

Efficient Algorithms for Earliest and Fastest Paths in Public Transport Networks

Mithinti Srikanth

Indian Institute of Technology Tirupati
Tirupati, Andhra Pradesh, India
srikanth.mithinti@gmail.com

G. Ramakrishna

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

ABSTRACT

Public transport administrators rely on efficient algorithms for various problems that arise in public transport networks. In particular, our study focused on designing linear-time algorithms for two fundamental path problems: the earliest arrival time (EAT) and the fastest path duration (FPD) on public transportation data. We conduct a comparative analysis with state-of-the-art algorithms. The results are quite promising, indicating substantial efficiency improvements. Specifically, the fastest path problem shows a remarkable 34-fold speedup, while the earliest arrival time problem exhibits an even more impressive 183-fold speedup. These findings highlight the effectiveness of our algorithms to solve EAT and FPD problems in public transport, and eventually help public administrators to enrich the urban transport experience.

Index Terms: Temporal Graph, Fastest Path, Earliest Arrival Path, Edge Scan Dependency Graph, Algorithm Engineering

1 INTRODUCTION

Route optimization on public transportation is crucial for urban public transport administrators involved in route planning. Intelligent route optimization algorithms are being used in transportation software, to enhance traffic flow and reduce environmental impact. These algorithms are used by various stakeholders to address critical questions such as travel time estimation, maximizing city coverage, and efficient urban transport experience. A vital component of public transportation networks is the scheduled timetable information, encompassing vehicle departure and arrival times at stops across various routes. This valuable data set is efficiently handled and represented using temporal graphs.

A *temporal graph* is a weighted directed graph in which departure time and duration time are assigned to each edge. Given a temporal graph G , a source vertex s , and a ready time at source vertex rt , the earliest arrival time problem (EAT) is to find the minimum arrival times of paths from s to all the vertices, in which the departure of each path is at least rt . Given a temporal graph G and a source vertex s , the fastest path duration problem (FPD) is to find minimum journey times from s to all the vertices, where the journey time of a path is the difference between its arrival time and the departure time.

Multiple graph representations are evolved based on the nature of the graph problems, associated applications, and various factors such as computation type and memory access patterns. Broadly, various path problems can be classified into two variants namely single-source and goal-oriented. Single-source problems aim to find values like distance or arrival time from one starting point to all vertices. Goal-oriented problems calculate path measures from a

specific source to a particular target. This paper focuses on single-source EAT and single-source FPD in public transport networks.

To date, edge stream algorithm is the latest algorithm to solve single-source EAT problem using edge-stream representation in public transport networks and real world temporal graphs [4] [11]. Single-pass and traversal based multi-pass algorithms are the best known algorithms to solve single-source FPD problem in real world temporal graphs, by using edge-stream and time-respecting graph (TRG) representations, respectively [11, 18].

In the edge-stream format all the edges of a graph are arranged in an array in non-decreasing order based on their departure time. Algorithms designed on the edge-stream format to solve EAT and FPD perform well due to spatial data locality. The drawback in these algorithms ([4, 11]) is that all edges of the graph are processed independent of the source vertex given in the query time.

The graph traversal based multi-pass TRG algorithm performs better than the single-pass algorithm to solve FPD, due to pruning. In the TRG algorithm, whenever a node (u, t) is visited, all the other nodes (u, t') , where $t' > t$ are visited, and all of their outgoing edges are processed. We observe the following two drawbacks in the TRG approach [18]. i) For every vertex u in G , all the departure nodes (u, t) are connected by a chain of edges, and the chain length is at least the temporal out-degree of u . In public transport networks, the temporal out-degree is quite large when compared to their static out-degree as shown in Table 1. From Fig 1, it is evident that around 45% of the total running time is spent towards processing chain edges, which is a bottleneck. ii) For every static edge (u, v) , all the temporal edges (u, v, t, λ) are processed if u is reached on or before t , whose final effect is due to one temporal edge. In other words, whenever a vertex u is visited at time t , processing (u, v, t', λ') is not required, if there exist another edge (u, v, t'', λ'') such that $t' + \lambda' > t'' + \lambda''$ or $t' < t$.

Data Sets	Temporal Out-Degree		Static Out-Degree	
	Max Out-Degree	Average Out-Degree	Max Out-Degree	Average Out-Degree
Chicago	1315	408	17	3
London	7948	675	7	1.3
Los Angeles	2069	142	7	1.3
Madrid	2940	425	8	1.5
Newyork	1480	521	3	1.2
Paris	83209	2599	61	3
Petersburg	11402	586	22	1.5
Sweden	17740	144	43	2
Switzerland	28315	310	49	2

Table 1: Temporal degree and static degree of public transport networks

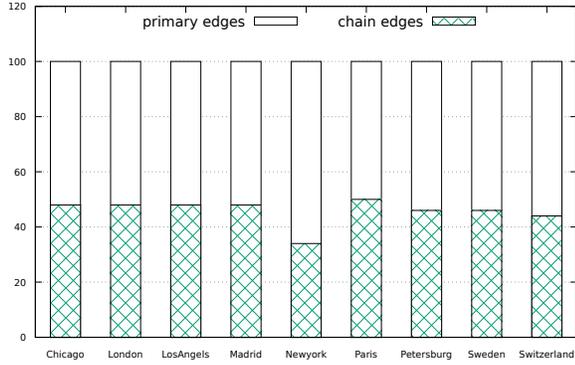


Figure 1: Time spent on chain edges processing in TRG based EAT algorithm

This motivates us to define certain dependencies between temporal edges and process only necessary dependencies to reduce the computations. Fortunately, only such important dependencies are captured in the edge-scan-dependency (ESD) graph [19]. However, in their approach, edges are arranged in a special order based on the associated ESD-graph and ignore the topological structure, to solve EAT problem. We explore structural properties of ESD-graph and use the topology of the ESD-graph to solve EAT and FPD problem, to overcome the drawbacks identified, and beat the existing results. Our key contributions are outlined as follows.

- *Characterization*: We introduce the notion of useful dominating paths and show that there is a one to one mapping between useful-dominating-paths in a temporal graph G and paths in the transformed graph \tilde{G} of G . This helps to eliminate traversal of many unnecessary paths in our algorithms.
- *Data-Structures*: We use the topology of edge-scan-dependency graph data structure to avoid the time validation computation, during the traversal of \tilde{G} , as every path in \tilde{G} corresponds to a time respecting path in G .
- *Algorithms*: For a temporal graph on n vertices and m edges, we devise $O(m + n)$ algorithms to solve the fastest path duration and earliest arrival time problems. In our algorithms, we make sure that every edge is processed at most once.
- *Speedup*: We run our algorithms on the nine real-time public transportation data sets. In practice, we avoid processing many edges. Thus, the fastest path duration algorithm obtains a 34-fold speed up and the earliest arrival path duration algorithm, obtains a 183-fold speed up, compared with the state-of-the-art algorithms.

