Multimodal Transit Scheduler: An Actor-based Concurrent Approach

Prasad Talasila, Aparajita Haldar, Suhas S. Pai, Neena Goveas, and Bharat M. Deshpande
BITS, Pilani – KK Birla Goa Campus
tsrkp@goa.bits-pilani.ac.in

***Abstract*—Interactive journey planning applications are a necessity to take advantage of redundancy in multimodal public transit networks. As the scale of the transit network grows in terms of the number of stations, number of connections / services, and the number of transit operators, manual itinerary search may fail to detect best possible itineraries. Another challenging factor is the need to include the constraints imposed by intermediate stations on the creation of itineraries. One such constraint is the transfer time needed to alight one vehicle, and board another vehicle at a station. Other constraints such as non-availability of transport vehicles, and congestion have local origins, but affect the set of feasible itineraries. We introduce a generic model for a dynamic network. In our proposed dynamic network model, we incorporate both the local constraints (of a station), and the global constraints (user preferences).**

In this work, we use an Actor-based concurrent approach to design a transit scheduler. We model each transit station as an Actor, thereby gaining the advantage of accounting for dynamism in transit networks. Our system performs concurrent itinerary search over transit schedules. Our approach allows real-time updates to timetables, and status of transit stations. Our approach also solves the centralized schedule management problem by incorporating local update capability into the transit scheduler. Thus we enable local network operators to manage the vehicular traffic at their respective stations. We are also able to filter the itinerary search results based on user preferences. We have implemented a multimodal transit scheduler that has a median query completion time of $\sim$0.9 seconds on the public transit networks of India.

I. Introduction

Public transit networks form the backbone of transportation systems in any society. Widespread, and efficient usage of public transit networks facilitates cost-effective increase in user mobility. Public transit networks often offer different modes of transport, i.e., flight, train, bus, metro, taxi etc. When a public transit network consists of more than one mode of transport, it is called a *multimodal transit network*. One of the problems plaguing the multimodal public transit networks is the lack of interactive journey applications which work well in a federated transit system. Existing journey planning applications fall into two categories.

1) Exclusive, operator-centric solutions
2) Third-party, centralized solutions

The first category mostly consists of independent network operators like Emirates airline [1], and Indian Railways [2] who can provide transit through their own network. The second category consists of third-party centralized solutions such as Euro Railways [3], Google Transit [4], and Expedia [5]. In centralized solutions, the independent transit network operators would not be able to exercise real-time control over the use of their timetables in searching for schedules.

In this paper, we propose a decentralized, multi-operator approach to transit network journey planners. In a federated system of transit network operators such as Euro Railways, the independent transit network operators, and passengers would greatly benefit from distributed control over the transit schedules. The independent operators can update their transit schedules as per the local conditions; passengers can then query for feasible travel itineraries as per the current state of the federated transit network.

Most existing transit scheduling algorithms focus on quick generation of itinerary [6]. The usual approach taken is to represent a transit schedule as a time-varying graph on which itinerary search is performed as a variation of shortest path computation. Very few approaches consider user preferences during the selection of transit itinerary [7], [8]. Most of the existing approaches have significant graph pre-processing requirements as well [6]. The existing approaches also do not provide real-time control over the transit network to the operators.

We make two original contributions in this work. Our first contribution is a conceptual generalization of a dynamic network. We generalize the notion of time-varying network to introduce the concept of a generic edge, and a generic path.

Our second contribution is the application of Actor concurrency model to itinerary search problem. We developed a Transit Scheduler application that is a native concurrent solution to the itinerary search problem. We represent each station as an Actor of the Actor concurrency model. Actors can encapsulate complex behaviours, and have the dynamic, targeted messaging capability. Thus we are able to provide a rich computational model for a transit station; we also implemented a real-time state update capability for all the stations. Our approach allows independent transit operators to control the transit stations owned by them.

Our solution allows multiple network operators to control the transit network in real-time. We tested our implementation on air, rail, and bus transit networks of India. When run on a laptop computer with Intel core i5 processor, and 8GB RAM, our application is able to concurrently search the public transit timetables of India with a median query completion time of 0.9 seconds.

