Smart Personalized Routing For Smart Cities

Abdeltawab M. Hendawi[1] Aqeel Rustum[2] Amr A. Ahmadain[1] David Hazel[3]
Ankur Teredesai[3] Dev Oliver[4] Mohamed Ali[3] John A. Stankovic[1]

[1] *University of Virginia, VA, USA,* {hendawi, aaa9aj, stankovic}@virginia.edu
[2] Saudi Aramco, DH, KSA, binrusas@aramco.com.sa
[3] University of Washington, Tacoma, WA, USA, {dhazel, ankurt, mhali}@uw.edu
[4] (ESRI) Environmental Systems Research Institute, CA, USA, doliver@esri.com

***Abstract*—In smart cities, commuters have the opportunities for smart routing that may enable selecting a route with less car accidents, or one that is more scenic, or perhaps a straight and flat route. Such smart personalization requires a data management framework that goes beyond a static road network graph. This paper introduces *PreGo*, a novel system developed to provide real time personalized routing. The recommended routes by *PreGo* are smart and personalized in the sense of being (1) adjustable to individual users preferences, (2) subjective to the trip start time, and (3) sensitive to changes of the road conditions. Extensive experimental evaluation using real and synthetic data demonstrates the efficiency of the *PreGo* system.**

I. INTRODUCTION

Modern smart cities aim at managing the city's infrastructures that include, but are not limited to, transportation systems, hospitals and health-care systems, urban planning, water and waste management, energy and power plants, and many other services. Routing from a source location to a destination is an essential service in people's daily life and an integral part in the smart cities blueprint [2], [5], [22].

Since smart cities integrate a massive number of sensors, (e.g., wearable devices), that embed location sensing, e.g., GPS-device, a tremendous amount of geo-tagged information, (e.g., GPS traces, accidents reports, air quality), are available [8]. Accordingly, it is feasible to expect users to look for a smarter and a personalized routing that goes beyond just finding the shortest path. Actually, personalization needs not be defined narrowly, but should be broad to include safety, special attractions, services, disruptions, social-proximity to other destinations, familiarity with neighborhoods, and numerous such factors.

In this work, we first formulate the personalized routing query problem as a preference augmented routing problem. Then we systematically design a solution that enables heterogeneous geo-tagged data sets to be stored, retrieved and inferred in conjunction with road network data. We then demonstrate the utility of the *PreGo* system [1] to process personalized routing queries. This problem is challenging because accurate extraction and representation of various attributes of the road network during different time periods of the day is a burdensome matter. Systematic consolidation of recent changes in maps while combining and processing users' personalized routing queries based on many attributes is also challenging. Evaluating each individual personalized query while assuring the overall system efficiency and scalability to serve large numbers of users is another arduous issue. In the domain of using geo-tagged data for route recommendation [9], [30], [39], previous work focuses on analyzing users' past trips in order to infer their preferred route to commute. As they consider GPS trajectories as a sole source, limited personalization is offered, e.g., using shortest distance or travel time. Hence, recommendations here provide very limited personalization.

[1] The name *PreGo* has two parts, *Pre* for preferred and *Go* that refers to routes. *Prego* is also an Italian word that means *You are welcome!* or *Please!*.

In this paper, we develop the *PreGo* system that is constructed from available geo-tagged data and that supports the evaluation of personalized routing queries with the following merits. (1) *PreGo* finds the best route according to a user's set of personal preferences. (2) *PreGo* does not depend on a static snapshot of the underlying road network. Rather, it computes the best route taking into consideration various cost elements at different time instances of the day. (3) For not in a rush user, *PreGo* recommends a trip *start time* such that user's preferences are best fulfilled. (4) Personalization parameters are controlled by each user's preference weights. (5) *PreGo* is internally crafted to scale to large road network graphs, a wide range of time sensitive preferences, and heavy routing request workloads.

To achieve this while preserving efficiency and scalability, *PreGo* leverages the Attributes Time Aggregated Graph (*ATAG*) structure. The *ATAG* structure is a representation of the underlying road network in which intersections are represented as nodes and roads as edges, and each edge can have multiple attributes or features, each of which could have multiple weights at different time slots during the day. For the construction of the *ATAG*, we use sets of different data sources to obtain the actual costs for each attribute in the graph during various time slots of the day, i.e., rush hours, early morning, or late night. The *ATAG* is instantly maintained in real time fashion once new data is received. This allows *PreGo* to provide routing recommendations based on a fresh snapshot of the underlying road map. For example, real time traffic, road closures, or car accidents data, trigger real-time updates to corresponding edges weight in the *ATAG* structures and, hence, in the results returned by *PreGo* are up to date.

To process such multi-preferences routing queries, the *PreGo* framework is equipped with the Time-Parameterized Multi-Preference Shortest Path (*TP_SP*) algorithm that efficiently extracts the best route from the *ATAG*. Via a single traverse of the up-to-date *ATAG* structure, the *TP_SP* algorithm discovers the best path(s) from a source to a destination with respect to a combination of attributes by applying the $prune$ and $wait$ approach.

The $prune$ concept tends to stop the expansion of graph traverse at node n when all attributes' total cost values, (i.e., $Cost_vector$, the total cost for each attribute on the path from the source node s to node n), on this node are dominated by the values, (i.e., $Cost_vector$), on the destination node d. Hence, the $prune$ strategy assures system efficiency by saving the computational resources from being wasted in extraneous graph expansion.

The $wait$ concept enforces the *TP_SP* algorithm to wait and not to declare the in-hand path at the destination node d as the optimal path until all other branches are pruned. That means we are sure that the accumulated cost values, (i.e., $Cost_vector$), at the destination node represent the costs of the dominant path and it will not be beaten by any other branch.

In order to further decrease the response time to a query, we introduce the *bidirectional* version of our *TP_SP* algorithm. For this,

we augment the *TP_SP* algorithm with an additional thread that starts its execution from the destination node d and traverses to reach the start node s. During its backward traversing, this thread computes the minimum accumulated cost, named upper limit μ, from each visited node n to the destination node d. This upper limit is seen by both forward thread and backward thread while they are expanding. With this bidirectional approach, the unnecessary branching can be significantly reduced by using the concept of pruning through setting up the upper limit for the newly branching cost.

For those users that have flexible time to begin their travel, we provide the *best-start TP_SP* algorithm. This algorithm recommends the best time to start a trip such that at this time user's preferences are best fulfilled.

The main contributions of our work are:

- The design and development of the *PreGo* system that leverages the available geo-tagged data sources to process personalized routing queries.
- The construction of the *Attributes Time Aggregated Graph (ATAG)* to model the underlying road network in a such a way to support multi-preference optimal path search based on the time of the day.
- The development of the Time-Parameterized Multi-Preference Shortest Path (*TP_SP*) algorithm to obtain the set of optimal paths in a single traverse of the road network graph.
- The development of the *bidirectional TP_SP* algorithm as a more optimized version from the response time perspective, and the *best-start TP_SP* to suggest a trip start time.
- A user study to validate drivers' routing preferences.
- The extensive empirical evaluation on real and synthetic data sets to study the performance of *PreGo*. Experiments provide that *PreGo* outperforms competitive techniques by more than one order of magnitude.

The rest of the paper is organized as follows. The study of related work is given in Section II. Section III sets the preliminaries. Section IV introduces the *PreGo* system. Section V describes the construction and maintenance of the *ATAG* from real VGI data. Section VI explains the algorithms augmented in *PreGo* to process routing queries. The *PreGo* system is experimentally evaluated in Section VII. Finally, Section VIII concludes the paper.

II. RELATED WORK

Dynamic multi-preference routing is an important area in Spatial Computing. Previous approaches have been static (i.e., time is not considered) and single preference-based [18], [25], [32], [36], [10]; dynamic (i.e., time-based), focusing on only one preference objective [6], [13], [26], [28], [29], [35]; or static, focusing on multiple preference objectives [3], [11], [12], [38], [40].

Previous static, single-preference-based approaches include classical routing techniques such as Dijkstra algorithm [10], A* [19], hierarchical [32], [36], materialization [18], [25], etc. These approaches are typically of interest to drivers who are mainly concerned with a single preference (e.g., minimizing travel time). However, they do not account for time-dependent scenarios (routing in rush hour vs. non-rush hour traffic), and do not consider multiple preferences (reducing fuel consumption, minimizing distance travelled, etc.).

Previous dynamic approaches [6], [13], [20], [26], [28], [29], [35] account for the time-varying nature of one attribute for each edge in the network. For example, if the objective is to minimize travel time between a given source and destination (i.e., the fastest path), time-based approaches are able to factor in the differences between rush-hour and non-rush-hour. Then they return a different result for the fastest path depending on the time of day. Representative time-based approaches include [29], and [13]. In [29], the dynamic pickup and delivery problem with transfers is explored; here the idea is to identify (in a dynamic scenario) the shortest path from a node representing the pickup location to that of the delivery location. In [13], a dynamic approach for bidirectional A* search for time-dependent fast paths is presented. However, existing dynamic approaches only consider a single preference objective and cannot take advantage of multiple preferences based on multiple attributes (e.g., travel time, fuel consumption, safety, etc). The work in [20] supports traffic aware routing and focuses solely on travel time attribute.