2 PRELIMINARIES

A temporal graph is a weighted directed graph, in which multiple edges exist between the same pair of vertices, and the edges associate with time information. We use G to denote a temporal graph. Let $V(G)$ and $E(G)$ denote the set of vertices and set of edges, respectively in G . An edge in a temporal graph G is represented by a 4-tuple (u, v, t, λ) , where u and v are the end vertices of the edge, t and λ are positive integers, t denotes the departure time at u , and λ denotes the duration time from u to v ; $t + \lambda$ is considered as

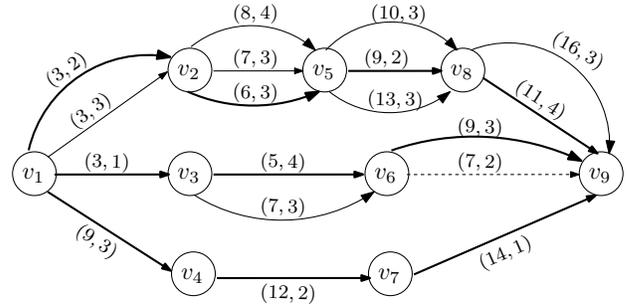


Figure 2: A temporal graph

arrival time at v . A sequence (e_1, e_2, \dots, e_k) of edges in G is a *time respecting path*, if it joins a sequence of vertices and the departure time of every edge is at least the arrival time of the previous edge. Let P be a time respecting in G . Then, $\text{dep}(P)$ and $\text{arr}(P)$ denote the departure and arrival times, respectively. Further, the journey time of P is defined as $\text{arr}(P) - \text{dep}(P)$. Let s and z be two vertices in G . A time-respecting path from s whose departure time is at least the given ready time rt , and the arrival time at z is minimum is referred to as an *earliest arrival path*. Similarly, a time-respecting path from s to z is a *fastest path* if its journey time is minimum over all the paths from s to z . The arrival time of the earliest arrival path and the journey time of the fastest path are known as *earliest arrival time* and *fastest path duration*. In Figure 2, for the given ready time 3, the sequence of $(3,1), (5,4), (9,3)$ edges forms an earliest arrival path from v_1 to v_9 , whose arrival time is 12 and journey time is 9. Further, the sequence of $(9,3), (12,2), (14,1)$ edges is a fastest path from v_1 to v_9 , because its journey time is 6. In the EAT and FPD problems, the goal is to compute the earliest arrival times and fastest path durations from a source vertex to all the vertices.

3 USEFUL DOMINATING PATHS AND ESDG

In this section, we first characterize the earliest arrival paths and fastest paths in temporal graphs. In other words, we define the notion of *useful dominating paths* and show that every earliest arrival path and every fastest path is a useful dominating path. Further, we use the known transformation to convert a temporal graph to an equivalent directed acyclic graph(DAG)[19]. Later, we prove that all the earliest arrival paths and fastest paths are preserved in the transformed DAG, which helps to design efficient algorithms.

Now, we describe about useful dominating paths. A sequence $(v_1, v_2, v_3, \dots, v_k)$ of vertices in G is a *route*, if there is an edge between every two consecutive vertices in the sequence. For a path $P = (e_1, e_2, \dots, e_k)$, and a route $r = (v_1, v_2, v_3, \dots, v_{k+1})$, we say that P *goes through* r , if for each i , $1 \leq i \leq k$, left and right end vertices of e_i are v_i and v_{i+1} , respectively. A sub-path of a path is a *prefix path* if both the paths start at the same vertex. Let $\mathbb{P}(s, z, r, t)$ denote the set of paths that depart at time t from s , go through the route $r = (s = u_1, \dots, u_k = z)$ of vertices and reach z . A path P in $\mathbb{P}(s, z, r, t)$ is a *dominating path* if, for every path Q in $\mathbb{P}(s, z, r, t)$, $\text{arr}(P) \leq \text{arr}(Q)$. A dominating path P in $\mathbb{P}(s, z, r, t)$ is a *useful*

dominating path if for every prefix path P' of P , P' is a dominating path.

In the route $r = (v_1, v_2, v_5, v_8, v_9)$ illustrated in Figure 2, the sequence of edges $(3, 2), (6, 3), (9, 2), (11, 4)$ constitutes a path denoted as P . Additionally, the edges $(3, 3), (6, 3), (9, 2), (11, 4)$ form a distinct path referred to as Q , and the edges $(3, 4), (7, 3), (10, 3), (16, 3)$ form yet another path called R . Since, $\text{arr}(R) > \text{arr}(P)$, R does not meet the criteria of a dominating path. Upon considering the prefix paths P' and Q' , of P and Q , respectively on route $r = (v_1, v_2)$, it becomes apparent that $\text{arr}(Q') > \text{arr}(P')$. Consequently, it is established that P is a useful dominating path, whereas Q is dominating, but not a useful dominating path.

LEMMA 3.1. *If there exists a path from s to z that departs at time t on a route r , then there exists a useful dominating path from s to z that starts at time t on the route r .*

PROOF. Let P be a dominating path that departs at t , on route $r = (s = u_1, \dots, u_k = z)$. We claim that there exists a useful dominating path on r . In the base case $k = 1$, and thus the claim holds true, because the number of prefix paths of P is one. By the induction hypothesis, the claim holds true for $k - 1$. In other words, if there is a path from u_1 to u_{k-1} that starts at time t on route r , then there exists a useful dominating path Q from u_1 to u_{k-1} that departs at t on route r . Let Q' be the sub path of the dominating path P from u_1 to u_{k-1} . Now we shall replace Q' of P with Q and the resultant dominating path is useful. \square

If all the prefix paths of a dominating path on a route are dominating, then a prefix path of any prefix path is dominating. This results the following corollary.

COROLLARY 3.2. *Every prefix path of a useful dominating path is a useful dominating path.*

An earliest arrival path is a dominating path on a route. Similarly, a fastest path is a dominating path on a route. Also, the departure times (and arrival times) of both dominating and useful dominating paths on the same route are equal. From these observations along with Lemma 3.1, we have the following corollaries. These corollaries helps to design a pruning strategy, by which exploring the useful dominating paths in temporal graphs will be sufficient to find earliest arrival and fastest paths.

COROLLARY 3.3. *For a temporal graph G , a ready time rt , and a source vertex s , let P be an earliest arrival path from s to z such that $\text{dep}(P) \geq rt$. Then there exists a useful dominating path Q from s to z such that $\text{route}(Q) = \text{route}(P)$, $\text{dep}(Q) = \text{dep}(P)$ and $\text{arr}(Q) = \text{arr}(P)$.*

COROLLARY 3.4. *Let P be a fastest path from s to z in a temporal graph G . Then there exists a useful dominating path Q from s to z such that $\text{route}(Q) = \text{route}(P)$, $\text{dep}(Q) = \text{dep}(P)$ and $\text{arr}(Q) = \text{arr}(P)$.*

Moving forward, we describe a graph transformation to preserve dominating paths. A temporal graph G can be converted to an equivalent edge scan dependency graph (ESDG) \tilde{G} , in which all the edges of G are treated as vertices or *nodes* in \tilde{G} , and edges or *dependencies* are added between the nodes based on certain constraints. Ni et al. have developed this transformation to solve EAT in the parallel setting [19]. In this work, we propose one to

mapping between useful dominating paths in G to paths in \tilde{G} in the following lemma, and further use this characterization to design efficient algorithms.