The rest of the paper is organized as follows. Section II provides summary of the prior literature. Section III defines

the itinerary search problem discussed in this paper. Section IV introduces the concepts of a generic edge, generic paths, and bridge functions. The same section also provides a summary on the Actor-view of a transport network; we use the Actor-view to introduce much needed concurrency into our solution. Section V gives details of our implementation approach, and the choices made during the implementation. Section VI describes the data sets from Indian public transport networks, and shows the performance of our Transport Scheduler on these data sets. Section VII contains conclusions.

II. Literature Review

Transit scheduling algorithms fall into either graph based or non-graph based approaches. Among the graph based approaches, time dependent (TD) graph based approaches proved suitable for complex transit query processing [6]. Among the non-graph based approaches, connection Scan Algorithm (CSA) [9], and Round-bAsed Public Transit Optimized Router (RAPTOR) [7] are capable of completing the transit query processing in milli seconds even on large transit networks [6]. Of the CSA, and RAPTOR approaches, CSA uses simpler data structures, and exploits cache locality properties of modern microprocessors [9], [10]. We use the *connection*, and *other means table* data structures from prior CSA-related work [9], [11].

We propose an Actor model-based approach to represent a transit network. Actor model is an established concurrency model [12]–[14]. Implementations of Actor models exist in different programming languages such as Elixir [15], [16], and Java [17]. Actor model-based simulation of transportation system has been attempted previously [18]. In the simulation of transportation system [18], the focus was on scalable simulation of transportation system with focus on participant behavior. In contrast, we focus on itinerary search on transit schedules. Agent models is another approach taken to represent transportation systems [19]. Multiagent-based systems have previously been used for simulations of travel patterns [20]. Instead of agents, we utilize actors to introduce autonomy for each station of a transit network.

III. Problem Statement

Definitions

A public transit system consists of a set of vehicles making trips between well-known transit stations as per a public timetable. The timetable of public transit system can be built from a fundamental concept known as a connection. A definition of connection is shown in Equation 1.

$$
\begin{aligned}
C &= (src, t_{dep}, dst, t_{arr}, vehicleID) \qquad (1)\\
where & \\
src &= \text{source station of the connection}\\
dst &= \text{destination station of the connection}\\
t_{dep} &= \text{vehicle departure time at the source station}\\
t_{arr} &= \text{vehicle arrival time at the destination station}\\
vehicleID &= \text{a unique vehicle identification number}
\end{aligned}
$$

A set of related, concatenating connections form a trip for a vehicle. A set of time-shifted trips form a route. Most public transit systems consist of unique vehicles covering well known routes. Thus we can represent the public transit timetable as consisting of a series of related connections. The theoretical underpinnings of a public transit timetable are detailed in [7], [9], [11], and [21].

A. Itinerary Search

A passenger using public transit systems utilizes a series of connections to move from one transit station to another. A strict ordering of the complete set of connections utilized by a passenger during one journey is referred to as an itinerary. Passengers would like to have time-efficient, hassle-free itineraries for their journeys. Often, passengers also wish to prefer certain kinds of connections; for example, a passenger can have a preference for train travel over all other modes of transport. Thus any itinerary generated for a passenger must match the preferences indicated by the passenger at the beginning of the itinerary search process.

The job of interactive journey planning algorithms / applications is to scan through the timetables of potentially multiple multimodal transit network operators to provide a time-efficient itinerary that matches user preferences. Interactivity would require itinerary search completion times in milli seconds.

B. Operator Control

Independent transit operators often cooperate to provide transit services over wider geographic areas. London transport system's Oyster Card [22] is one such example. The independent operators want to retain control over their transit networks, and timetables, yet cooperate with other transit providers to form federated transit services. Any transit scheduling application designed should be able to provide for operator control over their own transit networks, and timetables.

IV. Solution Approach

In this section, we develop the necessary theoretical foundations needed for developing the Transport Scheduler application. We evolve the definitions of time-varying networks to form a generic version of a dynamic network. In the case of time-varying networks, edge costs are a function of time. We define a dynamic network in which edge costs can be a function of multiple independent variables, i.e., we define a *generic edge*. We then extend the shortest path equation to work on generic edges.

In graph-centric network models, vertices act as passive meeting points for edges. We enhance a vertex (node) with capabilities to exhibit complex behaviors. Thus vertices play active role in determining the dynamics of a network.

