Efficient Data Structure and Algorithms for Minimum Transfers in Public Transportation Network

Mithinti Srikanth
cs18d501@iittp.ac.in
Indian Institute of Technology Tirupati
Tirupati, Andhra Pradesh, India

Aditya Pandey
ee20b053@iittp.ac.in
Indian Institute of Technology Tirupati
Tirupati, Andhra Pradesh, India

G. Ramakrishna
rama@iittp.ac.in
Indian Institute of Technology Tirupati
Tirupati, Andhra Pradesh, India

ABSTRACT

This paper explores the MIN-TRANSFERS problem, defined as the minimum number of vehicle transfers needed to travel from one location to all locations in a public transportation network, represented using a temporal graph. Solving this problem is crucial because transfers, which involve switching vehicles, add complexity and may extend journey times. This is especially important for elderly, disabled individuals, women, children, and anyone traveling with heavy luggage, as minimizing transfers can significantly ease their travel experience. While previous studies have focused on minimizing the number of edges rather than vehicle transfers. To address this, we introduce a novel data structure Temporal Paths Preservation (TPP)-graph, to calculate the minimum number of transfers from a source location to all the locations efficiently. We present five algorithms tailored for solving the one-to-all MIN-TRANSFERS problem using TPP-graph: the Single Queue Algorithm, the No Queue Algorithm, the Multiple Queue Algorithm, the Priority Queue Algorithm and the Multiple Priority Queue Algorithm. We assess the performance of these algorithms through experiments on real-world public transportation datasets, providing insights into their running times and efficiency.

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**; **Dynamic graph algorithms**; **Shortest paths**.

KEYWORDS

Temporal Graph, Minimum Number of Transfers, Public Transportation Network, Temporal Path Preservation Graph, Transformed Graph.

ACM Reference Format:
Mithinti Srikanth, Aditya Pandey, and G. Ramakrishna. 2024. Efficient Data Structure and Algorithms for Minimum Transfers in Public Transportation Network. In *2024 Sixteenth International Conference on Contemporary Computing (IC3-2024) (IC3 2024), August 08–10, 2024, Noida, India.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3675888.3676057

IC3 2024, August 08–10, 2024, Noida, India
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0972-2/24/08
https://doi.org/10.1145/3675888.3676057

1 INTRODUCTION

As the population increases, the demand for transportation grows, leading to heightened congestion, higher oil prices, air pollution, and resource depletion. Consequently, public transport is becoming an increasingly attractive alternative to private vehicles. Efficient public transport networks are essential for reducing operational costs and enhancing route flexibility. Although the public transport system is cost-effective, it need not be beneficial, particularly due to delays incurred while switching vehicles and subsequent cascading effects. In response to these challenges, this work focuses on the minimum-transfers (MIN-TRANSFERS) problem in public transport systems. Solution to this problem benefits all users, especially the elderly, disabled, women, children, and those with heavy luggage. Solving the MIN-TRANSFERS problem not only improves user satisfaction but also assists transportation administrators in creating more effective route plans.

A public transportation network can be represented as a temporal graph $G = (V, E)$, where V represents vertices (locations) and E denotes directed edges (routes) between these vertices, as illustrated in Figure 1. Each edge in a temporal graph is denoted by a tuple (u, v, t, λ, vid), where u is the starting vertex, v is the ending vertex, t is the start time at u, λ is the travel duration from u to v (assuming that the duration value in public transportation data is always greater than zero), and vid is the vehicle ID. This representation effectively captures both the edge's availability and the travel time. For example, as shown in Figure 1, the edge $(1, 2, 3, 2, 1)$ indicates that the journey starts at vertex v_1, ends at vertex v_2, departs at time 3 from v_1, and arrives at v_2 at time 5 using vehicle id 1.

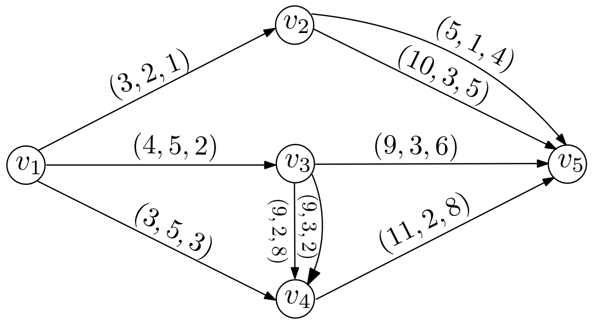


Figure 1: A Temporal Graph

If the departure time at each intermediate vertex is at least the arrival time of the preceding edge, then the path formed by a series of edges is called a *temporal path*. The number of times two

Table 1: Frequently Used Notations

Symbol	Description
$G = (V, E)$	Temporal Graph
$\tilde{G} = (\tilde{V}, \tilde{E})$	TPP-Graph
n, m	Number of vertices & edges in G
$\tilde{n}, \tilde{m}$	Number of vertices & edges in $\tilde{G}$
$\text{left}(x)$	Left vertex of a node x in $\tilde{G}$
$\text{right}(x)$	Right vertex of a node x in $\tilde{G}$
$level(x)$	Level of a node x in $\tilde{G}$
$l_{\max}$	Maximum number of nodes in a level of $\tilde{G}$
d_{max}	Maximum out degree of G
$\tilde{r}$	Number of reachable nodes from source nodes in $\tilde{G}$

consecutive edges in a temporal path have different vehicle IDs is called the number of *vehicle transfers* of that path. The smallest number of vehicle transfers over all temporal paths connecting a vertex s to a vertex z is called the *minimum transfers*. In this paper, we will use the terms "*hops*" and "*transfers*" interchangeably to refer to the count of vehicle changes required during a journey. From Figure 1, we can observe that the number of vehicle transfers of a temporal path $(v_1, 4, 5, 2, v_3)$, $(v_3, 9, 3, 2, v_4)$ is zero, where as it is one for the temporal path $(v_1, 4, 5, 2, v_3)$, $(v_3, 9, 2, 8, v_4)$. The goal of the MIN-TRANSFERS problem is to determine the minimum number of vehicle transfers needed for each vertex z in G to travel from a source vertex s.

MIN-TRANSFERS problem
Input: A temporal graph G, a source vertex s.
Output: Minimum transfers from s to all vertices in G.

Previous works have extensively studied several characteristics of various route optimisation problems. Specifically, research has focused on solving problems related to the shortest path [3, 5, 19–21], earliest arrival time [6, 12, 14, 21], fastest path [4, 5, 21], latest departure time [21], and several Pareto-optimal solutions have been proposed based on waiting time, duration, and the number of transfers[2, 7–9, 13]. However, the minimum transfer problem, which involves finding the minimum number of vehicle changes, still needs to be explored.

