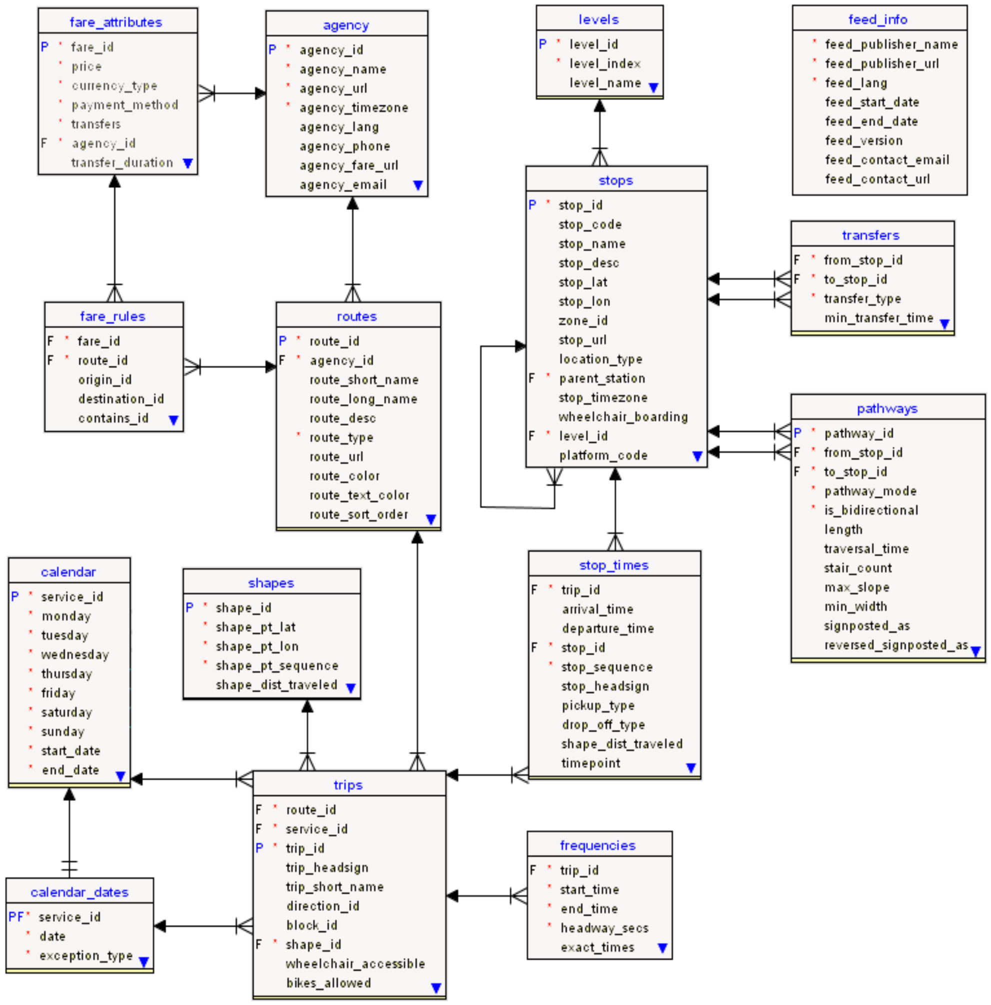
- **agency.txt:** Contains information about the transport agencies providing the transport service described by the GTFS data set. The file lists information like agency ID, agency name, URL of the agency website. It can also optionally show information like the URL of the ticket pushes site for the users and contact information like email and phone number.
- **stops.txt:** Contains information about transport stops and stations like stops name, stops ID, and the geographical location of the stops as latitude and longitude values. This location information is very useful for transport applications, especially for trip planning. It can be used to calculate the distance between the stops, which helps decide the next transition or determine the walkable distance between stops.
- **routes.txt:** describes the trajectories for each trip with general and geographical information. Each route is pass-through a set of stops or stations. Information like route ID, route name, and agency ID links each route with the agency that provides the service.
- **trips.txt:** describes the trip information like trip ID, head-sign, which is the text that appears on signs showing the destination of the trip for riders. The file contains a service Id field to refer to the service that this trip belongs. Also, the file shows the route ID of the route the trip uses.
- **stop_times.txt:** In this file, the arrival and departure times of the trip at each stop are listed.