Now, we shall look at the transformation from a temporal graph to the corresponding esdg. For a temporal graph $G = (V(G), E(G))$, the node set $V(\tilde{G})$, and the edge set $E(\tilde{G})$ are denoted as follows: $V(\tilde{G}) = \{v_e \mid e \in E(G)\}$, $E(\tilde{G}) = \{(v_e, v_f) \mid e = (u, v, \alpha, \omega), f = (v, w, \alpha', \omega') \in E(G), \text{ and no edge } (v, w, \alpha'', \omega'') \text{ exists such that } \alpha'' \geq \omega, \text{ and } \omega'' < \omega'\}$. For each edge $e = (u, v, t, \lambda)$ in G , there is a node v_e in \tilde{G} , and we define four values namely left vertex, right vertex, departure time and arrival time of node v_e as follows: $\text{left}(v_e) = u$, $\text{right}(v_e) = v$, $\text{dep}(v_e) = t$, $\text{arr}(v_e) = t + \lambda$. For the graph shown in Figure 2, the corresponding edge-scan-dependency graph is illustrated in Figure 3, in which each node of \tilde{G} is associated with a departure time and an arrival time.

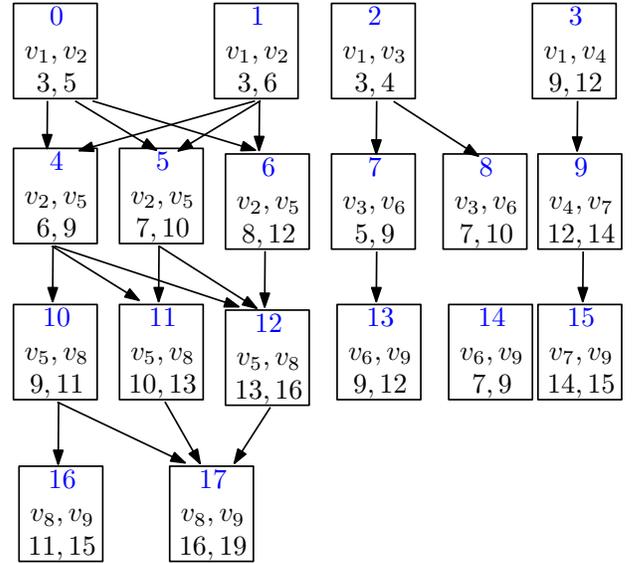


Figure 3: An edge-scan-dependency graph

LEMMA 3.5. *A sequence e_1, \dots, e_k of k edges in G is a useful dominating path if and only if a sequence v_{e_1}, \dots, v_{e_k} of k vertices in \tilde{G} is a path. Further, the journey time of the path from e_1 to e_{k-1} in G is equal to the journey time of the path from v_{e_1} to $v_{e_{k-1}}$ in \tilde{G} .*

PROOF. Let $e_{k-1} = (u, v, \alpha, \omega)$ and $e_k = (v, w, \alpha', \omega')$, and we use this notation in the following proofs. We first prove the forwarding direction of this lemma using induction on k . In the base case, k equals to 1, and e_1 is a useful dominating path. Because, each edge in G is represented as a node in ESGD graph \tilde{G} , there is a path of length 0 from v_e to v_e in \tilde{G} . Thus the base case holds true. Let P be the path formed with the sequence e_1, \dots, e_k of k edges. Let P' be the subpath of P on the first $k-1$ edges. P' is useful dominating path due to Corollary 3.2. Due to induction hypothesis, there is a path Q from v_{e_1} to $v_{e_{k-1}}$ in \tilde{G} . Because P is a dominating path, there does not exist any edge $(v, w, \alpha'', \omega'')$, st. $\alpha'' \geq \omega$ and $\omega'' < \omega'$. Hence, we would have added an edge between e_{k-1} and e_k in \tilde{G} . Thus,

the concatenation of path Q from v_{e_1} to $v_{e_{k-1}}$ and edge $(v_{e_{k-1}}, v_{e_k})$, results a path from v_{e_1} to v_{e_k} in \tilde{G} .

Now, we prove the backward direction of the lemma, using induction on k . The base case holds true when $k = 1$, because there is a one to one mapping between edges in G and nodes in \tilde{G} . Let P be the path formed with the sequence v_{e_1}, \dots, v_{e_k} of k nodes. Let P' be the subpath of P on the first $k - 1$ edges. Due to induction hypothesis, there is a useful dominating path Q from e_1 to e_{k-1} in G . Since an edge is added from e_{k-1} to e_k in \tilde{G} , there does not exist any edge $(v, w, \alpha'', \omega'')$, st. $\alpha'' \geq \omega$ and $\omega'' < \omega'$. Therefore, e_1, \dots, e_{k-1}, e_k is a useful dominating path in G .

The departure times of $v(e_1)$ and e_1 are equal, and the arrival times $v(e_k)$ and e_k are equal, from the transformation. Thus the journey times of both paths provided in the second part of the lemma are same. \square

In this section, we first provide the pseudo-code to solve EARLIEST ARRIVAL TIME problem in Algorithm 1, and prove the correctness in Theorem 3.6. Later, the pseudo-code of Algorithm 1 is enhanced in Algorithm 2, to bound the running time.

The key idea in Algorithm 1 is to explore from those nodes in \tilde{G} that correspond to edges in G , such that their left end vertex is the source vertex, and the departure time is at least rt . During the exploration, we identify all the reachable nodes and update the arrival times of their right end vertices, if the new arrival time is better than the existing one.

THEOREM 3.6. *Given an ESDG \tilde{G} of a temporal graph G , a source vertex s in G and a ready time rt , Algorithm 1 correctly computes the earliest arrival time from s to every vertex in G .*

PROOF. Let z be a vertex in G such that $z \neq s$. We now prove that, after all the iterations of Algorithm 1 are over, $\text{eat}[z]$ is equal to the earliest arrival time from s to z . In Algorithm 1, for every node x in \tilde{G} such that $\text{left}(x) = s$ and $\text{dep}(x) \geq rt$, we traverse from x , go through all the paths to find all reachable nodes in \tilde{G} . The arrival times of all these paths are considered and stored the minimum one in Line 5. Also, these paths in \tilde{G} precisely correspond to those useful dominating paths from s to z in G whose departure time is at least rt , due to Lemma 3.5. From Corollary 3.3, it is sufficient to consider useful dominating paths from s to z that depart at time at least rt , to obtain an earliest arrival path from s to z , for the given departure time rt . Thus the theorem holds true. \square

The correctness of Algorithm 1 to solve EARLIEST ARRIVAL TIME problem follows from Theorem 3.6. Moving forward, Algorithm 1 is enhanced to Algorithm 2 by processing the vertices and edges of \tilde{G} at most once, across multiple breadth first search traversals to bound the running time. This does not disturb the correctness of the algorithm, because processing a vertex x in \tilde{G} multiple times does not change the arrival time of $\text{right}(x)$.