Takamatsu et al., using a mathematical optimisation model, designed a timetable that ensures seamless transfers between buses and trains [18]. Effectively optimizing public transport network transfers through well-coordinated timetabling and vehicle scheduling [10, 11, 15–17]. During the vehicle scheduling and timetabling, various approaches haven been explored to reduce the number of vehicle transfers, and thus these problems focus to reduce transfers during the design time, whereas our focus is in the operations phase.

Jain et al. addressed the min hop problem by defining it as the minimum number of edges in a time-respecting path between two nodes, without considering the need for vehicle changes [8]. In contrast, our approach redefines min hops to specifically include the minimum number of vehicle changes required to travel from a source node to a destination node, thereby focusing more directly on enhancing the practicality of public transport systems.

One way to solve this problem is traversing through all possible paths from the source vertex to all other vertices in temporal graph and choose the path(s) with minimum number of transfers for each vertex. However, this approach is not efficient and not scalable, due to the potentially large number of paths we may need to explore for each vertex repeatedly. Another way to attack this problem is to explore and customize Dijkstra's algorithm. We recall that Dijkstra's algorithm works based on the principle that a subpath of a shortest path is a shortest subpath. However, this principle does not apply to the MINTRANSFERS problem. For instance, consider a path in a temporal graph depicted in Figure 1 consisting of three edges $\{\langle v_1, v_3, 4, 5, 2\rangle, \langle v_3, v_4, 9, 2, 8\rangle, \langle v_4, v_5, 11, 2, 8\rangle\}$ from vertex v_1 to vertex v_5. Here, the subpath from v_1 to v_4 using the first two edges does not minimise transfers because a more direct route exists. Consequently, rather than extending a single path, it is necessary to consider multiple paths to effectively address the MINTRANSFERS problem. Therefore, neither Dijkstra's algorithm nor its time-variant adaptations are suitable for solving this specific issue in temporal graphs.

While traversing paths in a temporal graph G, for any two consecutive edges e and e', it is necessary to confirm whether or not the departure time of e' is at least the arrival time of e. These computations are treated as *valid* or *invalid* computations accordingly. Refer to Table 2 and Figure 2, which are obtained on real-world datasets, showing that, on average, 51% of consecutive edges in the original temporal graph G are invalid due to non-compliance with time constraints. This high rate of invalid edges highlights the need for a more reliable framework.

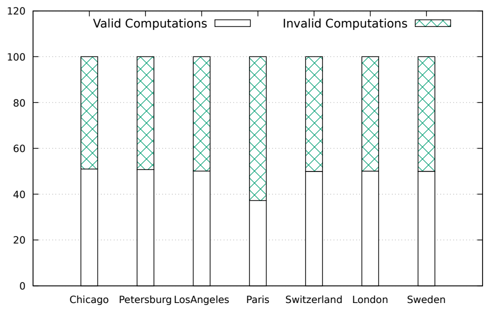


Figure 2: Number of valid and invalid computations

Table 2: Percentage of Invalid Computations

Data Sets	Valid Computations	Invalid Computations
Chicago	26943830	25923001
Petersburg	3650684893	3550146211
Los Angeles	250521728	249826629
Paris	7284699861	12294072358
Switzerland	15452811770	15480686863
London	10248905463	10233276479
Sweden	4241652453	4243510448

The technical motivation of designing a *temporal paths preservation*-graph data structure along with efficient algorithms is to address this significant issue by creating a structure containing only temporal paths. By doing so, we aim to enhance the efficiency of temporal data processing. This specialized approach ensures that all edges automatically meet the temporal validity criteria and significantly reduces the complexity and computational overhead associated with validating each edge pair in large-scale graph environments.

Our key contributions are outlined as follows.

- Data Structure. We introduce a novel temporal data representation, temporal paths preservation graph (TPP-graph), by transforming a temporal graph to preserve only time-respecting paths, effectively eliminating all others to prevent invalid computations.
- Algorithms. We have developed five distinct algorithms to address the one-to-all MIN-TRANSFERS problem using the TPP-graph. Each algorithm offers a unique approach to reduce the practical running time.
- Proof of Correctness. We provide rigorous proofs to verify the correctness of each algorithm, ensuring that they function accurately across a range of expected conditions and conform to the theoretical model.
- Experimental insights. Our algorithms have been empirically tested on real-world datasets, demonstrating their effectiveness and efficiency. Additionally, we have analyzed parameters that effect the running times of proposed algorithms.

These contributions collectively push the boundaries of how temporal graphs are utilized to solve transportation problems, explicitly optimizing the MIN-TRANSFERS problem in public transit systems.

2 TEMPORAL PATHS PRESERVATION GRAPH

In this section, we introduce a graph transformation on a temporal graph to preserve time-respecting paths and eliminate the rest of them to avoid invalid computations. Later, we prove that there is a one-to-one mapping between the time-respecting paths of a temporal graph and paths in the transformed graph. We conclude this section by presenting an algorithm to construct such a transformed graph.

TPP-graph. The graph transformation on a temporal graph results in a *Temporal Paths Preservation Graph*, and shortly known as TPP-graph. For a temporal graph G, we define *Temporal Paths Preservation Graph*, TPP-graph $\tilde{G}$, as follows. Every edge e in G corresponds to a node v_e in $\tilde{G}$. For any two edges e and e' in G, we add an edge from v_e to $v_{e'}$ in G', if right(e) = left(e') and dep$(e') \geq$ arr(e). Further for an edge $(v_e, v_{e'})$, the weight $w(v_e, v_{e'})$ is assigned as zero, if vehicle-ids of e and e' are same, otherwise the weight is assigned as one. For a path $\tilde{P}$ in $\tilde{G}$, the weight of $\tilde{P}$ is defined as the sum of the weights of its edges and is represented by $w(\tilde{P})$. For any edge (v_e, v'_e) in $\tilde{G}$, the departure time of v'_e is greater than the departure time of v_e, as the duration time on every edge is a positive number. Thus we do not have any cycles in $\tilde{G}$, and hence the TPP-graph is a directed acyclic graph (DAG).

We now prove that there is a one to one mapping between time respecting paths in G and paths in $\tilde{G}$ in Lemma 2.1.