### 2.2 Understand GTFS Data

There are three main objects in the GTFS: data routes, stops and trips. The route is a path that is pass-through a set of stops or stations. The trip refers to a journey made by a vehicle from an initial stop to an end stop. We can view or read GTFS data as one or more routes listed in the routes.txt file. Each route has one or more trips in the trips.txt file. Each trip visits a series of stops (stops.txt) at specified times in a specific sequence (stop_times.txt). Trips.txt and stoptimes.txt contain only time of day. The calendar.txt and calendar_dates.txt files determine which days a given trip runs [29]. Each file has a reference to the other related files. For example, trips.txt has reference to routes.txt file using the route ID field as foreigner key or reference. GTFS contains other files like fare_rules, shapes, and feeds_info [30]. Figure 1 shows the structure and the relations between the fields of the GTFS data file. As the GTFS aims to exchange transit information, specific preprocessing is needed so it can be used for other purposes. The GTFS data set is generally loaded into a relational database (e.g., MySQL, PostgreSQL, Oracle), where the developer will process the scheduling and transit data. In some cases, a database with a particular geographical object querying ability is used.

## 2.3 GTFS-Realtime

GTFS-Realtime is a standard created by Google to enable transit agencies to provide updates about their services in real-time. The set of data the GTFS-realtime feed provides contains vehicle positions, trip updates, and service alerts. Vehicle Positions contain data about the past, but Trip Updates contain data about the future. Typically, one GTFS-realtime feed can contain only one type of data out of these three types. Many governmental and business agencies have multiple GTFS - real-time feeds (one for vehicle positions, one for trip updates and one for service alerts).



## 3 THE ALGORITHM

Usually, a variation of label-setting and label-correcting algorithms, especially Dijkstra algorithms, are used to implement trip planning algorithms. The data structure used by the trip

*Figure 1:GTFS Structure*

planning algorithm to hold the GTFS data play a primary role in the algorithm performance and behaviour, as it can affect the speed and logic of data access. A trip-planning algorithm can be implemented at the stop level where every time the algorithm searches for a possible next stop or at the route level where the algorithm checks the available route each time. This algorithm will use both the stop level and route level search. We will consider the data in Figure 2 as an example to describe the algorithm steps. Each circle represents a stop. The number inside the circle is the stop_id. The routes are represented by a set of arrows denoted by letters.