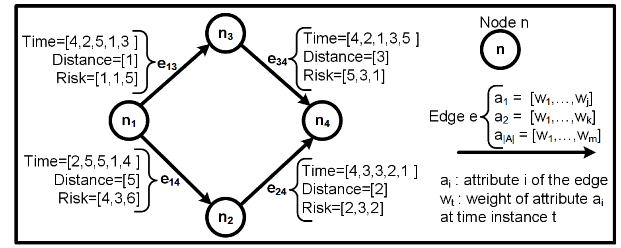


Fig. 1. Attributes Time Aggregated Graph (ATAG)

Previous static, multiple preference approaches account for multiple objectives in the network [3], [11], [12], [38], [40]. In [40], the skyline query problem is considered in a Wireless Sensor Network context with the objective of maximizing the network lifetime whereas [12] investigates how to find a set of paths (as opposed to a single path) so as to permit various choices concerning multiple criteria. However, existing approaches do not consider the dynamic nature of the network, which is important for gleaning information such as traffic light synchronization, curves in routes, safe periods to travel on routes, etc. Other work includes [33] that studies the dynamic routing with a focus on a single attribute. Authors of [15] provide a data mining technique for speed patterns extraction from large sets of traffic data. In [42], the authors focus on finding time-dependent shortest path under time constraints. They assume a single attributed edge. Existing work in the area of preference routing suffers from the following limitations. They focus on either obtaining the best route w.r.t. a single time-dependent attribute, or finding the best route w.r.t. multiple time-independent attributes on the road network graph.

The proposed *PreGo* system differentiates itself from the existing work by supporting multiple preference routing in variant time instances based on road networks that are fully constructed from the public available geo-tagged data sources.

III. PRELIMINARIES

In this section, we describe the concept of road network representation and modeling to serve the context of personalized multi-preference routing. Then, we set up a formal definition for the research problem we address in this paper. We also state the assumptions, and definitions as needed to frame the boundaries of this work.

A. Road Network Graph Modeling

Definition 1, Attributes Time Aggregated Graph (ATAG): The *ATAG* data structure is a directed graph $G(N, E, W)$ proposed to model the underlying road networks in the context of dynamic multi-preference routing [24]. Here, N represents the set of nodes and each $n \in N$ represents an intersection, or a dead end of a road. E is the set of edges where each $e(n, u) \in E$ represents a directed edge that links the two nodes n and u and indicates that e is traversable in the direction from node n to node u. W carries positive weights for the set of attributes/features A on each edge e. For example, $a \in A$ might represent the total travel time, distance, number of attractions, services, or car accidents, etc.

Each attribute a will have a vector of weights equivalent to the number of time slots of the day T, where $t \in T$ is a time interval $(t_s, t_e]$, e.g., {(6-10], (10-15], (15-19], (19-24], (24-6]}. Thus, the weight $w(e)_{at}$ is the weight of an edge e w.r.t. the attribute a at a start time in a time slot t of the day. Those weights also differ based on the day of the week. For example, the travel time on the University street is 10 minutes at 9:00am and 5 minutes at 11:00pm on Sundays, while it is 15 minutes during both times on Tuesdays. It is worth mentioning here that the *ATAG* is flexible in the sense that it allows different attributes to have different time intervals. For example, the *ATAG* structure in Figure 1 considers the whole day as one time interval for the distance attribute while it is divided into three and five intervals for the risk and travel time attributes, respectively.

We employ the *ATAG* structure to save all data in a way that allows computing the shortest path from a given source location to a destination. Intrinsically, *ATAG* supports multi-preference routing functions, e.g., less fuel consumption, less car accidents, less pollution, or paths with more services, at different start times.

Figure 1 illustrates the idea of storing multiple attributes for each single edge in the *ATAG* data structure. While this figure shows only three attributes, we emphasis that our *PreGo* framework and its *ATAG* structure and embedded algorithms can smoothly extend to accommodate other attributes as depicted experimentally in Section VII.

Definition 2, Cost of A Path (Route):

$$Cost(p)_t = < \sum_{i=1}^{k} w(e_i)_{a_1 t} \; , \; \sum_{i=1}^{k} w(e_i)_{a_2 t} \; , \; ... \; , \; \sum_{i=1}^{k} w(e_i)_{a_{|A|} t} >$$

Cost of a path p at time t consists of a vector of length $|A|$, i.e., number of attributes in the *ATAG*. Each value in this vector is obtained by the summation of all costs on the path edges w.r.t. a specific attribute a at the corresponding start time t.

Definition 3, An Optimal Path: An optimal path $\mathring{p}$ is the path with the minimum total cost w.r.t. at least one attribute among all possible paths between its two ends, (i.e., source and destination nodes), at a given start time. The set of optimal paths $\mathring{P}$ carries all possible optimal paths for all attributes at a certain start time t.

$$\mathring{P} = \bigcup_{t_s \in T, \forall a \in A} \{\mathring{p}\}$$

Definition 4, A Preferred Optimal Path: $Pref(\mathring{p})$ is the user preferred path among the set of optimal paths $\mathring{P}$. One user might prefer the path with minimum travel time, while another might prefer the one with lowest risk. This preferred optimal path is controlled by the user's given weighted-preferences. For example, a user who prefers the fastest path then the safest path and does not care about the distance can set the weighted-preferences as 80% travel time and 20% Risk. As will be shown later in Section VI-A, *PreGo* takes care of this by factoring the edges' weight using the user's given weighted-preferences.

B. Problem Definition

The main problem we address in this paper can be formalized as follows. Given a set of geo-tagged data, a source node n_s, a destination node n_d, and a user multi-preferences function $Pref$ on the set of the attributes A, (e.g., minimize travel time, fuel consumption, and car accident risk), we need to find the optimal path(s) $\mathring{p}$ from n_s to n_d that achieves the user multi-preference function at the start time t.

IV. SYSTEM OVERVIEW

In this section, we give a brief description of the proposed *PreGo* system. First, we outline the system architecture and highlight its main components. Next, we define the underlying geo-tagged data sources.

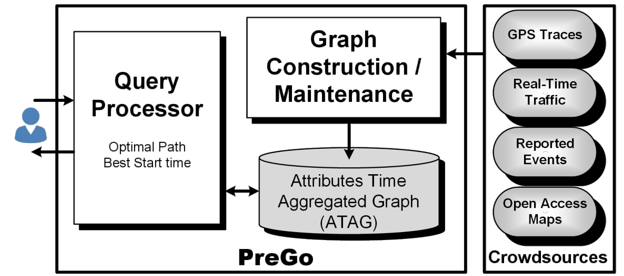


Fig. 2. The Architecture Of The *PreGo* System

A. System Architecture

The architecture of the *PreGo* system is given in Figure 2. *PreGo* consumes different types of geo-tagged data sources, namely, *public GPS traces*, *open access maps*, and *reported events*, e.g., car accidents. The system also has three main components, namely, the *attributes time aggregated graph (ATAG)* data structure, (Section III), the *graph construction and maintenance* module, (Section V), and the *query processing* module, (Section VI). This section discusses the data sources and overviews the other system components. Subsequent sections will detail the three main components of *PreGo*.

B. Geo-tagged Data Sources

The proposed framework relies on data extracted from heterogeneous data sets as follows.

(1) *Public GPS Traces.* Our main source is the GPS tracks for users' current and past trips. We consider this source as a core one for extracting the real travel time on each edge in the underlying road network. OpenStreeMap (OSM) allows the download of volunteered GPS traces filtered by areas of interest. We use a tool provided on the OSM wiki called JOSM [27] which allows us to view and download the GPS traces for certain areas on the map. In addition, we are able to get 15GB of public GPS traces [34].

(2) *Real Time Traffic.* This is a complementary source for the GPS traces to get an a fresh and more dense snapshot for the current traffic on the road network edges. This is easily obtainable through the traffic layer of the Google Maps APIs [16]. The standard plan of these APIs provides sufficient number of transactions for free.

(3) *Open Access Maps.* To obtain the base for the *ATAG* structure, we rely on the available free accessible map resources such as shape files [7], and the Minnesota traffic and map generator [4]. Through those resources, we are able to extract the road network graph as set of nodes, edges and compute the basic weights, e.g., distance. In addition, they give us the ability to extract some indicators about the near by services and points of interests around each edge, e.g., lakes, parks, commercial buildings, schools etc. (4) *Reported Events.* This source contains geotagged data for events reported by users on their ways, e.g., restaurant check-ins, care accident. For example, the number of recorded crimes and accidents events [31] around an edge gives an indicator of how risky this edge is. The TAREEG web-service [1] is used to obtain services and POI locations. Through its simple web interface, we are able to draw a rectangle that can cover a whole state and request the data for various types of services inside that rectangle. In addition to those public sources, *PreGo* allows its users to report past and instant geotagged point of interest, (e.g., park, restaurant), and events (e.g., car accident, fair).