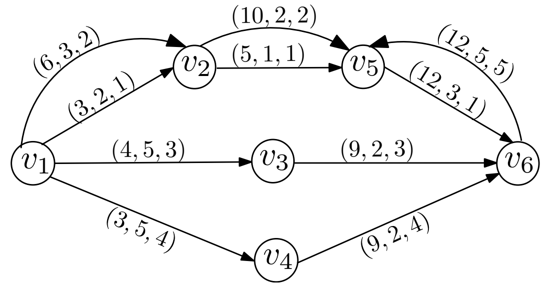


Figure 3: A Temporal Graph

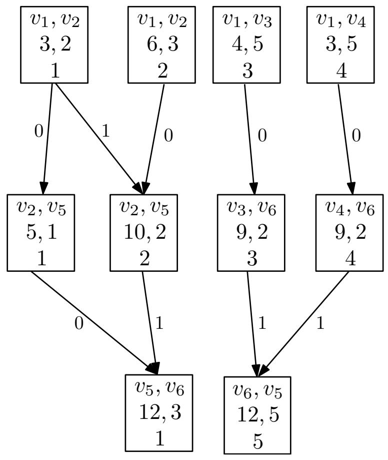


Figure 4: A TPP-graph in which each node x is represented as 5-tuple, left(x), right(x), dep(x), duration(x), vehicleId(x)

LEMMA 2.1. *Let P be a path formed with a sequence $(e_1, \ldots, e_k)$ of edges in G and P' be a sequence $(v_{e_1}, \ldots, v_{e_k})$ of vertices in $\tilde{G}$. P is a time respecting path in G if and only if P' is a path in $\tilde{G}$. Also, $transfers(P) = w(P')$.*

PROOF. From the definition of TPP-graph, for any two consecutive edges e_i and e_{i+1} in P, right(e_i) = left(e_{i+1}) and dep$(e_{i+1}) \geq$ arr(e_i) if and only if there is an edge from v_{e_i} to $v_{e_{i+1}}$. Thus, we can conclude that P is a path in G if and only if $\tilde{P}$ is a path in $\tilde{G}$. The way of assigning weights to the edges of a TPP-graph ensures the following, for any two consecutive edges e_i and e_{i+1} in P. $w(v_{e_i}, v_{e_{i+1}}) = 1$ when vid$(e_i) \neq$ vid(e_{i+1}) and $w(v_{e_i}, v_{e_{i+1}}) = 0$ when vid(e_i) = vid(e_{i+1}). Thus, $transfers(P) = w(P')$. □

TPP-graph Construction. Given a temporal graph G, we describe the pseudo code to obtain a TPP-graph $\tilde{G}$ of G in Algorithm 1. For each edge $e = (u, v, t, \lambda, vid)$, we navigate through all the outgoing edges e_i of v in decreasing order based on their departure times. As long as the departure time of e_i is at least the arrival time of e, we add an edge (e, e_i). While adding edges in the TPP graph, we optimize the process by reducing the number of checks between any two nodes x and y. For a node x in $\tilde{G}$ such that right$(x) = u$, we sort all y nodes in $\tilde{G}$ with left$(y) = u$ by their departure time t and store them in an array. We then traverse this array in decreasing order to establish dependencies, as shown in Line 5. If we find a y such that $\text{dep}(y) < \text{arr}(x)$, we can skip the remaining elements in the array. Further, weights are assigned to the edges using vehicle ids as shown in from Line 6 to Line 9. This process establishes the connectivity between nodes based on their vertices and vehicle ids, ensuring efficient traversal by avoiding unnecessary computations. For a temporal graph shown in Figure 3, the corresponding transformed TPP graph is shown in Figure 4.

Algorithm 1: Converting Temporal Graph to TPP-Graph

```
Input: A temporal graph $G$
Output: A TPP-graph $\tilde{G}$ of $G$
1  for each edge e in G do
2      $V(\tilde{G}) = V(\tilde{G}) \cup \{v_e\}$;
3  for each node x in $\tilde{G}$ do
4      $u = \text{right}(x)$ ;
5      for each outgoing edge $y(u, v)$ of x
         in decreasing order of departure time such that
        $arr(x) \leq dep(y)$ do
6          if $vid(x) = vid(y)$ then
7              $w(x, y) = 0$
8          else
9              $w(x, y) = 1$
10 Initialize an empty queue $q$ ;
11 Compute and store the in degree of each node in $d_{in}$ ;
12 for each node x in $\tilde{G}$ such that $d_{in}[x] = 0$ do
13     $q.insert(x)$ ; $level[x] = 0$
14 while $(|q| \geq 1)$ do
15     $x = q.front(); q.delete()$ ;
16     for each outgoing neighbor y of x in $\tilde{G}$ do
17         $level[y] = max(level[y], level[x] + 1)$ ;
18         $d_{in}[y] - -$ ;
19         if $d_{in}[y] = 0$ then
20             $q.insert(y)$
```

In the second phase of Algorithm 1, we assign a level number to all the nodes of the TPP-graph. We insert all nodes with zero in-degree into a queue. Then, we dequeue nodes one by one until the queue is empty. For each dequeued node x, we iterate through its adjacent nodes y in $\tilde{G}$. We update the level of each adjacent node y as $\max(\text{level}[y], \text{level}[x] + 1)$, and decrease the indegree of y. If the in-degree of y becomes zero after this update, we insert it in the queue q. The time complexity of this algorithm is $O(m * d_{max})$, where m denotes the number of edges in G and d_{max} denotes the maximum number of outgoing edges starting from a node in $\tilde{G}$.

3 PROPOSED ALGORITHMS FOR MINIMUM NUMBER OF TRANSFERS PROBLEM

We start this section by providing an overview of our algorithms. Later, we describe the notation required to follow the proposed algorithms. Further, we describe the pseudo code of four algorithms along with their merits and demerits.

Overview of our Algorithms. The transformation of a temporal graph G to a TPP-graph $\tilde{G}$ does not depend on a source vertex. This leads us to perform graph transformation exactly once in the preprocessing time to obtain the TPP-graph, using Algorithm 1. For each outgoing edge of a source vertex in G, there is a corresponding source node in $\tilde{G}$. In order to calculate the weight of all paths originating from source nodes, our algorithms share the fundamental idea to traverse from source nodes in a TPP-graph and find the reachable nodes. For exploring all the reachable nodes and computes the weights of all such paths from various source nodes, we design multiple graph traversal algorithms. We identify various challenges, explore multiple design choices to resolve them and use various data structures that are suitable to perform traversal on the TPP-graph. In particular, we propose five algorithms, namely the SQ-algorithm, the NQ-algorithm, the MQ-algorithm, the PQ-algorithm and the MPQ-algorithm, which are based on the use of a **s**ingle **q**ueue, **n**o **q**ueue, **m**ultiple **q**ueues, a **p**riority **q**ueue, and **m**ultiple **p**riority **q**ueues respectively.