A. Generic Edge

Most of the vehicle routing, and transit routing algorithms use graph theory notions of weighted, and directed graphs

TABLE I: The cost function of a decorated edge dependent on two independent variables.

a	b	cost
a_1	b_1	$f_1 = f(a_1, b_1, c_1)$
a_2	b_2	$f_2 = f(a_2, b_2, c_2)$
...		
a_i	b_i	$f_i = f(a_i, b_i, c_i)$
all other cases		∞

for network representation, and query processing [6]. In these approaches, a node is a passive junction that connects multiple edges. Edges themselves may be weighted, directed or both. Some transit routing algorithms such as CSA [9], transfer patterns [23] use time-varying edges to model transit networks. In CSA, transit edges are represented as connections which are nothing but an equivalent form of time-varying edge representation. In transfer pattern approach, the connections are stored on the edge as a compressed data structure. Both the approaches are based on the availability of a time-varying edge at certain (periodic) times as described by Equation 2.

$$cost(edge) = \begin{cases} C_i & at\ t = t_i, \\ \infty & \forall\ t\ \neq t_i. \end{cases} \tag{2}$$

In Equation 2, C_i represents the cost of using the edge at time t_i. In practice, C_i value depends on the connection available at that time. Thus we can represent the cost of using a connection as:

$$cost(C) = \begin{cases} t_{arr} - t_{dep} & when\ t = t_{dep}, \\ \infty & \forall \quad t \neq t_{dep}. \end{cases} \tag{3}$$

In Equation 2, we can understand the condition $t = t_i$ as a *constraint* on the edge. If the constraint is not satisfied, the edge become unusable. Similarly in Equation 3, $t = t_{dep}$ is a constraint on the edge; if that constraint is satisfied, the cost of using the edge is $t_{arr} - t_{dep}$.

We generalize the constraint on edge to form a *decorated edge* or a *generic edge*. Without loss of generality, we can say that all the edges of a network are dependent on two independent variables a, and b. Our definition of the cost function for a decorated edge is shown in Table I.

We can summarize the cost of a generic edge as:

$$cost(edge) = \begin{cases} f_i(a_i, b_i) & \forall\ a = a_i\ and\ b = b_i, \\ \infty & \text{for all other cases.} \end{cases} \tag{4}$$

B. Generic Path

In transit scheduling, we compute itineraries for a given user query. Since each itinerary is an ordered set of connections, an itinerary is equivalent to a path on a network. For mutimodal transit search queries, it is common to specify the conditions that an itinerary needs to satisfy. The conditions imposed on itineraries effectively become conditions on the network paths. Conditions or constraints imposed on the computed paths are called *path constraints*. Effectively, each user query comes with a set of path constraints. For example, the paths computed might only have to use air connections. Such conditions put on path computations can be understood as path constraints. We incorporate user preferences into itinerary search as path constraints.

TABLE II: Edge constraints, path constraints, and their compatibility

Edge Constraints	Path Constraints	Compatibility
flight	any mode	yes
flight	bus	no
train	bus or train	yes
	arrival before 5PM	only if computed $cost < 5PM$
train	bus-flight-bus	no

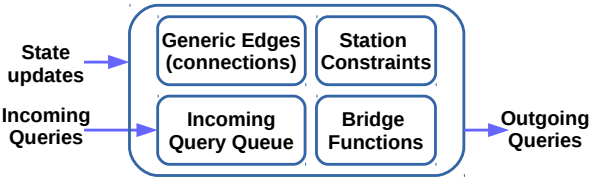


Fig. 1: An entity model for transit station. The station constraints, bridge functions, and generic edges together constitute the state, and behaviour of a station process.

Using the definition of decorated edge given in the previous section, we can redefine the equation for computation of path cost. A path can be extended using a decorated edge only if the constraints of the selected edge match with the existing constraints on the path.