V. GRAPH CONSTRUCTION AND MAINTENANCE

In this section, we explain how geotagged data sources are leveraged to construct and maintain the *ATAG* structure. The core of the

Algorithm 1 Extracting Travel Time Cost

Input: $ATAG\ \ G(N, E, W),\ \ GPS\ \ Trajectories\ \ TRAJ$

1: $wv(e)_t \leftarrow \phi$ *//weights vector for travel times on e at t*
2: $w(e)_t \leftarrow 0$ *//final travel time weight on e at t*
3: **for each** time slot $t \in T$ **do**
4: $TRAJ_t \leftarrow$ tracks in $TRAJ$ at time slot t
5: **for each** $traj \in TRAJ_t$ **do**
6: $E_{traj} \leftarrow MapMatch\ traj$ to its corresponding edge(s)
7: **for each** $e \in E_{traj}$ **do**
8: $wv(e)_t \leftarrow$ append time difference between first and last GPS points on e
9: **end for**
10: **end for**
11: **for each** $e \in E$ **do**
12: $w(e)_t \leftarrow$ average of weights in $wv(e)_t$
13: **end for**
14: Reflect $w(e)_t$ to $ATAG$
15: **end for**
16: **return** $ATAG$

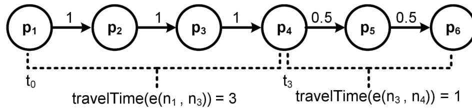


Fig. 3. Using GPS Tracks to Update Edge Weights

graph is obtained from the open access maps [1] by extracting the set of nodes, edges and weights for the distance attribute. The weights for other attributes, rather than the distance, are obtained according to the nature of each attribute. We can classify those weights into two categories based on the complexity to extract from the available data, (1) *simple weights*, e.g., risk, services, and (2) *complex weights*, e.g., travel time.

A. Simple Weights Extraction

The translation of the number of car accidents and crimes, e.g., NHTSA [31], into a risk weight for an edge is done through a straight forward process, i.e., normalizing this number by the edge distance as follows.

$$w(e)_{risk,t} = \frac{risks(e)_t}{distance(e)}$$

Normally, the smaller the value the safer the road. For those attributes with maximization dominance property, (e.g., find the most scenic path), we store the *complementary* weight. We do this to make the weights consistent with the minimization function in the optimal path calculation. For example, the optimal path based on the *services* attribute is the one that maximizes the number of services, e.g., restaurants and gas stations. Therefore, its weight is obtained as follows.

$$w(e)_{services,t} = \frac{NonService\ Addresses(e)_t}{total\ addresses(e)}$$

The number of available services on a road edge differs according to the time of the day. For example, the number of open Cinemas increases at night while the number of available athletic centers increases in the morning.

B. Complex Weights Extraction

In fact, the most complex weight to obtain is the travel time cost during specific time interval of the day. To achieve that, we leverage the public GPS traces. In the rest of this section, we focus on describing how to extract the travel time weights using the available GPS trajectories.

Main Idea. We mine the set of in-hand GPS tracks to pick up the valid ones, e.g., not violated speed limit nor direction of the edge. Then, for each edge in the graph, we obtain its weight by calculating the average travel time of those valid tracks in the corresponding time slot.

Algorithm. Algorithm 1 presents the steps to extract the travel time cost from a given set of trajectories. The input consists of the *ATAG* structure in addition to the in-hand GPS trajectories. The output is the *ATAG* with updated weights, (i.e., for travel time attribute). The algorithm starts by applying a kind of temporal clustering by examining each time slot and extracting only the relevant trajectories for that time slot, (Line 4 in Algorithm 1). For each trajectory relevant to a certain time slot t, the trajectory is map-matched [14] to its corresponding edges. Internally, we apply a kind of spatial clustering using the R-tree index [17] to retrieve all trajectories inside one portion of the *ATAG*. The travel time cost of each edge e that the trajectory $traj$ spans is then computed and appended to the vector $wv(e)_t$, (Lines 6 to 8). The $wv(e)_t$ contains a vector of weights for edge e at time slot t. Then, the algorithm calculates the average of the values in $wv(e)_t$ and uses this as the final travel time weight for the edge e at time slot t; this information is stored in $w(e)_t$, (Line 12). Next, the value in $w(e)_t$ is pushed to the *ATAG*, (Line 14). Once all time slots are considered, the returned *ATAG* should hold travel time weights for all edges during the stated time slots, (Line 16).

Example. Figure 3 illustrates how the collected GPS trajectories are analyzed and leveraged to update the *ATAG* data structure. In this example, we have a track with six points, p_1, p_1, ... p_6. The value on the arrow between each pair of points represents the time cost it takes to travel from point p_i, to its next point p_j, e.g., cost from p_1 to p_2 is one minute. When we map-match this track to the equivalent nodes and edges in the *ATAG*, we find that p_1 is matched to node n_1, and p_4 is matched to node n_3. Thus, the total cost for the edge connecting n_1 and n_3 is three minutes, assuming the trip starts at time slot t_0. In the same way, the edge between nodes n_3 and n_4 is one minute, assuming the start at time slot t_3. In some cases, the GPS points appear after the start node or before the end node of an edge. To handle this situation, we extrapolate the weight according to the ratio between the covered distance by the GPS points to the total distance of that edge as follows.

$$w(e)_{travelTime} = \frac{time(p_l) - time(p_f)}{distance(p_f,\ p_l)} \times distance(e)$$

Here, p_f is the first mapped GPS point to e after/at its start node, and p_l is the last point before/at the end node of e.

C. Harmony Between Real-Time and History

The initial construction of the *ATAG* is completely based on the historical data as described earlier. Therefore, if *PreGo* provides route recommendation depending on this data without augmenting the current real-time data, (e.g., most recent GPS tracks, traffic, accidents etc), the recommended routes will not match the reality. On the opposite side, if *PreGo* uses the most recent data to override the existing values in the *ATAG*, road network history will be lost. To overcome this issue, we maintain two versions for the *ATAG* structure; one for the historical data and the other for the real-time snapshot. The later is used to answer those routing requests that span the current time slot. Beyond that slot, we depend on the earlier version. For example, assuming the current time slot is one hour, in this case, all queries with less than one hour travel time are processed through the real-time snapshot *ATAG* solely. However, for long distance queries, (e.g., inter-states trips), the *ATAG* historical version is employed to answer the portion after the first hour. This means, both versions are harmoniously used side-by-side. For users' trips that will start after one hour, we just dispatch the historical *ATAG* as it accommodates a more mature picture for the whole road network.

Algorithm 2 Time Parameterized Multi-Preference Shortest Path (*TP_SP*) Algorithm

Input: $ATAG\ G(N, E, W)$, Source node s , Destination node d , Start time t , User Weighted-Preferences $Pref$

1: *//Cost_vector*$_{\hat{n}n}$*: the aggregated attributes' costs from source s to node n through a predecessor node* $\hat{n}$
2: *//Set the travel time to t, and all other attributes' costs to zero for the Cost_vector at the source node s*
3: $Cost_vector_s \leftarrow [(t)\ ,\ (0\ \ \forall a \in A)]$
4: Insert $Cost_vector_s \rightarrow PQ$ *//PQ is a priority queue*
5: $Path_u \leftarrow \phi$ *//optimal path from s to current node u*
6: Visited-Nodes List $VL \leftarrow \phi$
7: $Global_set \leftarrow \phi$
8: $Solution_set \leftarrow \phi$
9: **while** PQ is not empty **do**
10: $Cost_vector_{\grave{v}v} \leftarrow PQ.dequeue()$
11: **for each** node u accessor to v **and** $u \notin VL$ **do**
12: $e \leftarrow e(v, u)$
13: $Cost_vector_{vu} \leftarrow [(t_v + w(e)_{travelTime, t_v})\ ,\ (a_v + w(e)_{a,t_v}\ \ \forall a \in A)]$
14: **if** not dominated$(Cost_vector_{vu}\ ,\ Global_set\ ,\ Pref)$ **and** not dominated$(Cost_vector_{vu}, Solution_set, Pref)$ **then**
15: Append $u \rightarrow Path_u$
16: **if** $Cost_vector_{vu}$ *accured at the destination* d **then**
17: Insert $Cost_vector_{vu}\ ,\ Path_u \rightarrow Solution_set$
18: Remove dominated vectors and paths from $Solution_set$
19: Remove dominated vectors and paths from $Global_set$
20: **else**
21: Insert $Cost_vector_{vu}\ ,\ Path_u \rightarrow Global_set$
22: **end if**
23: **end if**
24: **end for**
25: $VL \leftarrow v$
26: $ND \leftarrow$ *Get not dominated* vectors and paths from $Global_set$
27: Insert $ND \rightarrow PQ$
28: Remove ND from $Global_set$
29: **end while**
30: **return** $Solution_set$

VI. Query Processing

This section explains the *query processing module* inside the *PreGo* framework. Initially, it illustrates how the *ATAG* data structure is employed to answer multi-preference routing queries. The section introduces the Time-Parameterized Multi-Preference Shortest Path (*TP_SP*) algorithm and describes how it is leveraged to provide personalized routing inside *PreGo*. Then, the *bidirectional* version of (*TP_SP*) and the *best-start TP_SP* algorithms are provided.