Notation. For each node x in $\tilde{G}$, we use $nodeMinHops[x]$ to store the minimum number of transfers required to reach x from a source node. For each node x in $\tilde{G}$, there is a boolean flag $status[x]$, which holds *true* or *false*, to denote the respective vertex as active or passive. For each vertex z in G, $minHops[z]$ helps to store the minimum number of transfers from s to z. A node in $\tilde{G}$ that corresponds to an outgoing edge of a source vertex s in G is referred to as a *source node*. The `relax` function updates the transfer count at node y if a smaller number of transfers is found from node x through edge $w(x, y)$, ensuring that nodeMinHops$[y]$ always holds the minimum number of transfers required to reach y. The `relax` function returns true when it successfully decreases the transfer count, otherwise it returns false.

Given a TPP-graph of G and a source vertex s as input, the goal is to compute the minimum number of transfers from s to all other vertices in G and store the minimum number of transfers in $minHops$ array. All our algorithms start with a common initialization phase, the steps of which are described below.

Initialization. During the initialization phase, all source nodes in $\tilde{G}$ are marked as active and the rest of them are marked as passive in the status array. Also, *nodeMinHops* of source nodes are set to zero, whereas for the rest of the nodes, *nodeMinHops* is set to infinity. Similarly, *minHops* is set to zero for the source vertex in G and to infinity for the rest of them.

3.1 Single Queue Algorithm

Algorithm 2 is based on the usage of a single queue, and starts with the steps described in the initialization phase. Later, we insert all source nodes of $\tilde{G}$ in the queue. We then traverse the queue until it is empty, repeatedly removing a node from the queue, relax its outgoing edges, and insert the outgoing neighbours in the queue. After we relax an edge (x, y), we update the $minHops[\text{right}[y]]$, if the newly obtained number of transfers is lesser than the previous one. We only insert a node to the queue when it is not in the queue to avoid multiple copies. It is important to note that the same node can be processed multiple times in this approach. The time complexity of this algorithm is $O(\tilde{n} * d_{max})$, as every node is processed for every incoming edge in the worst case.

Algorithm 2: Single Queue Algorithm

```
Input: A TPP-graph $\tilde{G}$ of temporal graph $G$ with source
       vertex $s$.
Output: For each vertex $z$ in $G$, $minHops[z]$ stores the
        minimum number of transfers from $s$ to $z$.
1  for each vertex z in G do  $minHops[z] = \infty$;
2  $minHops[s] = 0$ ;
3  for each node x in $\tilde{G}$ do
4      $insideQ[x] = false$ ;
5      $nodeMinHops[x] = \infty$
6  Initialize an empty queue $q$ ;
7  for each node x in $\tilde{G}$ such that $x.u = s$ do
8      $q.insert(x)$ ; $insideQ[x] = true$ ;
9      $minHops[\mathrm{right}(x)] = 0$ ;
10     $nodeMinHops[x] = 0$ ;
11 while ($|q| \geq 1$) do
12     $x = q.front()$; $q.delete()$; $insideQ[x] = false$ ;
13     for each outgoing neighbor y of x in $\tilde{G}$ do
14         success =
             $relax(nodeMinHops[x], w(x, y), nodeMinHops[y])$;
15         $minHops[\mathrm{right}(y)] = \min\{minHops[\mathrm{right}(y)],$
             $nodeMinHops[y]\}$;
16         if success = true and $insideQ[y] = false$ then
17             $q.insert(y)$, $insideQ[y] = true$
```

Algorithm 3: No Queue Algorithm

```
Input: A TPP-graph $\tilde{G}$ of temporal graph $G$ with source
       vertex $s$.
Output: For each vertex $z$ in $G$, $minHops[z]$ stores the
        minimum number of transfers from $s$ to $z$.
1  for each vertex z in G do
2      $minHops[z] = \infty$
3  $minHops[s] = 0$ ;
4  for each node x in $\tilde{G}$ do
5      $status[x] = false$ ;
6      $nodeMinHops[x] = \infty$
7  for each node x in $\tilde{G}$ such that $x.u = s$ do
8      $status[x] = true$ ;
9      $nodeMinHops[x] = 0$ ;
10     $minHops[\mathrm{right}(x)] = 0$ ;
11 for each level node x in $\tilde{G}$ from 1 to L do
12     if status[x]=true then
13         for each outgoing neighbor y of x in $\tilde{G}$ do
14             $relax(nodeMinHops[x], w(x, y), nodeMinHops[y])$;
15             $minHops[\mathrm{right}(y)] = \min\{minHops[\mathrm{right}(y)],$
                 $nodeMinHops[y]\}$;
16             $status[y] = true$
```

Challenge. How to avoid redundant processing of nodes?

3.2 No Queue Algorithm

This algorithm is based on level order traversal and does not use any queue. We initialize the *nodeMinHops* and *minHops* arrays as described in the initialization section. At the beginning of the algorithm, all the source nodes appearing in level-1 are made as active. Then all the nodes in TPP-graph are processed in a level-order fashion. Then, in a level-order fashion starting from level 1, we check all nodes with an active status and relax their outgoing edges. For each outgoing edge (x, y) from an active node, we update $minHops[\mathrm{right}(y)]$ using the updated $nodeMinHops[x]$, and most importantly mark y as active as shown in Line 16. We then move on to process all the rest of the levels one by one. This approach ensures that each node is processed exactly once, unlike the previous algorithm, thus avoids redundant processing. The time complexity of this algorithm is $O(\tilde{n} + \tilde{m})$.

Challenge: How to ensure that all active nodes are processed exactly once while traversing the graph in level-order fashion, avoiding redundant processing and maintaining the correctness of the algorithm.

3.3 Multiple Queues Algorithm

We solve the drawback of the previous algorithm by pruning several nodes in a TPP-graph, with the help of multiple queues. After initializing the necessary arrays such as *nodeMinHops* and *minHops*, for each level $1 \leq i \leq l$, we create an empty queue $q[i]$ corresponds to maintain the active nodes in level i of $\tilde{G}$.