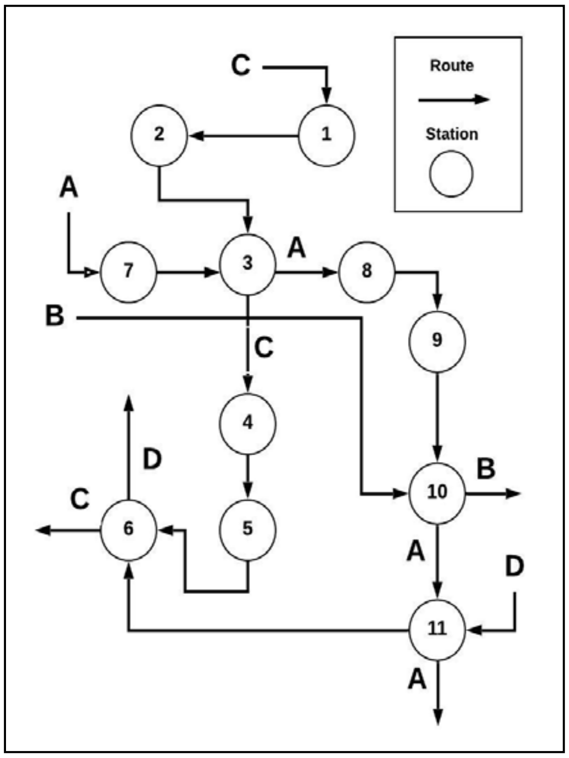


*Figure 2: GTFS Routing Network Example*

## 3.1 Preprocessing

In the preprocessing steps, we create and fill the data structures with the GTFS data. The task here is to make the data more understandable by reducing the joining effort and faster and easier access by the algorithm code. Preprocessing contain the following steps:

### 3.1.1 Find the routes

The route file in the GTFS data provides an ID for each route, but some routes may have different stops set to visit during different trips [31]. We distinguish each route by the list of stops it visits in a specific order. We use the stoptimes file to extract this list for each route and create a route data structure containing all information provided about that route in the route file and a list of all stops this route visit, ordered by the visiting sequence. For example, in Figure 2, the list of stops for route A is {7,3,8,9,10,11}. Finally, a list of route objects will be created. The route structure has a VISITED index variable used to indicate whether the route is visited or not. Initially, the VISITED variable will initialize to -1 to indicate that no stop in this route is visited yet, and prevent rechecking the already checked routes. For example, if we use route B to reach stop10 and then from that stop, the algorithm starts to check all possible routes and put all the next stops in the waiting list to be checked, the VISITED variable for route A will be set to stop10. Thus only the stops before stop10 can be checked next time using route A, and if all the stops in the route are checked, the VISITED will be set to zero, the index of the first stop.

### 3.1.2 Create stops list

The stop data structure contains all the static information about a stop provided by the stops file of the GTFS data with a list of all routes that pass through this stop. The route list can be extracted using the route data from the previous step. For example, in Figure 2, the route list of stop3 is {A, C}.

### 3.1.3 Find distance between stops

The stops file contains the latitude and longitude of each stop location. We use the Haversine formula [32] to calculate each stop's distance with all the other stops. The formula is shown in Figure 3. If the distance between two stops is walkable, we create a walkable route between these stops. The walkable route uses the same route data structure with all the fields set to the “walk” string, and the stops list contains only

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi 1 \cdot \cos\varphi 2 \cdot \sin^2(\Delta\lambda/2)$$

$$c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{(1-a)})$$

$$d = R \cdot c$$

$$\text{where } \varphi \text{ is latitude}, \lambda \text{ is longitude}, R \text{ is earth's radius (mean radius} = 6{,}371km)$$

*Figure 3: Haversine formula*

the two stops, which are linked by this walkable distance. Walkable routes are important when the user must change the route by walking to another stop.

### 3.1.4 Trips timing data extraction

After using the algorithm to find possible solutions, a time check is needed to validate the routes according to the trip's timetable, considering the time parameter provided by the user's query. Time data is collected from the stoptimes file and calendar_date file. The stopstimes file shows the arrival and departure times at the stops the trip visits. Each set of trips on a specific route is denoted as a service with a unique service_id. The calander_dates file has two fields, start_date and end_date, which denote the period when the service is available. The trips file links each trip with the route and the service it belongs. In this step, we extract this data into a time data structure and create a dictionary to link each route and trip with its service id.

### 3.1.5 Creating dictionaries

The above steps are about loading the GTFS data into memory. All the stops, routes, and time data are loaded into lists of stop data structure, route data structure, and time data structure. We increase the algorithm execution speed by use dictionaries for the indexes of these lists. A dictionary will map the stop ID to an integer value, representing the index of the stops data structure in the list. Thus, we can access a stop object in a list of stops with the stop ID using the dictionary. That will eliminate the need to search the list for a specific stop and increase the execution speed as the dictionary's mapping time is O(1) [33].