A. Find Optimal Paths

To find the set of optimal paths ($\mathring{P}$) for a given source-destination pair of nodes at a certain start time, we propose the Time Parameterized Multi-Preference Shortest Path *TP_SP* algorithm. **Main Idea.** In a single traverse, the *TP_SP* algorithm can find all optimal paths, i.e, the one with minimum total weight, at a given start time w.r.t. to all/subset of the attributes stored in the *ATAG* structure. This is achieved based on two concepts, (1) The *prune* concept cuts off the expansion of all in-hand branches, obtained after expanding the start node in *ATAG*, except for those branches that are likely to contribute to the optimal paths result. We call one of these branches a non-dominated sub-route. This internally means there is at least one attribute's weight in this route that dominates the weights of the same attribute(s) in all other ongoing branches as well as the weight of the obtained route, if any, at the destination node. (2) The *wait* concept tells the *TP_SP* algorithm to wait and not to consider the first route to reach the destination node as the optimal until all the ongoing branches are pruned and there is no further expansion. By doing this, we guarantee that the paths collected at the destination node represents the optimal set of paths.

Algorithm. The pseudocode in Algorithm 2 outlines the steps to get the set of time-parameterized multi-preference shortest paths from the underlying *ATAG* structure. The algorithm starts by initializing the used sets and priority queue data structures as follows, (Lines 3 to 8 in Algorithm 2). The $Cost_vector_{\grave{v}v}$ carries the costs for all attributes at the node v. This represents the total costs for the sub-route from the start node s to v passing through a predecessor node $\grave{v}$.

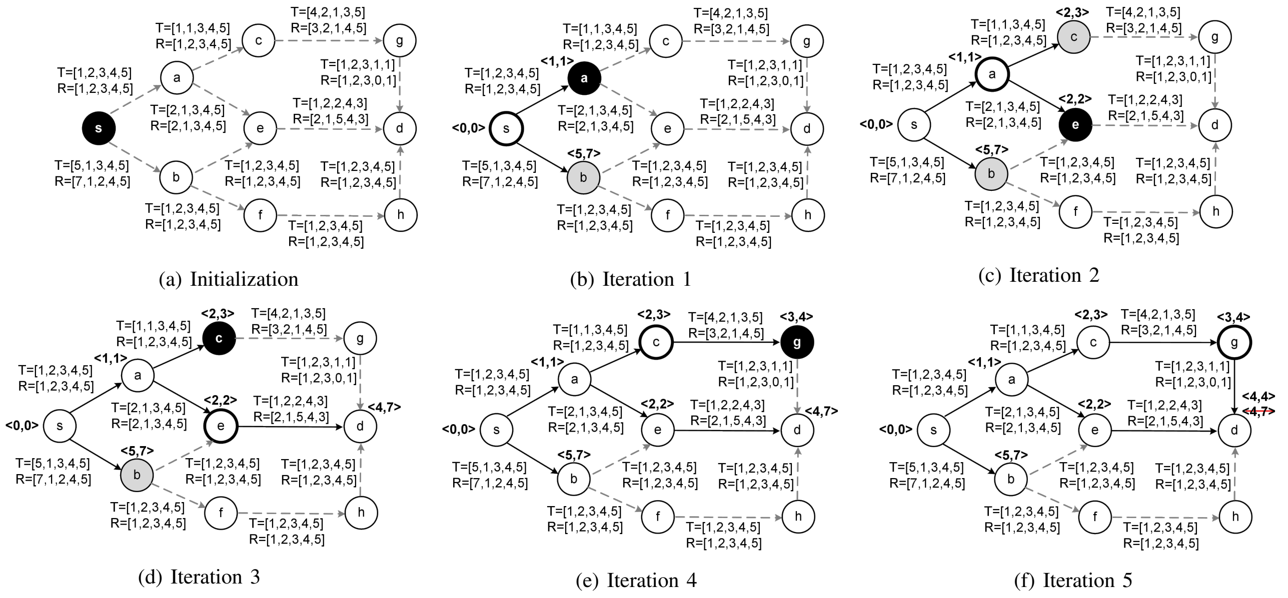
The first cost element in any cost vector represents the travel time cost. Thus, at the start node s, we initialize its cost vector, $Cost_vector_s$, by setting the travel time to the given start time t and other attributes' cost are set to zero, (Line 3). The $Global_set$ holds all nodes that are still open for possible expansion, (Line 7). $Path_u$ holds the optimal path to node u w.r.t. at least one attribute. When the graph traversing reaches the destination node d, $Path_u$ should be a final optimal path from s to d. The $Solution_set$ should maintain the cost vectors for those routes that successfully reach the destination node d, (Line 8). In addition, this set is going to be used to prune the $Global_set$ through removing all dominated cost vectors. Later at the end of the algorithm, $Solution_set$ will be carrying only the set of optimal paths $\mathring{P}$ for the given trip start time t.

Essentially, the priority queue PQ is meant for ordering the expansion. It is going to be populated with non-dominated cost vectors being extracted from the $Global_set$ along with the information about their sub-routes in $Path_u$. The non-dominated cost vector is the one that has the minimum cost for at least one attribute compared to other cost vectors in $Global_set$. Intuitively, the cost vector at s is the first guess to PQ, (Line 4).

Then, the algorithm iterates through the non-dominated cost vectors kept at PQ. We first dequeue a cost vector $Cost_vector_{\grave{v}v}$ from PQ and then start expanding from its last node which is v to its successors. A new cost vector $Cost_vector_{vu}$ is generated for each successor node u. That is done by summing up the values of the previous cost vector $Cost_vector_{\grave{v}v}$ and the weights of the recent passed edge $e(v, u)$ at the equivalent time slot of t_v. The t_v is the start time for the expansion process at node v which equals the total travel time costs from s to reach v through $\grave{v}$ plus the given start time t, (Lines 12 and 13).

The $Cost_vector_{vu}$ is examined to be non-dominated through the comparison with its competitors on the node u, i.e., $Cost_vector_{\grave{u}u}$, as well as those on the destination node d, i.e., those in $Solution_set$. This is achieved by calling the dominated() function that takes as arguments, the cost vector to be examined, the set of competitive cost vectors, and the set of user weighted-preferences $Pref$. This function leverages the user's weighted-preferences to decide which cost vector is the dominant one as follows. It normalizes the cost of each attribute according to the current maximum cost value of that attribute in the competitive set of vectors. Then it applies the user's weights by multiplying each attribute cost in all cost vectors by the corresponding user weighted-preference in $Pref$. Then the cost vector with the lowest total value for all attributes is the dominant one, (not dominated). In case the user has no weighted-preferences, absolute costs are compared in pairs. Therefore, two cost vectors can dominate each other, i.e., each vector is dominant w.r.t. one attribute.

The $Cost_vector_{vu}$ is pruned from further expansion, if it is completely dominated, otherwise, the algorithm proceeds and checks if this sub-route reaches the destination node, i.e., u is d. If this not the case, we just insert $Cost_vector_{vu}$ to the $Global_set$, (Line 21). Otherwise, we insert it into the $Solution_set$, accordingly, we remove from the $Solution_set$ and the $Global_set$ all cost vectors that are dominated by $Cost_vector_{vu}$, (Lines 17 to 19). Consequently, the number of cost vectors to be considered for further expansion is reduced, in turns, this leads to less CPU and memory overheads.



(a) Initialization

(b) Iteration 1

(c) Iteration 2

(d) Iteration 3

(e) Iteration 4

(f) Iteration 5

Fig. 4. Example For Tracing The TP_SP Algorithm

Before we proceed for next iteration of expansion, we update the PQ by moving to it all non-dominated cost vectors from the $Global_set$, (Line 26 to 28). Here, we need to highlight that the algorithm does not return the final answer until it is sure that the PQ is empty. This means routes in $Solution_set$ can not be dominated anymore. Finally, the $Solution_set$ that carries the set of optimal paths at the start time t for all attributes on *ATAG* is returned.

Example. To better understand how the algorithm works, Figure 4 gives an example to search for the set of shortest paths from a source node s to a destination node d at the start time t=0. For the sake of simplicity, we assume two attributes to be considered; travel time and risk. Both change over the time of the day and both are to be minimized. The graph's state and the dominance relationship are traced in each iteration. Furthermore, the distribution of the generated cost vectors is given in Table II, (note: C_V stands for $Cost_vector$).

We use the black background to indicate the nodes with non-dominated cost vectors and the gray background to indicate the nodes which have valid cost vectors in the $Global_set$. Also, we use the bold border to indicate the node that has just been expanded.

Initially, a cost vector is generated for the source node s with a start time and risk level both were set to zero and then be inserted into PQ to kick off the expansion as shown in Figure 4(a).

In the first iteration, two cost vectors were generated and added to the $Global_set$ as a result of expanding s, i.e., one for node a and one for node b. Because the cost vector for node a dominates the one for b, we delete a from the $Global_set$ and then insert it into the PQ to be considered for next expansion, Figure 4(b). It is noticeable that we have picked up the risk and travel time weights in the first time slot. However, at node a, the start time is two, so we use the weights in the second time slot.

In the second iteration, two more cost vectors are generated for each of the nodes c and e, and then added to the $Global_set$. The cost vector of node e is the only non-dominated one, therefore, it was nominated for the next expansion, Figure 4(c).

In the third iteration, we have the first route reaching the destination d with cost vector equals to $< 4, 7 >$. As a result, the cost vector for node b got removed from the $Global_set$ because of being dominated by the cost vector of the just discovered route, Figure 4(d). At this moment, the PQ is populated with the cost vector of c because it was the only one exists at the $Global_set$.