For all source nodes, we insert them into the appropriate queue according to their level number. We also maintain an array *insideQ* to mark whether a node is inside a queue or not. The algorithm then iterates through all queues level-wise. For each queue, it removes a node x until the queue is empty, relaxes its outgoing edges (x, y) and updates the necessary array entries such as $nodeMinHops[y]$ and $minHops[\mathrm{right}(y)]$. Following the level number of vertex y, it is inserted into the corresponding queue if y is not present in the queue. This process continues until all the queues are empty level-wise. Note that in this approach, we process a node at most once, ensuring efficient pruning without redundancy. The time complexity of this algorithm is $O(\tilde{r} + l)$, here $\tilde{r}$ is the number of reachable nodes from source nodes in $\tilde{G}$ and l is number of levels in the $\tilde{G}$.

Challenge. How to avoid processing empty queues?

3.4 Priority Queue Algorithm

This algorithm maintains a minimum priority queue, which avoids traversing levels that do not contain active nodes. In Algorithm 5, we initialize the *nodeMinHops* and *minHops* arrays as described in the initialization section. We then use both a priority queue and an array of queues to traverse the graph efficiently. For source nodes, we insert their level number into the priority queue if the queue at that level is empty.

We then insert the node into the corresponding queue according to its level number and update the status of the node in *insideQ*

Algorithm 4: Multiple Queues Algorithm

```
Input: A TPP-graph $\tilde{G}$ of temporal graph $G$ with source vertex $s$.
Output: For each vertex $z$ in $G$, $minHops[z]$ stores the minimum number of transfers from $s$ to $z$.
1  for each vertex $z$ in $G$ do
2      $minHops[z] = \infty$
3  $minHops[s] = 0$ ;
4  for each node $x$ in $\tilde{G}$ do
5      $insideQ[x] = false$; $nodeMinHops[x] = \infty$
6  Initialize array of empty queue $q$ of size $L$ ;
7  for each node $x$ in $\tilde{G}$ such that $x.u = s$ do
8      $q[level(x)].insert(x)$ ; $insideQ[x] = true$ ;
9      $minHops[\text{right}(x)] = 0$ ;
10     $nodeMinHops[x] = 0$ ;
11 for $i = 0$ to $L$ do
12     while $(|q[i]| \geq 1)$ do
13         $x = q[i].front()$; $q[i].delete()$ $insideQ[x] = false$ ;
14         for each outgoing neighbor $y$ of $x$ in $\tilde{G}$ do
15             $relax(nodeMinHops[x], w(x,y), nodeMinHops[y])$;
16             $minHops[\text{right}(y)] = \min\{minHops[\text{right}(y)], nodeMinHops[y]\}$;
17             if $insideQ[y] = false$ then
18                 $q[level(y)].insert(y)$, $insideQ[y] = true$
```

as true. While the priority queue is not empty, we dequeue the level number from the priority queue and iterate through all nodes present in the queue corresponding to the level number. For each node x in the queue, we remove it, relax the outgoing edges (x, y), update $nodeMinHops[y]$ and $minHops[\text{right}(y)]$, and insert the node y into the corresponding queue, if y is not present in the queue. Most importantly, when a vertex is inserted into the corresponding queue for the first time, we insert the level of y into the priority queue to register the level numbers with active nodes. This approach ensures efficient traversal of the graph, processing each node at most once and skipping levels with no active nodes. The time complexity of this algorithm is $O(\tilde{r} \log l)$.

Multiple Priority Queues Algorithm. Through our experiments, we observe there is an overhead in Algorithm 4 (MQ) and Algorithm 5 (PQ), due to the `if-condition` inside the `inner-for-loop`. In both of these algorithms, a statement such as `if-condition` inside the `inner-for-loop` executes for each active edge, whereas a statement outside the `inner-for-loop` executes for each active node. The number of active edges is typically more than the number of active nodes. These observations help us to design an algorithm based on multiple priority queues, shortly referred as MPQ-algorithm. In this algorithm, we maintain a priority queue instead of a standard queue for each level, to store the active nodes belonging to the respective level. In Algorithm 4, we replace the standard queue with a priority queue and introduce a flag for each node to monitor whether it is settled. After deleting a node x from a

Algorithm 5: Priority Queue Algorithm

```
Input: A TPP-graph $\tilde{G}$ of temporal graph $G$ with source vertex $s$.
Output: For each vertex $z$ in $G$, $minHops[z]$ stores the minimum number of transfers from $s$ to $z$.
1  for each vertex $z$ in $G$ do
2      $minHops[z] = \infty$
3  $minHops[s] = 0$ ;
4  for each node $x$ in $\tilde{G}$ do
5      $insideQ[x] = false$; $nodeMinHops[x] = \infty$
6  Initialize an array of empty queues, denoted by $q$, with size $L$
7  Initialize an empty priority queue $pq$
8  for each node $x$ in $\tilde{G}$ such that $left(x) = s$ do
9      if $q[level(x)].isempty()$ then $pq.insert(level(x))$;
10     $q[level(x)].insert(x)$ ; $insideQ[x] = true$ ;
11     $minHops[\text{right}(x)] = 0$ ;
12     $nodeMinHops[x] = 0$ ;
13 while $pq.isnotempty()$ do
14     $i = pq.extractMin()$ ;
15     while $(|q[i]| \geq 1)$ do
16         $x = q[i].front()$; $q[i].delete()$ $insideQ[x] = false$ ;
17         for each outgoing neighbor $y$ of $x$ in $\tilde{G}$ do
18             $relax(nodeMinHops[x], w(x,y), nodeMinHops[y])$;
               $minHops[\text{right}(y)] = \min\{minHops[\text{right}(y)], nodeMinHops[y]\}$;
19             if $insideQ[y] = false$ then
20                 if $q[level(y)].isempty()$ then
21                     $pq.insert(level(y))$
22                 $q[level(y)].insert(y)$, $insideQ[y] = true$
```

priority queue, we verify whether x is settled or not. We explore the neighbours of x only when x is deleted for the first time (settled) from the priority queue. In other words, neighbours of x are not explored if x is settled in the previous iterations. Essentially, a conditional statement from MQ-algorithm and PQ-algorithm is moved from inside the `inner-for-loop` to outside the `inner-for-loop` to obtain this algorithm. The time complexity of this algorithm is $O(l + \tilde{r} \log l_{max})$.

4 ALGORITHMS CORRECTNESS

The main focus of this section is to present the proof of correctness of the SQ-algorithm (Algorithm 2), the NQ-algorithm (Algorithm 3), the MQ-algorithm (Algorithm 4) the PQ-algorithm (Algorithm 5) and the MPQ-algorithm.