Algorithm 2 works as follows. Given a source vertex s , ready time rt at the source, and an ESDG graph \tilde{G} , the algorithm initializes the earliest arrival time $\text{eat}[z]$ for each vertex z in $V(G)$. The earliest arrival time represents the minimum arrival time to reach vertex z from the source vertex s . During the initialization phase in Line 2, all vertices, except the source, are set to have an earliest arrival time of ∞ , while the source vertex is set to rt . Also, $\text{visited}[x]$ for every

Algorithm 1: Earliest Arrival Time in Temporal Graph

Input: A source vertex s , ready time at source rt and edge scan dependency graph \tilde{G} of a temporal graph G .
Output: For each vertex z in G , the earliest arrival time from s to z .

```

1 for each vertex  $z$  in  $V(G) - s$  do  $\text{eat}[z] = \infty$ ;
2  $\text{eat}[s] = rt$ ;
3 for each node  $x$  in  $\tilde{G}$  such that  $\text{left}(x) = s$  and  $\text{dep}(x) \geq rt$  do
4   | for each path  $p$  from  $x$  to a node  $y$  in  $\tilde{G}$  do
5   |   |  $\text{eat}[\text{right}(y)] = \min\{\text{eat}[\text{right}(y)], \text{arr}(y)\}$ ;
6   | end
7 end

```

Algorithm 2: Earliest Arrival Time in Temporal Graph

Input: A source vertex s , ready time at source rt and edge scan dependency graph \tilde{G} of a temporal graph G .
Output: For each vertex z in G , the earliest arrival time from s to z .

```

1 for each vertex  $z$  in  $V(G) - s$  do  $\text{eat}[z] = \infty$ ;
2  $\text{eat}[s] = rt$ ;
3 for each node  $x = (u, v, t, \lambda)$  in  $V(\tilde{G})$  do  $\text{visited}[x] = \text{false}$ ;
4 for each node  $x = (u, v, t, \lambda)$  in  $V(\tilde{G})$  such that  $\text{left}(x) = s$ 
   and  $\text{dep}(x) \geq rt$  do
5   |  $q.\text{insert}(x)$ ;  $\text{visited}[x] = \text{true}$ ;
6   | while  $(|q| \geq 1)$  do
7   |   |  $x = q.\text{pop}()$ ;
8   |   |  $\text{eat}[\text{right}(x)] = \min(\text{eat}[\text{right}(x)], \text{arr}(x))$ ;
9   |   | for each neighbor  $y$  of  $x$  in  $V(\tilde{G})$  such that
   |   |   |  $\text{visited}[y] = \text{false}$  do
   |   |   |   |  $q.\text{insert}(y)$ ;  $\text{visited}[y] = \text{true}$ 
   |   |   | end
10  |   | end
11  | end
12  | end
13 end

```

node x in \tilde{G} is set to *false* in Line 3, to indicate that all nodes in the beginning are not visited. We now, go through all the nodes x in \tilde{G} such that $\text{left}(x) = s$ and $\text{dep}(x) \geq rt$, and perform a breadth first kind of traversal from x , as follows. The queue Q is also initialized with x and $\text{visited}[x]$ is updated with true. After the initialization, the algorithm proceeds to process edges by deleting them one by one from the queue and perform relaxation in Line 8. Further, all the unvisited neighbours of x are added to the queue. These two steps are repeated until the queue becomes empty. Finally, for every vertex z in G , $\text{eat}[z]$ holds the earliest arrival time to reach z .

4 EFFICIENT ALGORITHM FOR FASTEST PATH DURATION

In this section, we propose efficient algorithms for computing the fastest path duration from the given source vertex to all the vertices at high-level in Algorithm 3 and the complete details in Algorithm 4, and the algorithm correctness is proved in Theorem 4.1.

The fundamental idea in Algorithm 3 is to perform a graph traversal from those nodes in \tilde{G} that correspond to the outgoing edges of a source vertex in G . During each graph traversal phase, we propagate the starting time to all the reachable nodes and update the journey times of their right end vertices, if the new journey time is better than the existing one.

Algorithm 3: 1 - All Fastest Path Algorithm using ESDG

Input: A source vertex s , and an edge scan dependency graph \tilde{G} of a temporal graph G

Output: For each vertex z in G , the fastest duration from s to z .

```

1 for each vertex  $z$  in  $G \setminus s$  do journey[ $z$ ] =  $\infty$ ;
2 journey[ $s$ ] = 0;
3 for each node  $x$  in  $\tilde{G}$ , such that  $\text{left}(x) = s$  do
4   for each path  $p$  from  $x$  to a node  $y$  in  $\tilde{G}$  do
5     journey[right( $y$ )] =
6     min{journey[right( $y$ )], arr( $y$ ) - dep( $x$ )};
7   end
8 end

```

THEOREM 4.1. *Given an ESDG \tilde{G} of a temporal graph G , a source vertex s in G , Algorithm 3 correctly computes the fastest path duration from s to every vertex z in G .*

PROOF. Let z be a vertex in G such that $z \neq s$. We now show that, after all the iterations of Algorithm 3 are over, journey[z] is equal to the journey time of a fastest path from s to z . In Algorithm 3, for every node x in \tilde{G} such that $\text{left}(x) = s$, we traverse from x , go through all the paths to find all reachable nodes in \tilde{G} . Out of all these paths, let us observe all those paths that end at y , such that $\text{right}(y) = z$. The journey times of all these paths are considered and stored the minimum one in journey[z] in Line 5. Also, these paths in \tilde{G} precisely correspond to the useful dominating paths from s to z in G due to Lemma 3.5. From Corollary 3.4, it is sufficient to consider useful dominating paths from s to z to obtain a fastest path from s to z . Thus the theorem holds true. \square

The correctness of Algorithm 3 is formally established in Theorem 4.1.