In the fourth iteration, a cost vector was generated for node g due to the expansion from node c. Then it is inserted into the PQ, Figure 4(e).

At the fifth iteration, a new cost vector appears at the destination d coming from g with dominant costs over the previous one. Therefore, the $Solution_set$ is updated by removing the previous cost vector and adding the in-hand one, Figure 4(f). Finally, the algorithm terminates because the PQ is empty. At the end the $Solution_set$ is returned containing $[s, a, c, g, d]$ as the shortest route with total travel time cost equals to four and total risk equals to four.

B. Finding Bidirectional Optimal Paths

In order to speed up the process of finding the optimal path and hence reducing the user's waiting time to get the answer for his routing query, we introducea *bidirectional* version of the TP_SP algorithm.

Main Idea. The main idea of the *Bidirectional TP_SP* algorithm is to start the optimal path finding process from the two ends of the routing query. This means, for each single routing query from a source location to a destination, we run *two* optimal path finding threads. One thread starts from the source node towards the destination node and is guided by the start time as in the above *TP_SP* and the other thread starts from the destination node back to the source node without considering the start time. Thus, the edges weights for all time instances are taken into consideration while the backward thread is expanding.

Intrinsically, when a backward branch meets a forward branch, we combine their cost values at the correct time instance, decided by the forward thread. Then, we set the combined cost value as an upper limit value μ which is used as a tool for pruning. As we get more meetings between forward and backward threads, we reset μ to the minimum value among the combined costs. We employ the μ to stop the expansion of all in-hand forward and backward branches that exceed the value of μ. For example, assuming that we have μ is set to 20 minutes for travel time cost and 5 as the risk cost. Then, all branches produced from the forward thread and cost more than these two values to go from the start node to the current node is pruned and prevented from further expansion. The same happens for all backward branches that their current costs to the destination

Algorithm 3 Bidirectional *TP_SP* Algorithm

Input: $ATAG\ G(N, E, W)$, Source node s , Destination node d , Start time t

1: //*Cost_vector*$_{\grave{n}n}$*: the aggregated attributes' costs from source s to node n through a predecessor node* $\grave{n}$
2: //*Set the travel time to t, and all other attributes' costs to zero for the Cost_vector at the source node s*
3: //μ *the time dependent upper cost vector of the first meeting between the forward and backward threads. Assuming they first meet at node v, then* $\mu \leftarrow Cost_vector_v + \beta_v$*, where* β_v *is the cost vector it takes to the destination node d from node v and this value is generated by the backward thread.*
4: $\mathcal{M} \leftarrow \phi$ //Hash-Map data structure for maintaining the settled nodes by the backward thread
5: $\mu \leftarrow \infty$
6: Call Backward_Thread, Call Forward_Thread

exceeds μ. With each extra forward-backward meeting, a pruning action is fired to clip the current expanding branches. When there is no more branches to prune or to expand, the in-hand μ represents the minimum cost and its reference route is declared as the optimal path. By doing this enhancement, at one side, we cut down up to half of the response time, (time since a query is issued to an answer is reported back), compared to the regular *TP_SP*. However, on the other side, we have to pay extra computation and memory overheads as a result of carrying the cost values for all time instances during the backward thread expansion.

Algorithm. The pseudo code in Algorithm 3 outlines the steps of the bidirectional algorithm. It also shows how to call both the forward and the backward search threads for finding the optimal path(s) from the underlying *ATAG* structure. The algorithm at Line 4 starts by initializing a common hash-map data structure named $\mathcal{M}$ that is used to (1) maintain the computed time dependent cost vectors for each settled node by the backward thread, and (2) from which the forward thread can get the heuristic time dependent cost vector of any node. At Line 5, the upper bound cost vector μ, which is the cost of a potential s to d path is initialized to infinity. Latter μ is going to be modified to a non-dominated cost vector resulting from any intersection between the forward and backward threads. Then the algorithm runs the backward and forward threads to traverse the graph from the end and start nodes simultaneously (Line 6 in Algorithm 3). The algorithm obtains the path with the lowest time-dependent accumulated costs in three phases. In each phase the behavior of both threads varies according to the constraints introduced at each one. The phases are as follows. In phase one, both of the forward and backward search's are up and running. The forward search works in the same way as the *TP_SP* algorithm, except that a heuristic cost vector, (i.e., the cost of each attribute on the path to the destination node d),

$$h_cost_vectors^{\leftarrow}_{n,t}, \forall t \in T$$

is computed at each node involved in the dominance relationship computation. Initially, the heuristic cost at each node, (except the destination node), that hasn't been settled yet by the backward thread is set to infinity as shown with this formula:

$$h_cost_vectors^{\leftarrow}_{n,t}\infty, \forall t \in T$$

Indeed, involving the heuristic function in this phase does not have an impact on the dominance relationship computation. That is because it is going to be the same for any settled nodes at this phase. Therefore, for this phase, we ignore it in the dominance relationship computation. On the other side, the backward thread traverses the graph using breadth first search and computes the minimum heuristic time dependent cost vectors for each visited node. The nodes settled by the backward thread are maintained in the common hash-map $\mathcal{M}$.

Algorithm 4 Backward_Thread

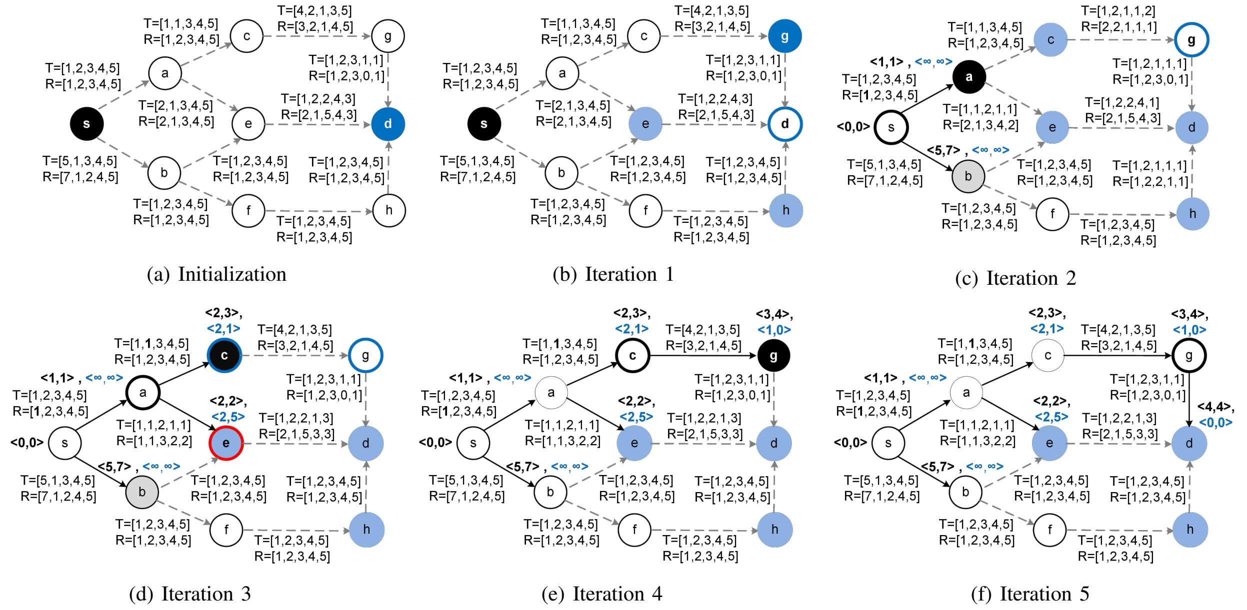
1: **for** each time slot $t \in T$ **do**
2: $\quad Cost_vector_d(t) \leftarrow 0\ \forall a \in A$
3: **end for**
4: Insert [d, $Cost_vector_d$] $\leftarrow \mathcal{M}\ PQ \leftarrow d$ //*PQ is a priority queue*
5: **while** PQ is not empty **do**
6: $\quad u \leftarrow PQ.dequeue()$
7: $\quad$**for each** node v predecessor to u **do**
8: $\quad\quad e \leftarrow e(v, u)$
9: $\quad\quad$**for each** time slot $t \in T$ **do**
10: $\quad\quad\quad Cost_vector_v(t) \leftarrow (w(e)_{a,t}\ \forall a \in A) + Cost_vectors_u(t + w(e)_{travelTime,t})$
11: $\quad\quad\quad$**if** $Cost_vectors_v(t)$ *not dominated by* μ **then**
12: $\quad\quad\quad\quad$**if** $\mathcal{M}.get(v)_t isnull$ **then**
13: $\quad\quad\quad\quad\quad$Insert $[v_t, Cost_vectors_v] \leftarrow \mathcal{M}$
14: $\quad\quad\quad\quad$**else**
15: $\quad\quad\quad\quad\quad \mathcal{M}.get(v)_t \leftarrow nonDominated(\mathcal{M}.get(v)_t, Cost_vectors_v)$
16: $\quad\quad\quad\quad$**end if**
17: $\quad\quad\quad$**end if**
18: $\quad\quad$**end for**
19: $\quad\quad$//*schedule the node, if it has at least one valid vector*
20: $\quad\quad$**if** $\mathcal{M}.get(v).size() > 0$ **then**
21: $\quad\quad\quad$Insert $v \leftarrow PQ$
22: $\quad\quad$**end if**
23: $\quad$**end for**
24: **end while**

This phase terminates once the two threads intersect at any node $v \in V$ for the first time. By then, the time dependent upper bound cost vector μ, which is initially set to ∞, is modified and set to the accrued cost vector on node v at time t_v plus the heuristic cost vector at the same node. Pruning of the expanding branches begins in this situation where all cost vectors dominated by μ at the $Global_set$ are going to be removed. In phase two, the forward search thread keeps working as before, but with additional constraint such that only cost vectors that are not dominated by μ are considered for expansion. The same constraint is applied on the backward thread where any new visited node is added to $\mathcal{M}$, if at least one of its time dependent cost vectors is not dominated by μ for at least one time slot. Therefore, it is allowed to proceed until no more nodes are eligible for expansion. At this moment, the algorithm transfers to the third phase. In phase three, only the forward search thread continues, with an additional constraint that only the nodes maintained at $\mathcal{M}$ are eligible to proceeds for expansion. The forward search terminates when it reaches the destination node d returning the optimal path resides at the $Solution_set$.