We start with proof of correction of the SQ-algorithm. Although SQ-algorithm inserts a same node x multiple times in a queue, we can observe that x is inserted in the queue only when $nodeMinHops[x]$ is reduced. The number of node transfers for every node is an integer bounded by the number of edges in the corresponding path. Therefore, either the number of node transfers for a node is decreased or no node is inserted in the queue in each iteration of the outer loop of Algorithm 2. In other words, either the node transfers for a node is decreased or the number of nodes in the queue is reduced. This process eventually settle all nodes to a minimum number of transfers in $\tilde{G}$. Additionally, this guarantees that there will be a minimum number of hops from a source vertex to each other reachable vertex in G.

Moving forward, we now consider at the proof of correctness of other four algorithms. Despite various differences in the algorithms NQ, MQ, PQ and MPQ, all of these algorithms follow a certain type of level order traversal. In other words, before a node in level k is visited, another node in one of its previous levels must have been visited, excluding the source nodes. This helps us to prove the correctness of four algorithms using Lemma 4.1 and Theorem 4.2.

LEMMA 4.1. *Let $\tilde{G}$ be an TPP-graph of a temporal graph G and s be a source vertex in G. Let $P = (x, y)$ be a path in $\tilde{G}$ such that, left$(x) = s$, right$(y) = z$, $1 \leq$ level(x), <= level$(y) \leq k$. At the end of $k - 1^{th}$ iteration, Algorithm 3 computes the number of hops of P.*

PROOF. To prove this lemma, we apply induction on k. In the base case, we consider a path on single node v_e in $\tilde{G}$ such that, left$(v_e) = s$ and level$(v_e) = 1$. The base case holds true before the after the initialization, as number of hops to the right vertices of source nodes are initialized with zero, during the initialization phase. There is a path P from x to y such that left$(x) = s$, right$(y) = z$, $1 \leq$ level$(x) \leq$ level$(y) \leq k$, due to the premise of the lemma. According to the level(x) and level(y), we consider the following three cases.

The lemma is true when $1 \leq$ level(x), level$(y) < k$, due to induction hypothesis. Now, we consider another case where $1 \leq$ level$(x) < k$ and level$(y) = k$. As y appears in level k, the predecessor y' of y in P appears in level at most $k - 1$. As a result of the induction hypothesis, the algorithm calculates the number of transfers from x to y' at the conclusion of the $k - 2^{nd}$ iteration. In addition, the node y' must now have a true status because of Line 16 in Algorithm 3. Stated differently, at the end of processing level $k - 2$, the vertex y' is active and appears in a level that is at most $k - 1$. During the $k - 1^{th}$ iteration, the number of hopes of node y is updated using the number of transfers of y' and hence the claim is true. In the last case, levels of both x and y are k. These kind of nodes are made as active and processed during the initialization phase. Therefore, the algorithm ensures to compute the number of transfers at the end of of $k - 1^{th}$ iteration. □

THEOREM 4.2. *Given a temporal graph G and a source vertex s, upon completion of Algorithm 3, the value of $minTransfers[z]$ for each vertex z in G represents the minimum number of transfers from the given source vertex s to z.*

PROOF. Let P be a min-transfer path from s to z. Then we have an equivalent path Q in $\tilde{G}$, such that $transfers(P) = transfers(Q)$ due to Lemma 2.1. Algorithm 3 computes the number of transfers of all paths including Q because of Lemma 4.1. Finally, the number of transfers of all paths ending at a node y, such that right$(y) = z$ are compared and stores the minimum number of transfers in $minHops[z]$, as shown in Line 15 of Algorithm 3. Thus the theorem holds true. □

5 EXPERIMENTS

In this section, we describe our experimental setup, along with dataset characteristics. Later, we provide the performance of our algorithms. Finally, we analyze the behavior of our algorithms based on certain parameters and present important findings using graph plots.

Experimental setup. The experimentation is conducted on a machine equipped with an INTEL XEON E5-2620 v4 CPU, operating at a frequency of 2.20 GHz, featuring 32 GB of primary memory and 512 MB cache memory. The compiler used is gcc version 5.4.0.

Dataset Preparation and Characteristics. As part of the data pre-processing for obtaining a temporal graph from GTFS data, we first downloaded the GTFS dataset as cited in [1]. We extracted crucial attributes including starting vertex, ending vertex, starting time, travel time, and vehicle ID, and relabeled the vertices from 0 to $n - 1$. Subsequently, we transformed this data into a weighted directed acyclic graph, termed the TPP graph, using Algorithm 1. This graph is stored in CSR format to facilitate constant-time access to neighbors and weights. Each node in this format is assigned an ID linked to five specific attributes. We then applied our four algorithms to six real-time public transportation datasets, with their detailed characteristics presented in Table 3.

Table 3: Dataset Characteristics

Data Sets	**n**	**m**=$\tilde{n}$	$\tilde{m}$	**l**
Chicago	240	98,157	26,943,830	680
London	20,843	14,064,967	10,248,905,463	2,055
Los Angels	13,975	1,979,340	250,521,728	1,910
Paris	411	1,068,284	7,284,699,861	93
Petersburg	7,573	4,437,010	3,650,684,893	1,641
Sweden	45,727	6,567,745	4,241,652,453	2,205

5.1 Performance of our five algorithms

We used Algorithm 1 on 6 real world datasets to construct the corresponding TPP-graph $\tilde{G}$. Then we ran SQ-algorithm, NQ-algorithm, MQ-algorithm and PQ-algorithm on TPP-graphs starting from a random source vertex, and measured the running time. We run the same experiment ten times by considering a random source vertex each time, and measure the average running time of ten experiments. The average running time obtained for each algorithm on each dataset is reported in Table 4. The single queue based algorithm suffers from redundant computations. We avoid this and prune many computations effectively in the rest of the three algorithms. The speedup of NQ-Algortihm, MQ-algorithm and PQ-algorithm with respect to SQ-algorithm for all 6 real world datasets is shown in Figure 5.