Let there be two neighbouring nodes (v_a, v_b) connected by a decorated edge e_j. Let $path_i$ be the path computed from *source* to v_a; the cost of $path_i$ is denoted by $cost(path_i)$. Let $path_{i+1}$ be the the extension of $path_i$ with the decorated edge e_j. Let us denote the cost of $path_{i+1}$ as $cost(path_{i+1})$. Then the path cost computation can be represented as:

$$cost(path_{i+1}) = cost(path_i) + cost(e_j) \tag{5}$$

The computed path cost for $cost(path_{i+1})$ given in Equation 5 becomes finite only if the edge constraints of e_j are compatible with the path constraints of $path_i$. Otherwise, the computed path cost for $cost(path_{i+1})$ becomes infinite. A few sample edge, and path constraints are shown in Table II.

C. Bridge Functions

In transit scheduling, passengers often change vehicles at intermediate stations. Thus we need to consider time required to alight one vehicle, and board another vehicle at a station (called *transfer time*). It is impractical to schedule a connecting journey if the passenger does not get enough time to transfer vehicles. The transfer time is a function of transit station. We propose a generic approach to consider such station-specific constraints. We propose a *bridge function* that considers the suitability of a connection to extend a given path by utilizing station-specific local constraints (*station constraints*). Bridge functions give station manager ability to control traffic flow

via their own station. Bridge functions are similar in nature to cost functions of a decorated edge. Even bridge functions have to satisfy the constraint compatibility requirements during path cost computations.

D. Transit Station

In our approach, each transport station of the transit network exists independently as a stand alone entity with internal state, and behaviours. We use Actors model to represent transit stations. We model all the stations as independent Actors that help search the timetables to answer user queries on itineraries. We find that all the stations need to go through the same algorithmic steps in answering user queries. The only difference is in terms of the station constraints, and the generic connections that are available to each of the stations. Thus we can represent a station using the abstract schematic shown in Figure 1.

Here, each station entity has internal state that can be updated in real-time in a distributed manner by a transit operator. Transit operator controls the state of a station process in order to restrict the kind of itineraries that can be formed using that particular station as a participating node.

V. Implementation

A. Architecture

Our Transport Scheduler application is implemented using Elixir - a functional, concurrent programming language [15]. Elixir + Erlang Open Telecom Platform (OTP) has given us the advantage of implementing station entities using Actor concurrency model. The architecture of transport scheduler application is shown in Figure 2. The Elixir modules that form the base for our application are:

Input Parser (IP) that reads station-specific local variable values, and connection information from data set files.

Network Constructor, and Controller (NC) that spawns a separate Station process for each station, and initializes the spawned stations with data; NC also maintains a registry of process identifiers for all transit station processes.

Station process that maintains its state using a finite state machine (FSM) based on local variable values, and stores its outgoing connections data. Each station process uses an internal data structure to represent all the local information, and also selects from a set of predefined functions for manipulating any dependent local variables.

Update Controller (UC) that performs create, read, update, and delete (CRUD) operations on a station state. For example, an authorized transport manager may change the congestion level at a station which is reflected in transfer times, or mark it as having disturbance which renders the station unavailable for a period of time. UC helps in incorporating the effects of accident / incident related events into transit stations.

Query Collector (QC) that initiates query processing at source transit station, and collects query results from destination station. One dedicated query collector exists for each user query. QC also filters the itinerary result set based on user preferences.

User Query Controller (UQC) that receives user queries for itinerary search. UQC spawns a new QC for each incoming query, and hands over the job of query processing to the newly spawned QC.

In addition, supervisor processes that are an integral part of Elixir fault-tolerant applications are implemented to restart any dead processes. An Erlang construct called Erlang Term Storage (ETS), an in-memory table, is used to store final itineraries generated. Itineraries are built iteratively from source to detination station processes, and intermediate results are communicated between processes via Elixir message passing mechanism.

B. Application API

Three kinds of queries are supported by the application. They are: itinerary search, station schedule - both fetch, and update, and station update - create a new station, update a station, fetch details of a station. The corresponding query API URLs are shown in the Table III. All the necessary data exchange is done using JavaScript Object Notation (JSON) format.