## 3.2 The Output Data Structure

The algorithm input is the initial stop ID and the destination stop ID. The output is a list of all possible trip plans where each plan may contain one or a combination of routes. We used the word PATH to denote a plan (a solution). Each PATH contains a list of one or more transition made from one stop to another using a specific route. We denote these transitions as MOVE and the list of MOVE in each PATH as WAY. The PATH data structure contains attributes to denote the initial stop, and the destination stop, with their names. The MOVE data structure contains the start-stop ID where this transition starts from, the end-stop ID, the route ID used to make this transition, and other information like service ID, the stops and the route names. Figure 4 shows the structure of PATH and MOVE. For example, in Figure 2, the algorithm is used to find the possible solution from stop2 to stop6. Solutions will be a list that contains two PATHes. The first solution is a PATH containing only one MOVE in the WAY list. This MOVE describes a transition from stop2 to stop6 using route C. The second solution is a PATH, which contains three MOVEs in the WAY list. The first MOVE mentions a transition from stop2 to stop3 using route C. The second MOVE mentions a transition from stop3 to stop11 using route A, the third MOVE is from stop11 to stop6 using route D.

## 3.3 Definition

- TRANSIT LIMIT (TL): A constant number determines the limit of change between transport trips (for example, between two buses). This value is used to

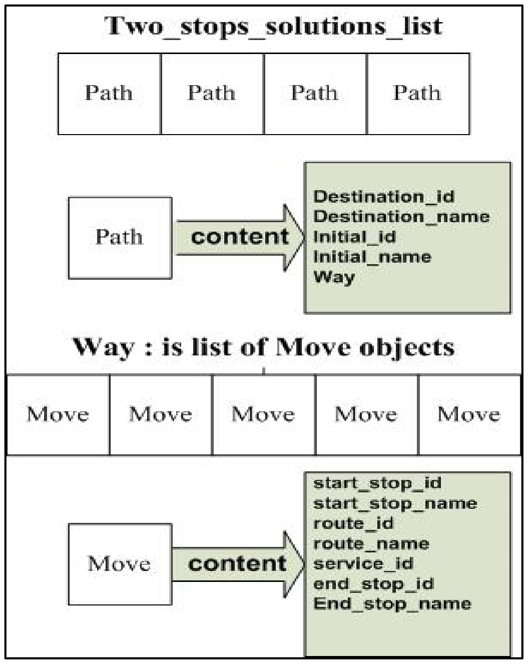


*Figure 4: Path and Move Data Structure*

  stop the searching and prevent the algorithm from stuck in an endless loop. Studies show that the typical transit limit is four transit per plan [34], as people did not prefer to use plans with many transits, as it causes west of time and extra cost.
- INITIAL: The initial stop where the planning starts from.
- FINAL: The final step of (the destination).
- MOVE (S1, S2, R): Denote a possible transfer from the stop S1 to stop S2 using the route R.
- PATH: A list of MOVEs where for each consecutive two MOVEs PATH[i] and PATH [i+1], PATH[i].S2 = PATH[i+1].S1 each PATH represent a possible route solution.
- SOLUTION LIST (SOL_LIST): A list of PATH used to store all the possible solutions.
- TRANS_COUNT: Counter represents the number of transit during the travel till the Current stop
- QUEUE ITEM(C, TRANS_COUNT, WAY_PASSED): a data structure used to store algorithm parameters, where

WAY_PASSED is a PATH variable, C is a stop currently reached starting from INITIAL stop and going through WAY_PASSED, TRANS_COUNT is an integer represents a count of transit mad to travel the WAY_PASSED.