Example. For the sake of simplicity, and to better understand how the bidirectional algorithm works, we use the same example used for explaining the *TP_SP* algorithm. The graph's state and the dominance relationship are traced in each iteration. Furthermore, the generated cost vectors of the forward thread are provided in Table I, while the tracing of the backward thread is given if Figure 8.

As indicated earlier, we use the black background to indicate the nodes with non-dominated cost vector and the gray dotted background to indicate the nodes that has valid cost vectors in the $Global_set$. Also, we use the bold border to indicate the node that has just been expanded. On the other hand, we use the blue background to show the settled nodes by the backward thread and the blue bold border to indicate the node that has just been expanded.

Initially, a cost vector is generated for the source node s with a start time and risk level which are both set to zero and then inserted into the PQ to kick off the forward thread. On the other side, a cost vector per each time slot is generated for the destination node d and initialized to zero for each attribute and then added to the common Hash-map $\mathcal{M}$ and queued to trigger the backward thread to start



(a) Initialization (b) Iteration 1 (c) Iteration 2

(d) Iteration 3 (e) Iteration 4 (f) Iteration 5

Fig. 5. Example For Tracing The Bidirectional TP_SP Algorithm (best viewed in color)

traversing as shown in Figure 5(a). To focus more on explaining the backward thread, we assume that the backward thread is one step ahead from the forward thread in this example. In the first iteration of the backward thread as shown in Figure 5(b), nodes [g,e,h] are settled and five cost vectors are generated for each node as per the number of time slots defined in this example, Figure 8(a) and then added to the shared data structure $\mathcal{M}$ and queued for next expansion as well.

In the second iteration of the backward thread, node c was settled as a result of the expansion on node g as shown in Figure 5(c) and Figure 8(b). On the other hand, the forward thread had just started expanding on the source node s and as a result, two cost vectors were generated and added to the $Global_set$; one for node a and one for node b.

Since the time dependent costs at both nodes are still infinity, (i.e., because they have not been settled yet by the forward thread), we ignore the those costs in computation of the dominance relationship between nodes a and b. Accordingly, the cost vector for node a is dominant on the one for b, and therefore, it is deleted from the $Global_set$ and then inserted into the PQ for next expansion. Due

TABLE I. Cost Vectors For Forward Thread

Iteration	*Global_set*	*PQ*	*Solution_set*	μ
0	ϕ	$C_V_s = <0,0>$	ϕ	∞
1	$C_V_{sb} = <5,7>$	$C_V_{sa} = <1,1>$	ϕ	∞
2	~~$C_V_{sb} = <5,7>$~~	$C_V_{ac} = <2,3>$	ϕ	~~$<4,7>$~~
	~~$C_V_{ae} = <2,2>$~~		ϕ	$<4,4>$
3	ϕ	~~$C_V_{cg} = <3,4>$~~	ϕ	$<4,4>$
4	ϕ	ϕ	$C_V_{gd} = <4,4>$	$<4,4>$

to the expansion on node a in the second iteration of the forward thread, the forward and the backward thread had intersected for the first time at node e. As a result, the upper bound cost vector μ were set to $<4,7>$ and the phase were shifted to the next node. In the same iteration μ was updated to $<4,4>$ (which is less than its previous value), when the two threads intersected again at node c. As a result, the pruning happened for the $costvector$ of nodes e and b due to being dominated by μ and left only with the cost vector of node c at the $Global_set$. Therefore, it was nominated for the next expansion Figure 5(d). In the third iteration of the forward thread, the cost vector of node g, which was generated as the result of expansion on node c, was the only one at the $Global_set$. Therefore, it was nominated for the next expansion, 5(e). In the fourth iteration of the forward thread, the destination node d was reached for the first time. Therefore, the accrued cost vector was added to the $Solution_set$ and the forward thread is terminated. At the end the $Solution_set$ is returned containing [s,a,c,g,d] as the optimal route w.r.t the travel time and risk attributes with total travel time cost equal to four and total risk equal to four, Figure 5(f). Due to space considerations, We do not show the expansion of the backward thread on iteration two since the further expansions will not affect the final result.

Optimizing The Bidirectional algorithm. In the above $bidirectional\ TP_SP$ algorithm, the potential optimal paths cost can be discovered and/or updated with every intersection between the forward and the backward threads at a node $v \in V$. However, the navigation through the rest of the route, which is from v to d, is not known by that time. Hence, the forward thread keeps proceeding from the intersection node to find out the remaining directions to the destination node. In other words, the backward thread works as a guidance to the forward thread as it just calculates the total cost for the potential optimal path.

As an optimization, we propose to assign the navigation task to the backward thread. So that when the intersection with the forward thread occurs, the total cost vector and navigation are available. To achieve this, the backward thread needs to maintain a reference for each heuristic time dependent cost vector. By this optimization, the navigation time decreases. However, it requires extra memory to maintain the paths' references while traversing to meet the forward thread.

C. Finding The Best Start Time

In fact, the cost of an optimal route for a given source-destination pair of locations and w.r.t multiple dynamic attributes is highly influenced by the start time parameter. Based on that, *PreGo* is advanced with the *Best-start TP_SP* algorithm for finding those routes by determining the start time at which they can be achieved. Basically, our proposed approach for finding the best start time inherits its core idea from the *TP_SP* algorithm. The *Best-start TP_SP* algorithm can be summarized in four main steps as follows. (1) Initially, the expansion starts from the sources node s with initial cost vectors as many as the number of the predefined time slots of the day. Next, we

proceed the expansion with the non dominated cost vectors extracted from the in-hand cost vectors at the $Global_set$ (2) When arriving at the destination node d, only we maintain the non-dominated cost vectors accrued at d in the $Solution_set$ and then, we prune the $Global_set$ by eliminating the dominated cost vectors of the settled nodes. (3) We wait for the in-hand ongoing expansions for potential better routes. (4) Finally, when all expansions are either pruned or reached the destination, the algorithm terminates and returns the cost vectors in the $Solution_set$ along with their associated start time.

TABLE II. THE GENERATED COST VECTORS

Iteration	*Global_set*	*PQ*	*Solution_set*
0	ϕ	$C_V_s = < 0, 0 >$	ϕ
1	$C_V_{sb} = < 5, 7 >$	$C_V_{sa} = < 1, 1 >$	ϕ
2	$C_V_{sb} = < 5, 7 >$ $C_V_{ac} = < 2, 3 >$	$C_V_{ae} = < 2, 2 >$	ϕ
3	~~$C_V_{sb} = < 5, 7 >$~~	$C_V_{ac} = < 2, 3 >$	$C_V_{ed} = < 4, 7 >$
4	ϕ	$C_V_{cg} = < 3, 4 >$	$C_V_{ed} = < 4, 7 >$
5	ϕ	ϕ	~~$C_V_{ed} = < 4, 7 >$~~ $C_V_{gd} = < 4, 4 >$

The time asymptotic time complexity of the *TP_SP* algorithm is $O(|E|(|A|logT + log|N|))$ where T is the number of time instants, |N| is the number of nodes in N , $|E|$ is the number of edges in E in the *(ATAG)* graph and $|A|$ is the number of attributes in A.

VII. EXPERIMENTAL EVALUATION

The purpose of these experiments is to evaluate the efficiency and the scalability of the proposed algorithms within the *PreGo* framework in terms of two factors; CPU time and memory consumption.

A. Experimental Setup

The evaluation of the proposed approach was conducted through running various workloads of routing queries. Each query consists of source-destination pair of nodes and a given start time. We vary the values for four different parameters and observe the effect the overall performance. The parameters we focus on in this study are; (1) the number of preferred attributes, (2) the distance between the source node and the destination, (3) the number of queries, and (4) the number of time slots of the day. All experiments were conducted on a Windows 7 workstation equipped with Intel(R) Xeon(R) CPU E5-1607 v2 @ 3.00GHz processor and 32GB RAM.