Turning our attention to enhance the efficiency, we avoid processing the same node multiple times. For instance, consider two nodes, x_i and x_j , in \tilde{G} with $\text{left}(x_i) = \text{left}(x_j) = s$, selected during iterations i and j of the outer loop in Line 3 of Algorithm 3. Let y be a reachable node from both x_i and x_j , with P_i denoting the path from x_i to y and P_j representing the path from x_j to y . If $\text{dep}(x_j) \leq \text{dep}(x_i)$, then it follows that $\text{journey}(P_j) \geq \text{journey}(P_i)$ since $\text{arr}(P_i) = \text{arr}(P_j)$. This insightful observation leads to a critical optimization: a node processed in the i^{th} iteration of the outer loop need not be processed again in the j^{th} iteration if $\text{dep}(x_j) \leq \text{dep}(x_i)$, as the journey from s to $\text{right}(y)$ does not decrease. To harness the advantages of this optimization, we traverse nodes in the edge scan dependency graph whose left vertex is the source vertex, by ordering them in the non-increasing order based

Algorithm 4: 1 - All Fastest Path Algorithm using ESDG

Input: A source vertex s , and an edge scan dependency graph \tilde{G} of a temporal graph G

Output: For each vertex z in G , the fastest duration from s to z .

```

1 for each vertex  $z$  in  $G \setminus s$  do journey[ $z$ ] =  $\infty$ ;
2 journey[ $s$ ] = 0;
3 for each node  $x$  in  $\tilde{G}$  do  $st[x] = -1$ ;
4 for each node  $x$  in  $\tilde{G}$  such that  $\text{left}(x) = s$  and decreasing
   order of  $x.t$  do
5    $q.\text{insert}(x)$ ;  $st[x] = x.t$ ;
6   while ( $|q| \geq 1$ ) do
7      $x = q.\text{pop}()$ ;
8     journey[ $x.v$ ] =  $\min(\text{journey}[x.v], \text{arr}(x) - st[x])$ ;
9     for each neighbor  $y$  of  $x$  in  $\tilde{G}$  such that  $st[y] == -1$ 
10    do  $st[y] = st[x]$ ;  $q.\text{insert}(y)$ ;
11 end

```

on their departure times. This avoids the redundant computation of processing the same nodes multiple times.

We now provide the detailed description of our algorithm to solve FASTEST PATH DURATION, whose pseudo-code is given in Algorithm 4. For each vertex z in G , the variable journey[z] stores the journey time, representing the fastest path duration from the source vertex s to z . For each node x in \tilde{G} , we use $st[x]$ to store the starting time of a journey that departs as late as possible from s and reaches node x . We use q to denote the queue data structure, which helps to perform breadth first kind of traversal. During the initialization phase (lines 1-3), we set the fastest path duration of all vertices, except the source vertex, as ∞ . The fastest path duration of the source vertex is set to 0. Also, the starting time for all the nodes of \tilde{G} is set to -1, to indicate that none of them are visited in the beginning. We now go through each node x in \tilde{G} such that $x.u = s$ in decreasing order based on the departure time of the nodes, and perform breadth first search kind of traversal to identify reachable nodes from x as follows. We insert x in the queue q , and initialize $st[x]$ with its departure time. As long as the queue is not empty, we extract a node x from the queue and update journey[$x.v$] if $x.t + x.\lambda - st[x]$ is lesser than journey[$x.v$]. Afterwards, we insert each neighbor y of x into the queue q and update $st[y]$ with $st[x]$, if y has not been visited yet. Finally, journey[z] holds the fastest path duration from s to any vertex z due to Theorem 4.1.

Time Complexity Analysis. We perform breadth first search kind of traversals from multiple vertices of \tilde{G} , in Algorithm 1 and Algorithm 2. Although multiple breadth first search traversals are performed, we make sure that each vertex of \tilde{G} is inserted in the underlying queue and process their incident edges at most once, with the help of visited[] array and st[] array, in the respective algorithms. Thus the asymptotic running times of our algorithms is $O(|V(\tilde{G})| + |E(\tilde{G})|)$. From the ESD graph construction, $|V(\tilde{G})| = |E(G)|$. For each temporal edge $e = (u, w, t, \lambda)$ of G , the out-degree of v_e in \tilde{G} is equal to the out-degree of w in the static road network associated with G . Also, the average out degree of

a vertex in \tilde{G} is equal to the average out degree of the static road network associated with a public transport network G , which is denoted by Δ . Consequently, the asymptotic running time of our algorithms is $O(m \times \Delta + n)$. This value of Δ turned out to be a small constant in all the real world public transport networks. From Table 1, we can observe that the max degree and average degree of vertices in real world transport networks are 61 and 3, respectively. Eventually, the time complexity of our algorithms is bounded by $O(m + n)$.

5 IMPLEMENTATION DETAILS

In this section, we discuss the implementation details of the ESD-graph data structure, and present various optimizations in the context of implementing Algorithm 2 and Algorithm 4.

ESD-graph data structure. We process real-time public transportation data available in the General Transit Feed Specification (GTFS) format, and transform to an ESD graph, as described in Section 3. We construct ESD-graph data structure from a given temporal graph using the pre-processing algorithm illustrated in [19]. We store an ESD graph \tilde{G} in our data-structure (offset, neighbour, left, right, departure, duration) whose parts are described below. The topology of \tilde{G} is captured using Compressed Sparse Row (CSR) format. CSR ensures fast access to neighbour information, a critical need for our algorithms using two arrays `offset[]` and `neighbours[]`. For each vertex v in \tilde{G} , the neighbours of v are located from position `offset[v]` to `offset[v + 1] - 1` in `neighbours[]`. For each vertex v_e in \tilde{G} , we maintain the following four attributes: `left[v_e]` denotes the left vertex of e , `right[v_e]` denotes the right vertex of e , `dep[v_e]` denotes the departure time of e , and `arr[v_e]` denotes the arrival time of e . This format excels in rapid neighbour information retrieval and graph operations, ideal for managing public transportation network data. Notably, this format is highly space-efficient and accessing neighbour information is achieved in constant time. For each vertex v in G , we maintain a sequence of vertices in \tilde{G} , which are correspond to the outgoing edges of v . This helps to retrieve the necessary vertices in \tilde{G} in Line 4 of Algorithm 2 and Line 4 of Algorithm 4, efficiently.

Optimizations. We describe various optimizations that improve the running time of our algorithms in practice. In Line 8 of Algorithm 2, if the earliest arrival time of a vertex is not updated, then we can ignore exploring its neighbours. This optimization is based on the following observation. If e and e' corresponds to incoming edges of a vertex in G such that their arrival times are same, then the outgoing neighbours of v_e and $v_{e'}$ are same in \tilde{G} .

We employ a bit optimization technique in Algorithm 2. In particular, an array B of n bits are used, and utilize a single bit of B rather than one byte, to represent the visited status of a vertex. This optimization involves resetting all the bits of B at the beginning to initialize all the vertices of \tilde{G} as unvisited. Setting an individual bit in B helps to mark a vertex as visited. These two operations can be performed in constant time. This optimization helps to achieve a notable reduction in memory consumption and an improvement in processing speed.

6 EXPERIMENTS

In this section, we discuss the experimental setup and various experiments carried out on Algorithms 2 and 4, and highlighting the speedup achieved over state-of-the-art algorithms.

Technical Specifications and Data sets. 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. We have used nine different public transport network data sets [25, 26] for our experiments. The statistics for each data set are given in Table 2.