- Queue: A Queue of Queue_Item, stored according to the execution order

### 3.4 Algorithm

Figure 5 shows the pseudocode of the algorithm.

```
INPUT : INITIAL, FINAL
OUTPUT: SOL_LIST
        Step1: Initialize  QUEUE by add QUEUE_ITEM(INITIAL,0,EMPTY WAY_PASSED)
        and set SOL_LIST =empty
Step 2: IF (Queue is empty ) then Exit
       Else
       pop the first QUEUE_ITEM Q and  Set Current = Q.C , WAY_PASSED =
       Q.WAY_PASSED ,
       TRANS_COUNT =Q.TRANS.
Step3: IF ((Current == FINAL) OR(TRANS_COUNT== TRANS_LIMIT)) then  Goto Step2.
Step4: For each unvisited route  Ri  that pass through Current:
      IF Ri.VISITED not refer to Current or any earliar stop in Ri then:
      starting from Current, IF next Stops in Ri contain the Final then :
        ▪ Add MOVE( Current, FINAL, Ri) to  a copy of WAY_PASSED.
        ▪ Create new PATH using the newally created WAY_PASSED
        ▪ Add PATH to SOL_LIST.

Step5: For each unvisited route  Ri  that pass through Current:
      ▪ For each Stop S in Ri after Current :
          • Add MOVE(Current, S, Ri) to  a copy of WAY_PASSED name it
            NEW_WAY_PASSED.
          • Add QUEUE_ITEM(S,TREANS_COUNT + 1, NEW_WAY_PASSED ) to QUEUE
          • Mark Ri as visited from Current by updating Ri.VISITED

Step6: Goto Step2
```

*Figure 5:Trip planning algorithm for GTFS data*

## 4 IMPROVE THE PERFORMANCE USING REDIS DATABASE

Any path planning algorithm is time-consuming, especially with extensive data. For the trip planning applications, the server handles the overhead of running the planning algorithm. Every time a user requests a plan, the server must run the algorithm and search the GTFS data. Here we proposed a technique to increase the server performance and reduce the overhead by search the trip plans between all the stops of the GTFS data and store it in a Redis database. The result will be retrieved from the Redis server for all user search requests without rerunning the algorithm.

## 4.1 Redis DataBase

Redis is an in-memory NoSql database that offers high performance. Redis developed ANSI C and worked in most POSIX systems [35]. Redis stores the data as a key-value and can be used as a massage broker or for session management. For example, an HTML page with its resources can be serialized as a string and stored in Redis structure to provide faster page load. It has five data structures Strings, Lists, Set, Hash, and Sorted Set. Many programming languages support Redis. For each language, there is a set of libraries and packages that support connecting and manipulate data in the Redis server. We used StackExchange.Redis package in the project. This package can be installed using the Nuget Package Manager.

## 4.2 Redis Structure For Trip Plans

The trip plan structure resulting from the algorithm is a PATH data structure defined earlier; the most informative part of the PATH data structure is the WAY attribute, a list of MOVE objects. The task here is to define a Redis data structure model that stores all possible plans between two stops (list of PATHs). Both MOVES and PATH attributes are converted by concatenated to a single string using a special-character string "||" as a value separator. We experiment with two models. The first model separated the PATH and the MOVE list it contains, and the second model keeps PATH and its MOVE list in a single string.

### 4.2.1 Model -1 HASH and LIST

This model uses the indexing concept. We use a Hash with Key equal to both stops IDs separated by "____". The Hash store number of values fields equal to the number of solutions for these two stops, the value for each field store a List name (a reference to a Redis List) where each element in the list represent a MOVE mad by this Path.

### 4.2.2 Model-2 LIST

We use a Redis List structure to store all possible solutions (PATH list) for any two stops. The list Key will be the two stops IDs separated by "___". Each value in the list represents a PATH object. All the data stored in the Path object is converted to string and concreted together as a single string. Both models are illustrated in Figure 6.