All experiments are based on an actual Java implementation of the *TP_SP* algorithm and its variations, the *ATAG* data structure, and the whole *PreGo* system. The competitive techniques are implemented in Java as well and tested in the same environment. The source code of our implementation is accessible through [37], and the system interface and usage scenarios are described in [23].

The experiments were performed with generated routing queries on real road network map of Washington State, USA. This map has 535,451 nodes and 1,283,539 edges. The 15GB of GPS traces [34] is the core source for the attributes in the *ATAG* structure. The types of data sets described in Section IV-B, (e.g., reported car accidents), are not sufficient for performance evaluation. So, we generated synthetic data sets to serve the experimental evaluation purposes. We divide the day equally into five time slots. We also build a query generator to generate batches of queries, (from 10K to 50K), according to the need of each set of experiments.

B. Competitive Approach

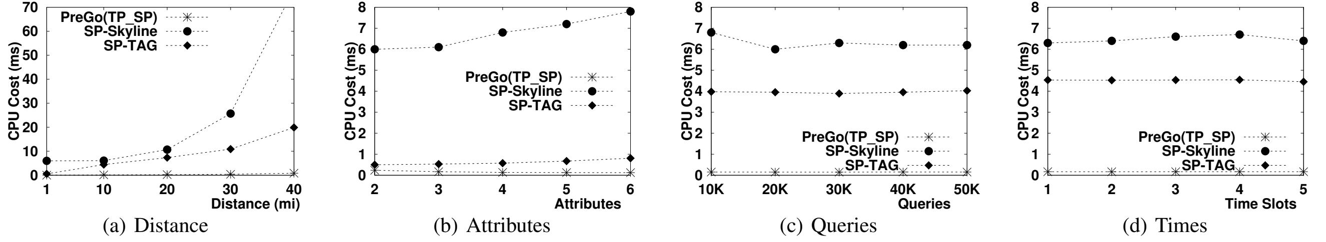
To provide a sound experimental evaluation of the proposed *TP_SP* algorithm and its bidirectional and best-start time versions, we need to compare its performance in terms of CPU time and memory consumption with a comparable algorithm. The most recent and competitive work is the *SP-Skyline* [41], where skyline approach is used to provide personalized routing. This work has a basic limitation, lack of pruning while traversing the road network graph. In other words, pruning happens at the end node. The number of supported attributes and time slots of the day and lack of best start time is another issue. Another relevant work is the *SP-TAG* algorithm [6]. It supports time-dependent graph traverse but for single attribute. It is also chosen as it supports best start time.

C. Effect of the distance between source-destination nodes

We vary the distance between source and destination nodes from 1 to 40 miles while the number of attributes and queries are kept constant at three attributes and 10K queries respectively. It is observed that the CPU time and memory consumption increase for the four algorithms, (*SP_TAG, SP-Skyline, PreGo(TP_SP), bidirectional TP_SP*), as the distance increases, Figure 6(a), Figure 7(a), Figure 9(a), Figure 10(a). That makes sense because the number of expansions/branching increases with the distance. Accordingly, the number of generated cost vectors increases which in turns increases the memory usage. However, it is clear that both algorithms of *PreGo* are significantly more efficient than the *SP-TAG* and SP-Skyline algorithms. That is because *PreGo* consolidates all attributes in the dominance function which is used inside its optimized graph navigation to obtain the optimal path in just one traverse of the graph. Another reason is that *PreGo* algorithms applies early pruning which significantly cuts the irrelevant graph traversing. As expected, the *bidirectional TP_SP* algorithm behaves better than the regular *TP_SP*. For example, at one mile query, it takes on average 6.0ms and 0.58ms/query from the *SP-Skyline* and *SP-TAG* algorithms, respectively, while it takes only 0.026ms and 0.01ms from our *SP-TAG* and *bidirectional TP_SP* algorithms respectively. At 40 miles, it takes 77.6ms, 19.9ms, 0.848ms and 0.475ms from the four algorithms respectively. We obviously can see that *PreGo* algorithms behaves more than one and two order of magnitudes better than the *SP_TAG, SP-Skyline* algorithms, respectively. This behavior is repeated throughout the remaining results.

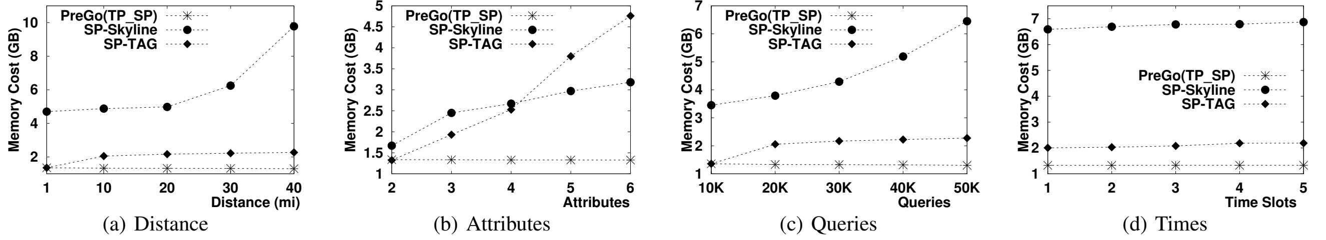
D. Effect of the number of preferred attributes

In this set of experiments, we increase the number of attributes to be considered in the optimal path(s) search from two to six attributes while maintaining the distance and the number of queries constant at 10 miles and 10K queries respectively. In general, we notice that the CPU time and memory consumption slightly decrease for the *TP_SP* as the number of preferred attributes increases, while it has a visible increasing trend in memory for the *bidirectional* Figure 9(b), Figure 10(b). It is a straight and sharp increasing pattern for the *SP-TAG and SP-Skyline* algoritms, Figure 6(b), Figure 7(b). The *bidirectional* outperforms other algorithms. This can be attributed to the fact that the number of non-dominated cost vectors queued for expansion at every iteration is decreasing due to involving more attributes that are working as filters in the dominance relationship computation, i.e., $prune$ concept. For example, consider four cost vectors with distance attribute only were involved in a dominance relationship. Assuming that those costs are 1, 1, 2, and 1 miles respectively. Therefore, three cost vectors are going to be queued for next expansions due to being not-dominated. With involving additional attribute such as travel time to these cost vectors, this will lead to have costs $< 1, 1 >, < 1, 2 >, < 2, 3 >, < 1, 2 >$. In this case, only one cost vector, i.e., $< 1, 1 >$, will be nominated and queued for the next expansion.



(a) Distance (b) Attributes (c) Queries (d) Times

Fig. 6. Performance Evaluation (CPU Time)



(a) Distance (b) Attributes (c) Queries (d) Times

Fig. 7. Performance Evaluation (Memory Overhead)

	g	e	h
1	<1,1>	<1,2>	<1,1>
2	<2,2>	<2,1>	<2,2>
3	<3,3>	<2,5>	<3,3>
4	<1,0>	<4,4>	<4,4>
5	<1,1>	<3,3>	<5,5>

(a) Iteration 1

	c	a	b	f
1	<5,4>	<3,2>	<3,2>	<3,3>
2	<3,2>	<3,6>	<3,5>	<6,6>
3	<2,1>	<5,6>	<4,5>	<4,4>
4	<5,6>	<4,5>	<6,9>	<7,7>
5	<6,6>	<2,4>	<8,8>	<10,10>

(b) Iteration 2

Fig. 8. Tracing of Backward Thread

E. Effect of the number of queries

We increase the number of queries from 10K to 50K while maintaining the distance and the number of preferred attributes constant at 10 miles and three attributes respectively. In general, we can observe that the CPU time for the four algorithms is fluctuating in a very narrow range as the number of queries increases. From the memory perspective, it is jus the *SP-Skyline* that increases while the number of queries increases. That is because, keeps all expansion results to the end of graph navigation. Therefore, it can be inferred that this parameter has no major impact on the total performance. However, there is a dramatic difference between the average CPU cost for the *SP-Skyline SP-TAG* at 6.8ms and 4ms, respectively, and the two algorithms of *PreGo* at 0.15ms and 0.075ms for the *TP_SP* and *bidirectional* respectively.

F. Effect of the number of time slots of the day

We range the number of time slots of the day from one, which means that there is no division of the day time, to five time slots while maintaining the distance and the number of preferred attributes constant at 10 miles and three attributes respectively. In total, it is observed that the CPU time and memory consumption are fluctuating within a small range a round four different cost values at 6.6ms/6.78GB, 4.5ms/2GB, 0.168ms/1.33GB, and 0.091ms/1.33GM for the the *SP-Skyline*, *SP-TAG*, *TP_SP*, and *bidirectional TP_SP* respectively, Figure 6(d), Figure 7(d), Figure 9(d), Figure 10(d). Hence, it can be deduced that this parameter does not have significant influence on the overall performance patterns.

G. Evaluation of Obtaining Best-Start Time

In this set of experiments, we examine the performance of the *PreGo Best-start* algorithm that is introduced to obtain the best time recommended to start a trip such that the optimal path can be obtained.