C. Station as Process (Actor)

Elixir implements Actor model of concurrency, and refers to each instantiated Actor as a process. Each station process

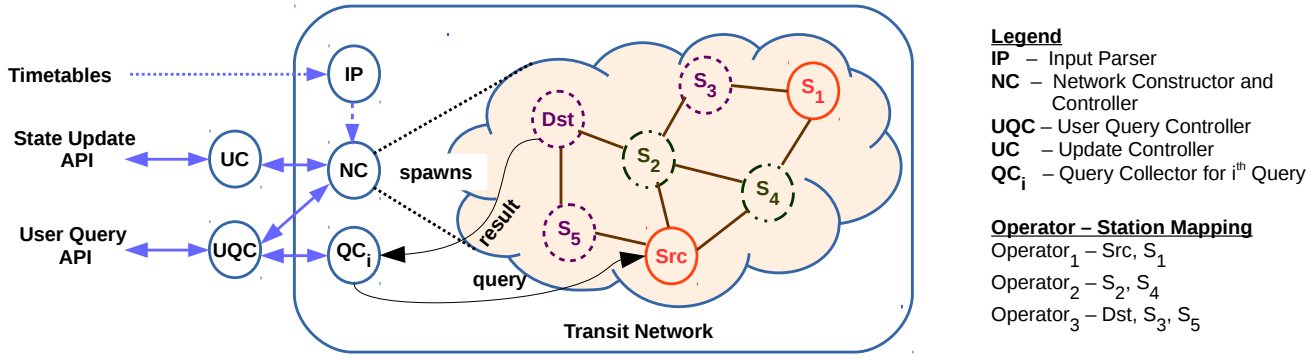


Fig. 2: Architecture of the Transport Scheduler application.

TABLE IV: Summary of timetables obtained from Indian public transport networks. (source: [11])

Name of Network	Mode of Transport	#Stations	#Connections
Indian Railways (IR)	train	2136	44876
Indian Flights (IFlights)	flight	80	2501
Long Distance Buses (LDB)	bus	147	12219
Complete Network (India)	multi-mode	2264	59555

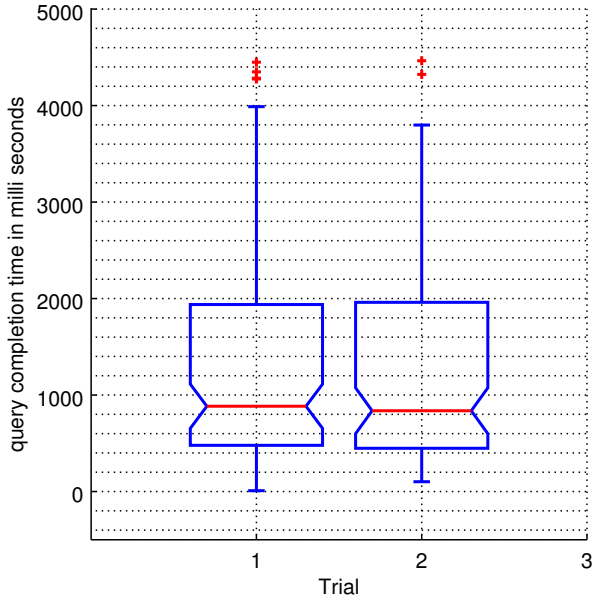
TABLE III: Query API URL strings

Module	**Query Type**	**Query String**
UQC	itinerary search	baseURL/api/search
UC	station schedule (fetch / update)	baseURL/api/station/schedule
UC	station create	baseURL/api/station/create
UC	station state (fetch / update)	baseURL/api/station/state