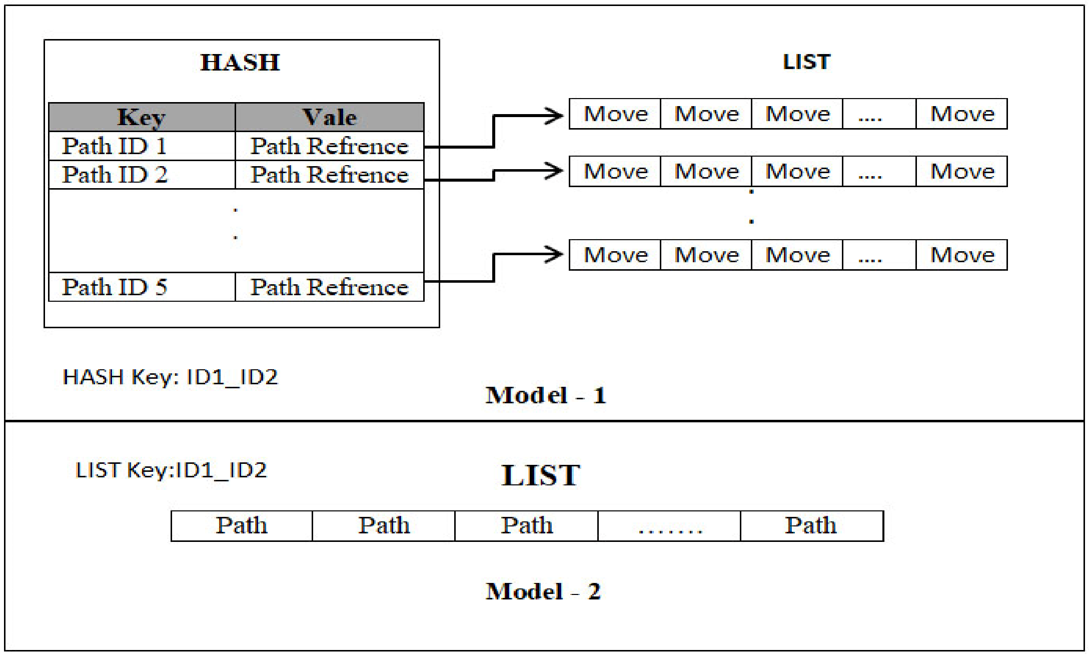


*Figure 6: Redis models for PATH data structure*

After the experiment, we found that the second model provided a faster performance. As retrieving the hash value takes O(1) time, then for n PATH stored in the hash will take O(n) besides O(1) for the indexing list. While the second model needs only O(1) to retrieve the solutions, and it easier for implementation, although it needs more back-end processing to extract the data from the string, so we depend on it in the C# project.

## 4.3 Measure and compare the performance

As performance enhancement is the key point behind using Redis, we execute the algorithm by choosing two random stops and recording the execution time. Then we compared that time with the time taken to request and retrieve the solution for the same stops from the Redis server, as we already stored all possible solutions between any two stops in Redis. Because GTFS data size affects the algorithm performance, we conduct 28 experiments with the GTFS data for two cities varying in size, Debrecen(1483 KB) and Budapest(42128KB). The experiment results are shown in Table 1.