5.2 Experimental Analysis and Key Insights

We now introduce a few key parameters that influence the performance of our algorithms in practice. During the graph traversals, we observe that our algorithms process a node or an edge multiple times. The number of nodes and the number of edges that are traversed including the duplicates are referred as *active nodes* and *active edges*, respectively. The number of levels in $\tilde{G}$ having at least

Table 4: Running Times in Seconds

Data Sets	SQ	NQ	MQ	PQ	MPQ
Chicago	0.72	0.53	0.56	0.55	**0.16**
London	8.76	10.25	**2.64**	2.66	7.31
Los Angels	82.69	48.07	61.66	40.85	**35.56**
Paris	8.20	106.48	80.91	76.05	**56.55**
Peters burg	589.30	351.06	285.73	297.01	**92.51**
Sweden	30.14	35.64	41.07	23.67	**7.89**

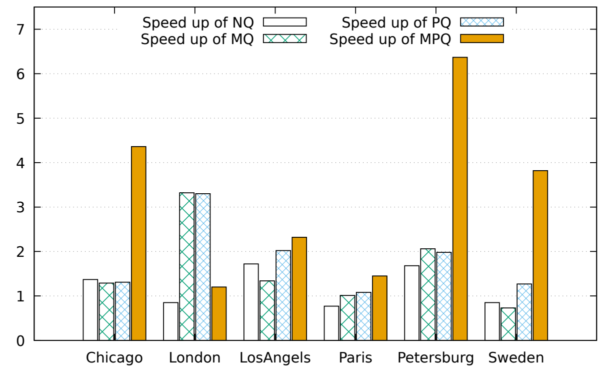
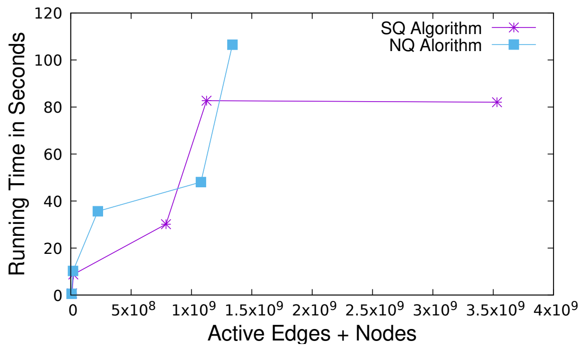


Figure 5: Speedup w.r.t Algorithm 2

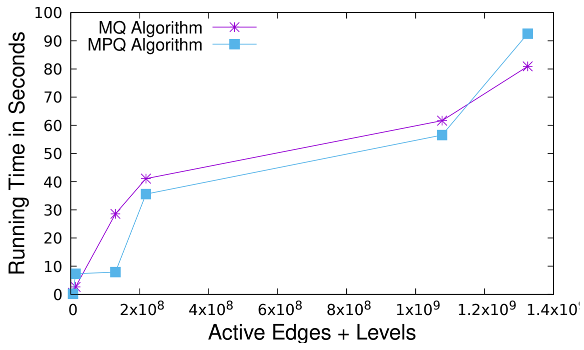
Table 5: Parameters affecting running time of proposed algorithms

DATASET	Active Edges in Algorithm 2	Active Edges in Algorithm 3, 4, 5	# Active Levels	# Empty Levels
Chicago	10804717	6504688	409	225
London	16730142	14040537	1,615	1332
Los Angeles	781966309	217975174	1,909	48
Paris	1122595325	1076681996	1,827	16
Peters burg	3517201931	1324941983	686	34
Sweden	211278134	129353104	1,749	569

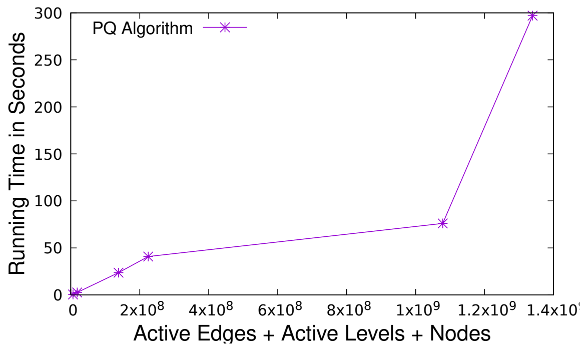
one active node is called *active levels*. Note that SQ-algorithm can traverse the same edge multiple times, where as at most once in other algorithms. Thus the number of active edges encountered during SQ-algorithm is higher than that of other algorithms, which is evident from Table 5. Furthermore, we can observe that the running time of the SQ algorithm increases as the number of active edges increases in Figure 6a The NQ-algorithm goes through all nodes exactly once, regardless of whether it is active or not. Thus the practical running time of NQ-algorithm depends on the number of active edges and the total of nodes, which is shown in Table 5 and Figure 6aFrom Table 5 and Figure 6b, it can be observed that the running time of MQ-algorithm and MPQ-algorithm depends on the number of active edges and total number of levels, as each level is visited independent on the number of active levels. Finally, we can observe the behaviour of PQ-algorithm from Figure 6c, in which we only traverse few active levels, and thus the running time is dependent upon the active edges and active levels.



(a) Influencing Parameters vs Running Time of Algorithm 2 and Algorithm 3



(b) Influencing Parameters vs Running Time of Algorithm 4 and MPQ algorithm



(c) Influencing Parameters vs Running Time of Algorithm 5

Figure 6: Impact of Influencing Parameters on the Running Time of Proposed Algorithms

6 APPLICATIONS OF OUR ALGORITHM

(1) To determine the set of destinations reachable from a source station with at-most k number of transfers, we can obtain min-hops array using proposed algorithms. Simply return the set of indices (destinations) for which the array value at that index is less than or equal to k. This set represents all destinations that can be reached from the source station within at most k transfers.

(2) If a customer needs to determine whether a destination can be reached from a source station in at most k transfers, our

algorithms can calculate the minimum transfers required. If k is less than this minimum, the path is not possible. Conversely, if k is greater than or equal to the minimum transfers, the path is feasible.

7 CONCLUSION

We designed an efficient data structure, the TPP-graph, to represent temporal graphs and devised five algorithms to address the MIN-TRANSFERS problem. Our algorithms compute the minimum number of transfers from a source vertex to all vertices in a temporal graph with the help of a **t**emporal **p**aths **p**reservation graph. These algorithms are designed to efficiently find routes with the fewest vehicle transfers, making them particularly useful in applications such as tour planning, where minimizing transfers is crucial for optimizing travel itineraries.

REFERENCES

[1] Bus Open Data Service. 2024. Regional and National Data Sets in GTFS Format. https://data.bus-data.dft.gov.uk/timetable/download/ Dataset is created using compliant data published to the Bus Open Data Service..