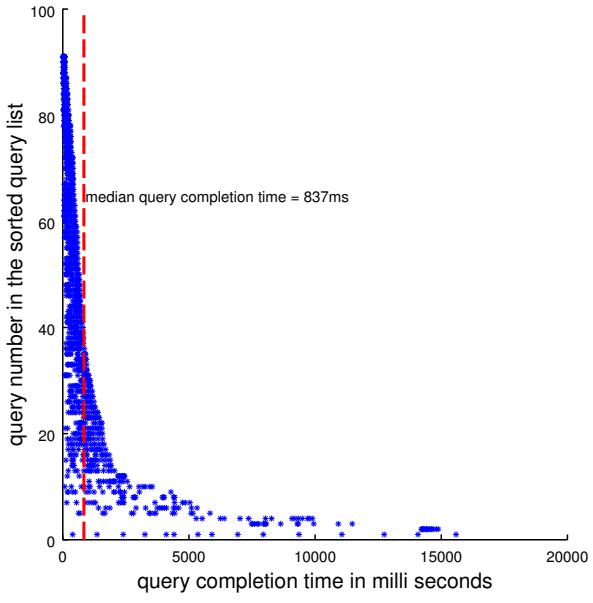
utilizes three distinct inputs. The first input is a set of outgoing connections represented in the format (vehicleID, source station, destination station, departure time, arrival time, mode of transport). The second input is a set of other means table (OMT) connections available due to footpath, and private transport connections between stations. These connections in OMT are described in [11]. The third input is a set of local variables, collectively called *station state*, that control the station behaviour. We use three station-specific local variables, namely, *congestion, delay*, and *disturbance*. The congestion is a three-valued variable holding low, medium, and high values; congestion variable indicates the level of congestion at the local station. Delay indicates the necessary transfer time to change vehicles at a station; As expected in a real-life scenario, we make the effective delay a function of the congestion level. Disturbance is a binary variable which indicates any local disruptions that stop the flow of vehicles via a transport station. If the local disturbance is set to yes, then all the incoming queries of a station are discarded. Effectively, the station becomes unavailable.

D. User Preference

User queries for itinerary search can contain transit mode specification. We utilize the user preference to filter the itinerary result set to discard undesirable results. The itineraries are filtered at QC. An itinerary result becomes undesirable if it contains any transit mode not preferred by the user. For example, a user choosing flight mode will have all the itineraries containing non-airline connections filtered out.



(a) Box plot on itinerary search times for two trials. In each trial, we execute 100 random queries.



(b) Query processing times for all members of the itinerary search results set.

Fig. 3: Itinerary search query performance of transport scheduler. Query collection times for the first fifteen queries are shown.

VI. Experimental Results

A. Data Set

We utilize the data set generated in a related project [11]. A summary table describing the data set (taken from [11]) is provided in table IV. Apart from 59,555 regular transit connections, the data set contains 151 other means (i.e., footpath, and private transport) connections. The data set has been derived from the transit network schedules of public transport network operators in India.

B. Implementation Platform

The transport scheduler has been implemented using Elixir language v1.4 compiled to run on Erlang Open Telecom Platform (Erlang/OTP) v19 [16]. All the tests have been run on a computer with Intel core i5 2GHz processor, and 8GB RAM running Ubuntu 16.04 x86_64 operating system. As discussed in Section V, our application incorporates user mode preferences, updates the state of a station using API, and outputs a list of itineraries for user queries.

C. Itinerary Generation Times

We conducted a set of randomized trials to understand the time response characteristics, and concurrent characteristics of Transport Scheduler. Because of the inherent concurrent nature of the architecture as described in Section V-A, the transport scheduler can concurrently propagate itinerary search query across the transit network. We issue one itinerary search query at a time, and measure the response time. Here, response time of a query is defined as the time taken to collect fifteen possible itinerary results from the network. We implemented a time out of 30 seconds to avoid deadlock for queries with less than fifteen feasible answers. A statistical analysis of the query response times for 100 queries results in a query response time distribution with median of 0.9 seconds, standard deviation of 4.1 seconds, skewness of 3.81, and kurtosis of 19.75. We performed multiple randomized itinerary search trials on the transport scheduler application. We show results of two randomized trials in Figure 3a using box plot.

For one of the two randomized trials, we also collected the timestamps of all the itineraries in the itinerary result set. We then sorted the queries in the ascending order of their completion time. Figure 3b illustrates the time at which different itineraries of the result set have been collected. A dotted line clearly marks the observed median of 823 milli seconds for this instance of randomized trial. Figure 3b clearly illustrates a right tailed distribution with heavy preference for low execution times.

VII. Conclusions

In this paper, we utilize the concept of constraint to provide a generic conceptual model for dynamic networks. We define edge, and path constraints for a dynamic network. We also redefine shortest path equation to satisfy edge, and path constraints. We use Actor concurrency model to adopt our proposed dynamic network model to represent public transit networks.

Our transit scheduler provides a native concurrent itinerary search service for multi-operator, and multi-modal transit networks. Our application generates multiple possible itineraries in response to each user query. We have tested our application with data from real-world transit networks. We are able to take advantage of the concurrent nature of the transit networks to provide a naturally concurrent solution.