*Table 1: Performance measurement in milliseconds*

| NO | Budapest | | Debrecen | |
|---|---|---|---|---|
| | Without Redis | Using Redis | Without Redis | Using Redis |
| 1 | 1.0381 | 0.08930 | 0.8575 | 0.08908 |
| 2 | 1.0452 | 0.08915 | 0.8587 | 0.08919 |
| 3 | 0.9909 | 0.08907 | 0.8510 | 0.08901 |
| 4 | 1.0320 | 0.08909 | 0.8946 | 0.08925 |
| 5 | 1.0551 | 0.08902 | 0.8953 | 0.08920 |
| 6 | 1.0398 | 0.08929 | 0.8720 | 0.08919 |
| 7 | 1.0492 | 0.08901 | 0.8600 | 0.08906 |
| 8 | 0.9930 | 0.08903 | 0.8795 | 0.08902 |
| 9 | 1.0418 | 0.08911 | 0.8901 | 0.08902 |
| 10 | 1.0335 | 0.08932 | 0.8684 | 0.08901 |
| 11 | 1.0071 | 0.08930 | 0.8562 | 0.08921 |
| 12 | 1.0004 | 0.08914 | 0.8786 | 0.08928 |
| 13 | 1.0255 | 0.08911 | 0.8583 | 0.08914 |
| 14 | 1.0050 | 0.08908 | 0.8625 | 0.08923 |
| 15 | 1.0268 | 0.08918 | 0.8557 | 0.08910 |
| 16 | 1.0254 | 0.08935 | 0.8540 | 0.08906 |
| 17 | 1.0515 | 0.08917 | 0.8544 | 0.08919 |
| 18 | 1.0300 | 0.08909 | 0.8763 | 0.08923 |
| 19 | 1.0476 | 0.08905 | 0.8998 | 0.08907 |
| 20 | 1.0519 | 0.08926 | 0.8849 | 0.08921 |
| 21 | 1.0005 | 0.08918 | 0.8615 | 0.08906 |
| 22 | 1.0585 | 0.08900 | 0.8785 | 0.08906 |
| 23 | 0.9961 | 0.08919 | 0.8954 | 0.08918 |
| 24 | 1.0340 | 0.08911 | 0.8865 | 0.08921 |
| 25 | 1.0253 | 0.08915 | 0.8932 | 0.08901 |
| 26 | 1.0215 | 0.08924 | 0.8815 | 0.08922 |
| 27 | 1.0317 | 0.08916 | 0.8848 | 0.08921 |
| 28 | 1.0381 | 0.08930 | 0.8575 | 0.08908 |
| Average | 1.0280 | 0.08915 | 0.8737 | 0.08914 |

The above results are computed using the same hardware and computation power ( CPU: Intel Core i7-3720QM 2.6 GHz, 6MB L3 cache, RAM: 8GB, OS: WIN10 64bit), it is clear that using Redis can produce high performance and reduce the server overhead as the computation time will be reduced. We can notify that, with both cities retrieving Redis data is faster than performing a real-time search, as shown in Figure 7. Also, we note that the average time required to search the Budapest data (1.0280 milliseconds) is greater than the average time required for Debrecen (0.08915). But using Redis, we do not have this variation as the recorded time is 0.08915 and 0.08914 for the two cities.

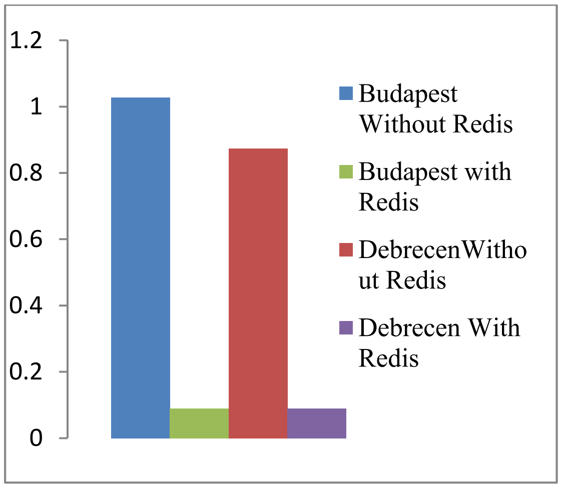


*Figure 7: Execution Time Comparison (Millisecond)*

## 5 SUMMERY

Sharing transport information is a critical factor toward a successful transportation system in modern and Smart Cities. Thus, the need to use a standard format for sharing transport data is increased. Transport agencies widely use GTFS (General Transit Feed Specification ) worldwide as a standard format to share and publish their data. Thus, the need to efficiently process and utilize GTFS is increased. Trip planning is one of the most demanded applications that use GTFS data. Such an application can tell the user which local transport trip or trips (e.g., bus, tram, metro) can travel between two locations. Algorithms that find a path in a graph are a bi-criterion or multi-criterion, where the criterion is the number of weight for the graph edges. We introduce a new variation of the trip planning algorithm, which ignore the criterion factor using transition limit to evaluate the search and considering both stop level and route level search. Thus, we enhance the performance by eliminating the time required to assign and process the arc's weight and reduce the algorithm implementation complexity. We introduce data structure and an implementation method that can produce better performance.