Data Sets	$ V(G) $	$ E(G) = V(\tilde{G}) $	$ E(\tilde{G}) $
Chicago	240	98157	44907
London	20843	14064967	12103649
Los Angels	13975	1979340	2320947
Madrid	4689	1994688	2753161
New York	987	514390	499713
Paris	411	1068284	50965
Peters burg	7573	4437010	6038003
Sweden	45727	6567745	12144520
Switzerland	29870	9261315	12147435

Table 2: Data Set Statistics

6.1 Performance of the Earliest Arrival Time Algorithm

We implemented the state-of-the-art algorithms and our algorithm to solve EAT problem in C++. Specifically, we examined two state of the art algorithms proposed in [11, 18]. The first earliest time algorithm, abbreviated as EDGE-STREAM-EAT, is based on an edge stream, in which the temporal edges are relaxed in non-decreasing order, based on their departure time [11]. The second earliest arrival time algorithm is based on a time respecting graph abbreviated as TRG-EAT, is obtained from the fastest path algorithm in [18] with minor changes.

We run state-of-the-art and proposed algorithms on 100 generated random queries, each consisting of two values: a source vertex and a ready time. The source vertices are randomly selected from 0 to n , where n denotes the number of vertices in the underlying graph, and the corresponding ready times are chosen randomly within the range of 0 to 100. We then use these generated queries to run all three algorithms on nine public transportation data sets, measuring the average query running time in milliseconds. Table 3 presents the average running times of our proposed algorithm. The speedups of our approach in comparison to the state-of-the-art algorithms edge-scan based algorithm [11] and TRG based algorithm [18] shown in the Fig. 4.

Computing the earliest arrival time using our approach, we achieved $183\times$ maximum and $24\times$ average speedup over algorithm [11] and $48\times$ maximum and $24\times$ average speedup over [18].

Earliest Arrival Time Algorithms Execution Time in milliseconds			
Data Sets	Edge-Stream EAT [11]	TRG EAT [18]	Our Approach Algorithm 2
Chicago	0.79	1.13	0.07
London	110.23	1,266.55	35.39
Los Angels	15.73	192.23	9.73
Madrid	14.51	240.42	5.42
New York	3.62	10.67	0.85
Paris	7.32	1.92	0.04
Petersburg	32.27	115.73	6.29
Sweden	51.93	270.18	31.36
Switzerland	70.70	150.34	15.85

Table 3: Run time analysis

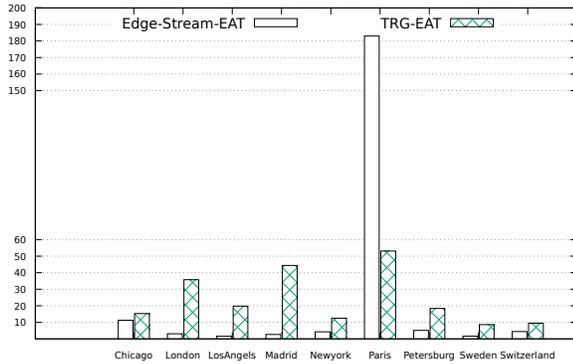


Figure 4: Speed up of Algorithm 2 w.r.t the state of the art algorithms [11, 18]

6.2 Performance of the Fastest Path Duration Algorithm

The state-of-the-art algorithms to solve FPD from [11] and [18] are abbreviated as EDGE-STREAM-FPD and TRG-FPD, respectively. The experiment involved generating 100 queries, each comprising 100 source vertices selected randomly from the range 0 to n , where n denotes the number of vertices in the underlying graph. These same queries were employed as input for the fastest path duration Algorithm 4 along with two state-of-the-art fastest path duration algorithms: the edge stream-based algorithm [11] and the TRG-based algorithm [18]. The evaluation was conducted on nine real world public transportation data sets as mentioned in the Table 2, and the average running time for a single query was measured in milliseconds. The resulting average running times for the proposed algorithms are presented in Table 4, while the speedups achieved by our approach compared to the state of the art algorithms are depicted in Fig. 5.

Computing the fastest path duration using our approach, we achieved 6 \times average speedup over two base line algorithms [11] and [18] with maximum 21 \times speedup over algorithm [11] and

34 \times speedup over algorithm proposed by [18]. In the public transportation application scenario, the ESD graph data-structure is constructed once, and queried multiple times. For the data-sets shown in Table 2, the pre-processing time required to construct the corresponding ESD-graph is bounded by one minute, in practice. Hence our algorithms and implementation focus to reduce the query execution time for a given source vertex, while solving EAT and FPD problems.

Fastest Path Duration Algorithms Execution Time in milliseconds			
Data Sets	EDGE-STREAM FPD [11]	TRG FPD [18]	Our Approach Algorithm 4
Chicago	1.90	1.83	0.52
London	6885.24	2880.80	1576.75
Los Angels	874.91	481.14	179.56
Madrid	1756.52	530.86	212.09
New York	91.72	18.81	4.44
Paris	1.67	7.77	0.23
Petersburg	1305.77	396.88	198.10
Sweden	1346.45	975.65	469.56
Switzerland	652.72	489.88	182.03

Table 4: Run time analysis

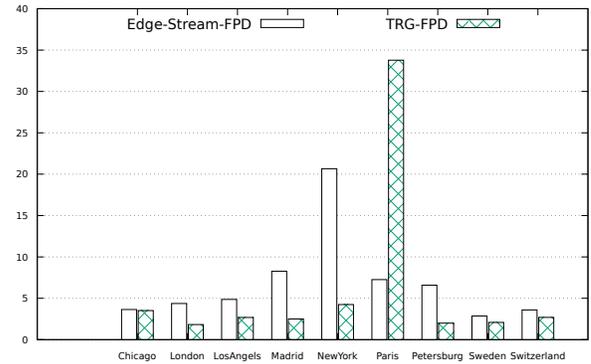


Figure 5: Speed up of Algorithm 4 w.r.t the state of the art algorithms [11, 18]

6.3 Key Insights

Our earliest arrival time algorithm process at most 2% of the edges and fastest path duration algorithm process at most 70% of the edges, as shown in Figure 6 and Figure 7. This is the key reason to the beat running time of the existing algorithms in practice.