[2] Szabolcs Duleba and Sarbast Moslem. 2019. Examining Pareto optimality in analytic hierarchy process on real Data: An application in public transport service development. *Expert Syst. Appl.* 116 (2019), 21–30. https://doi.org/10.1016/J.ESWA.2018.08.049

[3] Lan Gao. 2015. Research on the Application of an Improved Shortest Path Algorithmin Public Transport System. In *10th International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2015, November 4-6, 2015*, Leonard Barolli, Fatos Xhafa, Marek R. Ogiela, and Lidia Ogiela (Eds.). IEEE Computer Society, Krakow, Poland, 580–582. https://doi.org/10.1109/BWCCA.2015.38

[4] Sanaz Gheibi, Tania Banerjee, Sanjay Ranka, and Sartaj Sahni. 2021. An Effective Data Structure for Contact Sequence Temporal Graphs. In *2021 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, Athens, Greece, 1–8. https://doi.org/10.1109/ISCC53001.2021.9631469

[5] Sanaz Gheibi, Tania Banerjee, Sanjay Ranka, and Sartaj Sahni. 2024. Path Algorithms for Contact Sequence Temporal Graphs. *Algorithms* 17, 4 (2024), 1–19. https://doi.org/10.3390/a17040148

[6] Chirayu Anant Haryan, G. Ramakrishna, Rupesh Nasre, and Allam Dinesh Reddy. 2020. A GPU Algorithm for Earliest Arrival Time Problem in Public Transport Networks. In *27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020*. IEEE, Pune, India, 171–180. https://doi.org/10.1109/HIPC50609.2020.00031

[7] Peilan He, Guiyuan Jiang, Siew-Kei Lam, Yidan Sun, and Fangxin Ning. 2022. Exploring Public Transport Transfer Opportunities for Pareto Search of Multicriteria Journeys. *IEEE Trans. Intell. Transp. Syst.* 23, 12 (2022), 22895–22908. https://doi.org/10.1109/TITS.2022.3194523

[8] Anuj Jain and Sartaj K. Sahni. 2022. Algorithms for optimal min hop and foremost paths in interval temporal graphs. *Appl. Netw. Sci.* 7, 1 (2022), 60. https://doi.org/10.1007/S41109-022-00499-3

[9] Rainer Kujala, Christoffer Weckström, Milos N. Mladenovic, and Jari Saramäki. 2018. Travel times and transfers in public transport: Comprehensive accessibility analysis based on Pareto-optimal journeys. *Comput. Environ. Urban Syst.* 67 (2018), 41–54. https://doi.org/10.1016/J.COMPENVURBSYS.2017.08.012

[10] Tao Liu, Wen Ji, Konstantinos Gkiotsalitis, and Oded Cats. 2023. Optimizing public transport transfers by integrating timetable coordination and vehicle scheduling. *Comput. Ind. Eng.* 184 (2023), 109577. https://doi.org/10.1016/J.CIE.2023.109577

[11] Renzo Massobrio, Sergio Nesmachnow, Jonathan Muraña, and Bernabé Dorronsoro. 2022. Learning to optimize timetables for efficient transfers in public transportation systems. *Appl. Soft Comput.* 119 (2022), 108616. https://doi.org/10.1016/J.ASOC.2022.108616

[12] Sunil Kumar Maurya and Anshu S. Anand. 2022. A Novel GPU-Based Approach to Exploit Time-Respectingness in Public Transport Networks for Efficient Computation of Earliest Arrival Time. *IEEE Access* 10 (2022), 81877–81887. https://doi.org/10.1109/ACCESS.2022.3192443

[13] Matthias Müller-Hannemann and Karsten Weihe. 2001. Pareto Shortest Paths is Often Feasible in Practice. In *Algorithm Engineering, 5th International Workshop, WAE 2001, August 28-31, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2141)*, Gerth Stølting Brodal, Daniele Frigioni, and Alberto Marchetti-Spaccamela (Eds.). Springer, Aarhus,Denmark, 185–198. https://doi.org/10.1007/3-540-44688-5_15

[14] P. Ni, M. Hanai, W. J. Tan, C. Wang, and W. Cai. 2017. Parallel Algorithm for Single-Source Earliest-Arrival Problem in Temporal Graphs. In *Proc. of the ICPP*. IEEE Computer Society, Bristol, United Kingdom, 493–502. https://doi.org/10.1109/ICPP.2017.58

[15] Claudio E. Risso, Sergio Nesmachnow, and Diego Rossit. 2022. Smart Mobility for Public Transportation Systems: Improved Bus Timetabling for Synchronizing Transfers. In *Smart Cities - 5th Ibero-American Congress, ICSC-CITIES 2022, Cuenca, Ecuador, November 28-30, 2022, Revised Selected Papers (Communications in Computer and Information Science, Vol. 1706)*, Sergio Nesmachnow and Luis Hernández-Callejo (Eds.). Springer, Cuenca, Ecuador, 158–172. https://doi.org/10.1007/978-3-031-28454-0_11

[16] Yousef Shafahi and Alireza Khani. 2010. A practical model for transfer optimization in a transit network: Model formulations and solutions. *Transportation Research Part A: Policy and Practice* 44, 6 (2010), 377–389. https://doi.org/10.1016/j.tra.2010.03.007

[17] Xu Sun, Kun Lin, Pengpeng Jiao, Zelin Deng, and Wei He. 2021. Research on transfer optimization model of county transit network. *International Journal of Environmental Research and Public Health* 18, 9 (2021), 4962. https://doi.org/10.3390/ijerph18094962

[18] Mizuyo Takamatsu and Azuma Taguchi. 2020. Bus Timetable Design to Ensure Smooth Transfers in Areas with Low-Frequency Public Transportation Services. *Transp. Sci.* 54, 5 (2020), 1238–1250. https://doi.org/10.1287/TRSC.2019.0918

[19] Shuai Wang, Yang Yang, Xiaolin Hu, Jianmin Li, and Bingji Xu. 2016. Solving the K-shortest paths problem in timetable-based public transportation systems. *J. Intell. Transp. Syst.* 20, 5 (2016), 413–427. https://doi.org/10.1080/15472450.2015.1082911

[20] Rini Wongso, Cin Cin, Suhartono, and Joseph. 2018. TransTrip: A Shortest Path Finding Application for Jakarta Public Transportation using Dijkstra Algorithm. *J. Comput. Sci.* 14, 7 (2018), 939–944. https://doi.org/10.3844/JCSSP.2018.939.944

[21] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. 2016. Efficient Algorithms for Temporal Path Computation. *IEEE Transactions on Knowledge and Data Engineering* 28, 11 (2016), 2927–2942. https://doi.org/10.1109/TKDE.2016.2594065