Querying the server to run a trip planning algorithm with each user request is time-consuming and can cause a server overhead epically with large cites data. We introduce a method to enhance the server performance and reduce the server overhead using the Redis NoSQL database structures. Redis data structures are used to store all the trip plan between any two GTFS stops. Thus, we can serve user requests by querying the Redis server, which is much faster than run a searching algorithm using GTFS data in real-time. We propose two Redis models. The first model uses Redis HASH and LIST structure, where the second model uses LIST structure only. Both models provide similar performance but can be utilized differently by the software developer according to the development and the application needs.

We experiment with the server performance and show the performance enhancement using Budapest and Debrecen cities GTFS data. The experiments show that this method can produce a faster performance with nearly constant time regardless of the data size (city size).

The generated data can also provide a source of information for analysis and planning the local transport system in Smart Cities.

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

[2] J. Wong, “Leveraging the General Transit Feed Specification for Efficient Transit Analysis,” *Transp. Res. Rec. J. Transp. Res. Board*, vol. 2338, pp. 11–19, Dec. 2013, doi: 10.3141/2338-02.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] P. Hansen, "Bicriterion Path Problems BT - Multiple Criteria Decision Making Theory and Application," 1980, pp. 109–127.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] H. G. Daellenbach and C. A. De Kluyver, "Note on Multiple Objective Dynamic Programming," *J. Oper. Res. Soc.*, vol. 31, no. 7, pp. 591–594, Jul. 1980, doi: 10.1057/jors.1980.114.

[16] 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.

[17] 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.

[18] E. Machuca, L. Mandow, and J. Cruz, "An evaluation of heuristic functions for bicriterion shortest path problems," *New Trends Artif. Intell. Proc. EPIA'09*, Jan. 2009.

[19] 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*, 2008, pp. 480–484.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] "Debrecen Regional Transport Association." http://www.derke.hu/ (accessed Apr. 08, 2021).

[26] "BKK GTFS - OpenMobilityData." https://transitfeeds.com/p/bkk/42?fbclid=IwAR1vI63TXiBkXtLgGfptJw4LI5SDxeCUVFlK7xsGQ45RaUwR0qSVKwlGrdo (accessed Apr. 08, 2021).

[27] R. Velasquez, F. Rodriguez, M. Vargas Martin, and J. Ponce, "Mapping of the Transportation System of the City of Aguascalientes Using GTFS Data for the Generation of Intelligent Transportation Based on the Smart Cities Paradigm," 2020, pp. 177–185.

[28] "Reference | Static Transit | Google Developers." [Online]. Available: https://developers.google.com/transit/gtfs/reference.

[29] Q. Zervaas, *The Definitive Guide to GTFS (Consuming open public transportation data with the General Transit Feed Specifcation)*, First Edit. 2014.

[30] “General Transit Feed Specification.” [Online]. Available: https://gtfs.org/reference/static.

[31] A. Queiroz, V. Santos, D. Nascimento, and C. Santos Pires, *Conformity Analysis of GTFS Routes and Bus Trajectories*. 2019.

[32] C. C. Robusto, “The cosine-haversine formula,” *Am. Math. Mon.*, vol. 64, no. 1, pp. 38–40, 1957.

[33] “Dictionary<TKey,TValue> Class (System.Collections.Generic) | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0.

[34] S. 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, p. e0185333, Oct. 2017, doi: 10.1371/journal.pone.0185333.

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