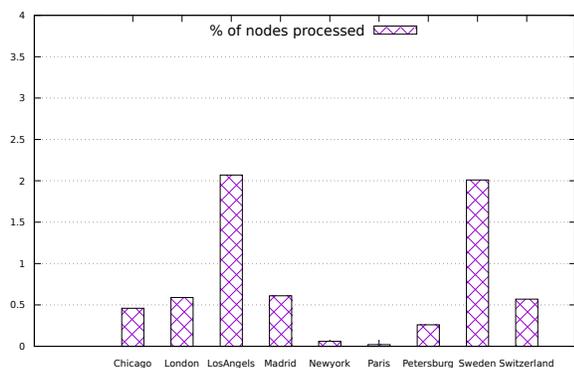


Figure 6: % of nodes processed by Algorithm 2

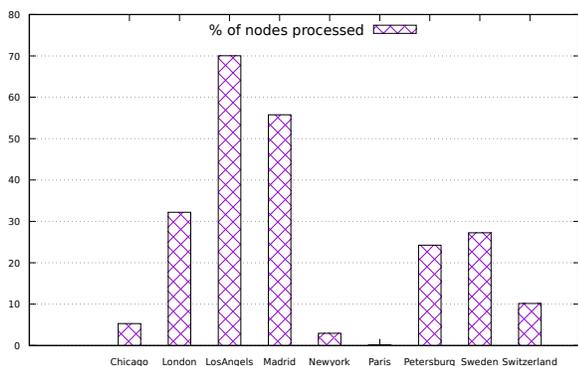


Figure 7: % of nodes processed by Algorithm 4

Data Sets	Our Approach Algorithm 2	
	Running Time (ms)	# Nodes Processed
Paris	0.04	210
Newyork	0.85	326
Chicago	0.07	456
Petersburg	6.29	11430
Madrid	5.42	12238
Los Angels	9.73	41069
Switzerland	15.85	52832
London	35.39	82976
Sweden	31.36	132250

Table 5: Average number of nodes in $E(G)$ processed in edge-scan-dependency-graph

Data Sets	Our Approach Algorithm 4	
	Running Time (ms)	# Nodes Processed
Paris	0.23	1517
Chicago	0.52	5931
Newyork	4.44	19779
Switzerland	182.03	1127507
Los Angels	179.56	1308683
Petersburg	198.1	1615717
Madrid	212.09	1655109
Sweden	469.56	2787875
London	1576.75	6232547

Table 6: Average number of nodes in $E(G)$ processed in edge-scan-dependency-graph

The percentage of edges of G (nodes of \tilde{G}) processed by Algorithm 2 and Algorithm 4 are shown in Figure 6 and Figure 7, respectively. For most of the data sets, the execution times of Algorithm 2 and Algorithm 4 are proportional to the number of nodes processed in the respective algorithms. For instance, the execution time of Algorithm 2 on data sets Sweden and London is high, as the number of nodes processed is higher. Similarly, the running time of Algorithm 4 on data-sets Sweden, Switzerland, and London are high, because the number of nodes being explored are higher. These insights can be observed from Table 5 and Table 6.

Many public transport administrators explore various insights on their cities and suggest the public to start a journey from a suitable time, that minimize the duration time. From our experiments, on many data-sets, we observe that the suitable time to start a journey is around 6 AM or 3 PM, to minimize the journey time.

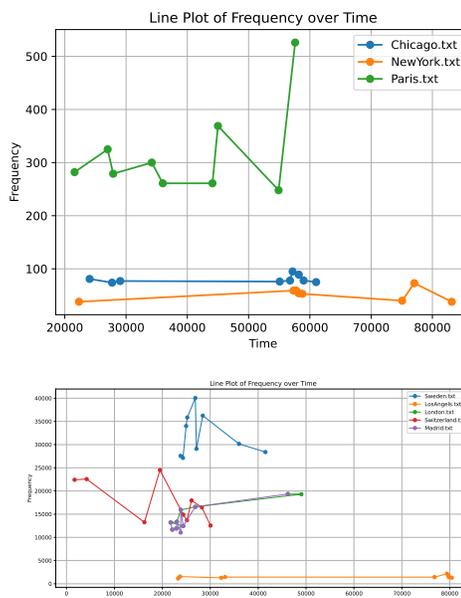


Figure 8: Frequencies of starting times

7 APPLICATIONS

Our Algorithms helps to solve various path problems in various domains that help public transport administrator.

- *How much time is required to visit any place in the city from the given source location?*

This question is addressed by calculating the fastest path duration to reach the farthest or longest location from the provided source location. This information is beneficial for public transport administrators in determining the time needed to reach any place in the city from a given source location. It aids in planning efficient routes and optimizing travel time.

- *Coverage Analysis - How many places can be covered in k hours from the given source location?*

Administrators can evaluate the number of places that can be covered within a designated time frame. For instance, they may want to ascertain how many locations can be visited within k hours from a specified source point. This analysis proves valuable for resource allocation and optimizing service coverage. This challenge is addressed by incorporating a modification in Algorithm 4, terminating each phase if the duration surpasses k hours.

- *Percentage Coverage - How much time is required to cover $k\%$ of the city from the given source location?*

Administrators can ascertain the time needed to cover a specific percentage of the city, such as determining the duration required to cover $k\%$ of the city from the provided source location. This insight aids in understanding the reach and accessibility of services across different areas. Addressing this challenge involves incorporating a termination condition in each phase of the Algorithm 3. Termination occurs when the coverage of a specific phase surpasses $k\%$. The algorithm monitors the maximum journey duration in each phase, and identify the smallest one, known as the global min-journey time. A local phase can be terminated if the local journey time exceeds either the global min-journey time or $k\%$ coverage in the current phase. Additionally, the global min journey time is updated whenever a better local journey time is achieved in the current phase.

8 RELATED WORK

The goal-oriented variants of various problems are extensively explored in the context of public transport networks. In particular, multiple algorithms such as RAPTOR, transfer patterns, connection scan accelerated, and trip-based have been designed to extract paths ranging from earliest arrival and profile search to multi-criteria paths [5–7, 20, 21]. When dealing with real world temporal graphs, the goal-oriented fastest paths can be retrieved using indexing techniques TTL and TOP-CHAIN [22, 23]. All these algorithms are primarily targeted for goal-oriented and not for single-source variant.

Our focus is now on single-source variants of EAT and FPD. Xuan et.al have designed a vertex centric algorithm to solve single-source EAT problem [24]. Later, this is improved using edge stream representation and the associated edge centric algorithm [11]. Similarly, FPD is targeted using edge centric [11] and vertex centric algorithms on the transformed graphs. In particular, many graph transformations [11, 13, 18] are developed to solve single-source shortest and

fastest path problems. In all these transformations, a temporal graph is transformed to an equivalent time respected graph, in which the departure and arrival times of temporal edges are treated as vertices and added edges to capture the necessary dependencies. The key idea in these works is to reduce the number of vertices and edges in the transformed graphs. In this paper, we have compared our results with edge-stream and TRG algorithms, which are the state-of-the-art techniques to solve EAT and FPD problems.

Due to the wide range of applications, building efficient solutions on public transport networks has received attention from both traditional and machine learning algorithms [1–3]. Recent research on public transportation encompasses diverse areas. Letelier et al. (2023) focused on compacting large public transport data [14]. Dahlmanns et al. (2023) optimized transportation networks considering congestion [15], while Cao et al. (2023) sought to enhance public transportation quality through dynamic bus departure times [16]. Drabicki et al. highlights the significant impact of capacity-constrained models on transportation outcomes [17]. These studies contribute to data efficiency, network optimization, and service quality improvements. Machine learning algorithms are used to find important features in the public transport data, influencing whether a vehicle stops on time or late [2]. Also, the prediction of arrival times of public transport vehicles is well studied and useful in daily routine [1].