Our solution allows network operators to provide real-time updates to station-specific local variables, such as delay, congestion, and local disturbance. Thus network operators are able to indirectly control the traffic flow through a transit station. The transport scheduler application we built incorporates user preferences in creating itineraries. Our experiments show that Transport Scheduler is able search public transit networks of India with a median query processing time of 0.9 seconds.

Source Code

The complete source code of Transport Scheduler application is available at *https://github.com/prasadtalasila/TransportScheduler*.

We welcome wide usage, and constructive feedback to help test the software.

Acknowledgment

The authors would like to thank Dr. A. Baskar, Dr. Ram Prasad Joshi for useful feedback on the theoretical framework suggested in this paper.

We thank Sushrut Sivaramakrishnan for help in data processing.

References

[1] "Emirates india," https://www.emirates.com/in/english/, accessed: 2017-04-20.

[2] "Welcome to indian railway passenger reservation enquiry," http://www.indianrailways.gov.in/, accessed: 2017-04-20.

[3] "Timetable: European train schedules," http://www.eurorailways.com/timetable/, accessed: 2017-04-20.

[4] Google, "Transit - google maps," https://maps.google.com/landing/transit/index.html, accessed: 2017-03-14.

[5] Expedia, "Expedia travel," https://www.expedia.co.in/, accessed: 2017-03-14.

[6] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, *Route Planning in Transportation Networks*. Cham: Springer International Publishing, 2016, pp. 19–80. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-49487-6_2

[7] D. Delling, T. Pajor, and R. F. Werneck, "Round-based public transit routing," *Transportation Science*, 2014.

[8] M. Müller-Hannemann, F. Schulz, D. Wagner, and C. Zaroliagis, "Timetable information: Models and algorithms," in *Algorithmic Methods for Railway Optimization*. Springer, 2007, pp. 67–90.

[9] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, "Intriguingly simple and fast transit routing," in *Experimental Algorithms*. Springer, 2013, pp. 43–54.

[10] B. Strasser and D. Wagner, "Connection scan accelerated," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics, 2014, pp. 125–137.

[11] T. Prasad, K. Sathyanarayanan, S. Tiwari, N. Goveas, and B. Deshpande, "t-csa: A fast and flexible csa implementation," in *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*, Jan 2016, pp. 1–6.

[12] H. Baker and C. Hewitt, "Laws for communicating parallel processes," 1977.

[13] G. A. Agha, "Actors: A model of concurrent computation in distributed systems." DTIC Document, Tech. Rep., 1985.

[14] G. Agha and C. Hewitt, "Concurrent programming using actors: Exploiting large-scale parallelism," in *Foundations of Software Technology and Theoretical Computer Science*. Springer, 1985, pp. 19–41.
[15] Plataformatec, "Elixir," http://elixir-lang.org/, accessed: 2017-03-14.
[16] "Erlang programming language," http://www.erlang.org/, accessed: 2017-03-14.
[17] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. ACM, 2009, pp. 11–20.
[18] S. Chen, F. Zhu, and F.-Y. Wang, "An erlang-based simulation approach of artificial transportation systems," in *Intelligent Transportation Systems (ITSC), 2016 IEEE 19th International Conference on*. IEEE, 2016, pp. 24–28.
[19] B. Raney and K. Nagel, "Truly agent-based strategy selection for transportation simulations," in *82nd Annual Meeting of the Transportation Research Board, Washington, DC*, 2003.
[20] K. Nagel and F. Marchal, "Computational methods for multi-agent simulations of travel behavior," in *Proceedings of the meeting of the International Association for Travel Behavior Research (IATBR)*, 2003.
[21] H. Bast and S. Storandt, "Flow-based guidebook routing." in *ALENEX*, 2014, pp. 155–165.
[22] T. for London, "Oyster online," https://oyster.tfl.gov.uk/oyster/entry.do, accessed: 2017-03-14.
[23] H. Bast, J. Sternisko, and S. Storandt, "Delay-robustness of transfer patterns in public transportation route planning," in *ATMOS-13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems-2013*, vol. 33. Schloss DagstuhlLeibniz-Zentrum fuer Informatik, 2013, pp. 42–54.