9 CONCLUSION

In this research, we addressed key path finding challenges in public transportation networks by developing efficient near linear-time algorithms for the earliest arrival time and fastest path duration problems. Utilizing edge-scan-dependency graphs, our approach significantly outperformed existing methods, evidenced by a 34-fold speedup in FPD and an unprecedented 183-fold improvement in EAT. Key to our success was the novel use of useful dominating paths and edge-scan-dependency graph data structures, ensuring minimal edge processing. These advancements were empirically validated on real-world datasets, demonstrating not only theoretical innovation but also practical effectiveness in urban transit systems. Our work marks a significant step forward in optimizing public transportation routes, offering powerful tools for transit planners and setting a new standard in algorithmic solutions for urban mobility challenges.

REFERENCES

- [1] D. Panowski and T. Zaharia, "Real-Time Public Transportation Prediction with Machine Learning Algorithms," in *2020 IEEE International Conference on Consumer Electronics (ICCE)*, Las Vegas, NV, USA, 2020, pp. 1–4. doi: 10.1109/ICCE46568.2020.9043077.
- [2] M. Alicea, A. P. McDonald, C. Tang, and J. Yang, "Exploring the Application of Machine Learning Algorithms to the City Public Bus Transport," in *2020 IEEE 14th International Conference on Big Data Science and Engineering (BigDataSE)*, Guangzhou, China, 2020, pp. 22–27. doi: 10.1109/BigDataSE50710.2020.00011.
- [3] T. Liu, W. Ji, K. Gkiotsalitis, and O. Cats, "Optimizing public transport transfers by integrating timetable coordination and vehicle scheduling," *Computers & Industrial Engineering*, vol. 184, p. 109577, 2023. doi: <https://doi.org/10.1016/j.cie.2023.109577>.
- [4] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, "Intriguingly Simple and Fast Transit Routing," in *Experimental Algorithms*, V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, Eds. Springer Berlin Heidelberg, 2013, pp. 43–54. isbn: 978-3-642-38527-8.
- [5] H. Bast et al., "Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns," in *Algorithms - ESA 2010, 18th Annual European Symposium, Liverpool, UK, September 6-8, 2010. Proceedings, Part I*, ser. Lecture Notes in

- Computer Science, vol. 6346. Springer, 2010, pp. 290-301. doi: 10.1007/978-3-642-15775-2_25.
- [6] D. Delling, T. Pajor, and R. F. Werneck, "Round-based public transit routing," *Transportation Science*, vol. 49, no. 3, pp. 591-604, 2015.
- [7] S. Witt, "Trip-Based Public Transit Routing," in *Proc. of the ESA -Algorithms*, Barcelona, Spain, 2015, pp. 1025-1036.
- [8] R. Chen and C. Gotsman, "Efficient fastest-path computations for road maps," *Computational Visual Media*, vol. 7, pp. 267-281, 2021.
- [9] S. Ahmadi et al., "A fast exact algorithm for the resource constrained shortest path problem," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 14, 2021, pp. 12217-12224.
- [10] R. El Shawi, J. Gudmundsson, and C. Levcopoulos, "Quickest path queries on transportation network," *Computational Geometry*, vol. 47, no. 7, pp. 695-709, 2014.
- [11] H. Wu et al., "Efficient algorithms for temporal path computation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 11, pp. 2927-2942, 2016.
- [12] A. Jain and S. K. Sahnii, "Algorithms for optimal min hop and foremost paths in interval temporal graphs," *Applied Network Science*, vol. 7, no. 1, p. 60, 2022.
- [13] P. Zschoche et al., "The complexity of finding small separators in temporal graphs," *Journal of Computer and System Sciences*, vol. 107, pp. 72-92, 2020.
- [14] B. Letelier et al., "Compacting Massive Public Transport Data," in *International Symposium on String Processing and Information Retrieval*, Pisa, Italy, 2023, pp. 310-322.
- [15] M. Dahlmanns, F. Kaiser, and D. Withhaut, "Optimizing the geometry of transportation networks in the presence of congestion," *Physical Review E*, vol. 108, no. 4, p. 044302, 2023.
- [16] S. Cao, S. A. Thamrin, and A. Chen, "Improving the Quality of Public Transportation by Dynamically Adjusting the Bus Departure Time," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, Tallinn, Estonia, 2023, pp. 834-843.
- [17] A. Drabicki, S. Ściga, and K. Chwastek, "Public transport effectiveness evaluation with capacity-constrained macroscopic assignment," in *2023 8th International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, Nice, France, 2023, pp. 1-6. doi: 10.1109/MT-ITS56129.2023.10241654.
- [18] S. Gheibi et al., "An Effective Data Structure for Contact Sequence Temporal Graphs," in *2021 IEEE Symposium on Computers and Communications (ISCC)*, Athens, Greece, 2021, pp. 1-8. doi: 10.1109/ISCC53001.2021.9631469.
- [19] P. Ni et al., "Parallel Algorithm for Single-Source Earliest-Arrival Problem in Temporal Graphs," in *Proc. of the ICPP*, Bristol, United Kingdom, 2017, pp. 493-502.
- [20] B. Strasser and D. Wagner, "Connection Scan Accelerated," in *Proc. of the 16th Workshop on ALENEX*, Portland, Oregon, USA, 2014, pp. 125-137.
- [21] J. Dibbelt et al., "Connection scan algorithm," *Journal of Experimental Algorithmics (JEA)*, vol. 23, pp. 1-56, 2018.
- [22] S. Wang et al., "Efficient route planning on public transportation networks: A labelling approach," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, Victoria, Australia, 2015, pp. 967-982.
- [23] H. Wu et al., "Reachability and time-based path queries in temporal graphs," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, Helsinki, Finland, 2016, pp. 145-156.
- [24] B.-M. Bui-Xuan, A. Ferreira, and A. Jarry, "Computing Shortest, Fastest, and Foremost Journeys in Dynamic Networks," *Int. J. Found. Comput. Sci.*, vol. 14, no. 2, pp. 267-285, 2003.
- [25] C. A. Haryan et al., "A GPU Algorithm for Earliest Arrival Time Problem in Public Transport Networks," in *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, Virtual, 2020, pp. 171-180. doi: 10.1109/HiPC50609.2020.00031.
- [26] transitfeeds.com, "OpenMobility Data," Dec. 2023. [Online]. Available: <https://transitfeeds.com>. [Accessed: December 1, 2023].
- [27] J. A. Enright, K. Meeks, and H. Molter, "Counting Temporal Paths," in *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023, March 7-9, 2023, Hamburg, Germany*, ser. LIPIcs, vol. 254. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 30:1-30:19. doi: 10.4230/LIPICS.STACS.2023.30.