With varying the distance between source and destination nodes, it is observed that the CPU time and memory consumption increase for the two algorithms as the distance increases. It is obvious that *PreGo* is significantly more efficient than the *SP-TAG Best-Start* from the CPU time perspective, Figure 11(a). For example, with 40 miles distant query, it takes 6.75ms from *SP-TAG*, while *PreGo* accomplishes it in just 1.116ms. That is because *PreGo* consolidates all attributes in one run. However, *PreGo* is approaching the *SP-TAG Best-Start* in the memory consumption as shown in Figure 12(a). That's because the increasing in the number of expansions with long distances. Accordingly, the number of generated cost vectors will increase which in turns increases the memory usage.

In the study of the effect of the number of attributes to be considered while obtaining the best start time, we get the following patterns. (1) For the CPU time, it is noticed that both algorithms have increasing trend. However, *PreGo* is growing slightly with smaller values than the *SP-TAG* which rises sharply with much larger values. (2) Regarding the memory consumption, both of them slightly increase but the cost was higher with *PreGo*. That is attributed to the consolidated set of attributes in one run, in contrast with a single attribute at a time in *SP-TAG Best-Start*.

When we vary the number of queries, we find that the CPU time slightly increases for both algorithms, however, *PreGo* acts with smaller values. For example, at 5K queries, *PreGo* accomplishes it in 0.14ms/query while *SP-TAG* takes 0.78ms on average, Figure 11(c). From memory consumption view, *SP-TAG Best-Start* almost goes steady at 1.3GB, whereas *PreGo* starts at 0.93GB, then it increases to reach the *SP-TAG Best-Start*, Figure 12(c).

When we test the effect of the number of time slots of the day, we notice that the CPU time and memory consumption are almost going steady with a very slight increasing trend with *PreGo*. On the other side, it is significantly increasing with SP-TAG Best-Start, Figures 11(d) and 12(d). In a nutshell, *PreGo* outperforms *SP-TAG* in finding the best start time.

H. User Study

We conduct a user study to validate if users really need a routing service that goes beyond the shortest travel time route. We also aim at examining the variability of weights for each routing option,

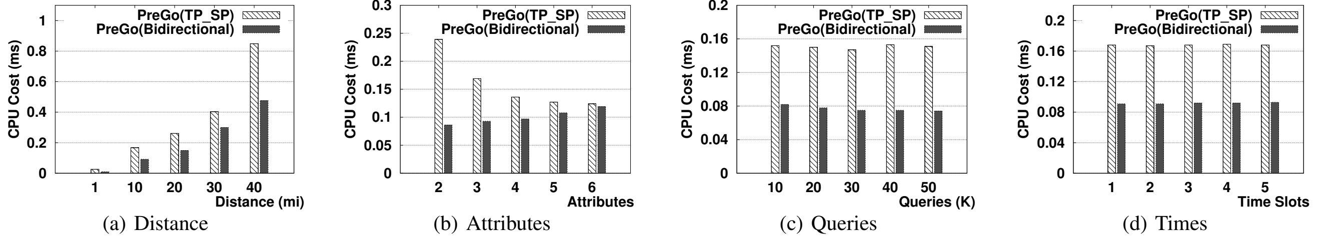


Fig. 9. Effect of *bidirectional* Processing on Performance (CPU Time)

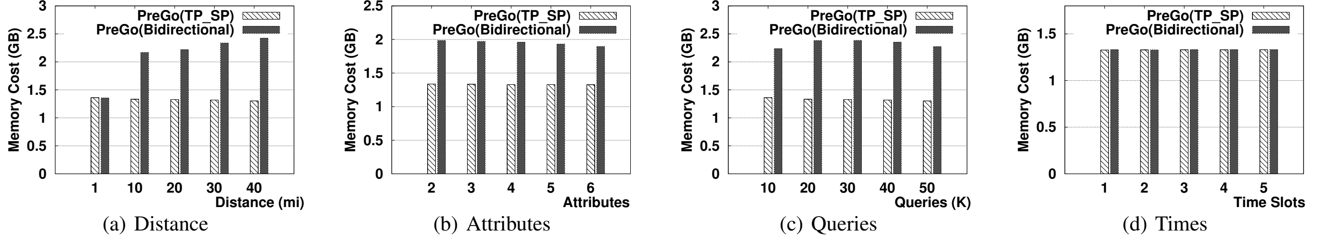


Fig. 10. Effect of *bidirectional* Processing on Performance (Memory Overhead)

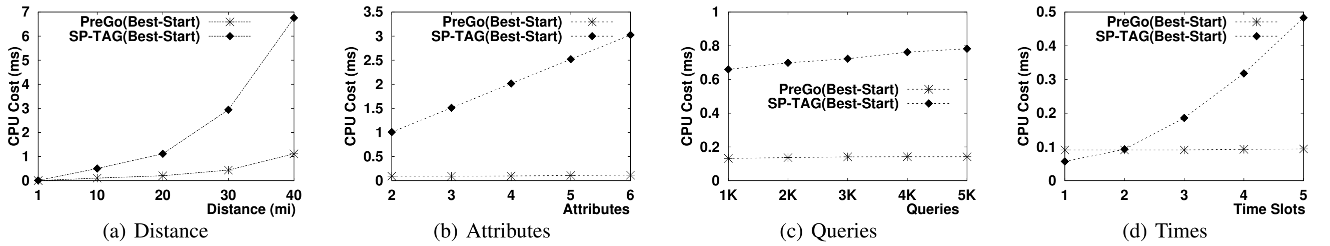


Fig. 11. Find Best-Start Time, Performance Evaluation (CPU Time)

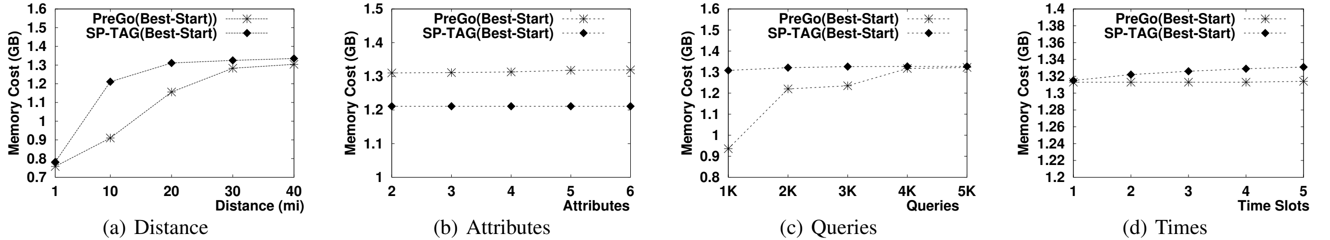


Fig. 12. Find Best-Start Time, Performance Evaluation (Memory Overhead)

(e.g., safest or scenic routes). We surveyed 58 adult participants around the University of Virginia and the University of Washington campuses. The participants consist of 28 females and 30 males and they represent non-students, city residents, students, staff, and faculty. The study asks the participants to rank and weight five different routes, (fastest in travel time, shortest in distance, safest which is lowest in car accidents, most scenic, and route with most services and stores) during four different driving conditions (in the morning to work/school, at night traveling back home, on a vacation for which you drive, and on weekends for shopping). The questions and all participants' responses are available to the reader for further analysis through [21]. Figure 13 gives the summary of the user study. The weighted average for each route during the four listed conditions are classified based on the gender. As expected, users take fastest route in their daily commute to work and school. Female drivers tend to care more about the safest path in their daily trips. At night, drivers prefer to go through a path that is safer. Though, male drivers slightly prefer the fastest route but, female drivers choose the safest route. From the third and fourth questions, all users choose to enjoy the scenic and most serviced paths if they travel for a vacation or shopping during a weekend. However, female users still consider safety more than the male users. From this study, we can conclude: (1) Drivers do have variety of routing preferences beyond shortest route, (2) Safest, most-scenic, and most-serviced paths are important options, (3) Users' preferences are dynamic and depend on the time of the day and trip conditions and (4) User's weights span a wide range of values that reflect their need for smart personalized routing.

VIII. Conclusion

This paper presents the *PreGo* system that leverages the geotagged data obtained from a variety of sources in smart cities to provide personalized multi-preference routing services. This work includes handling efficiency and scalability issues when the areas of maps being handled are large or the number of personalization parameters increases linearly. *PreGo* also supports optimizing start times, and integrating user's weighted preferences Those are key features missing from several research and commercial frameworks. Experimental evaluation demonstrates that the (*TP_SP*) family of algorithms within *PreGo* performs more than 100 times faster than the competitive techniques. A user study with 58 participants confirms the users' desire for personalized routing that offers more than the shortest path in time or distance.

Gender	Morning to work/school					Night back home					Vacation for a journey					Weekend for shopping				
	Fast	Short	Safe	Scenic	Services	Fast	Short	Safe	Scenic	Services	Fast	Short	Safe	Scenic	Services	Fast	Short	Safe	Scenic	Services
F	48.3	22.0	18.0	5.9	5.9	25.5	15.0	48.8	3.8	7.0	20.8	10.1	21.3	34.4	13.4	24.1	12.0	15.3	7.5	41.1
M	61.5	18.0	10.3	6.1	4.1	38.7	13.3	37.5	3.9	6.6	16.0	10.3	19.2	40.8	13.7	28.5	8.3	9.5	11.6	42.0

Fig. 13. Summary of The User Study. Average Weight (Percentage) For Each Route/Condition Classified by Gender.

REFERENCES

[1] L. Alarabi, A. Eldawy, R. Alghamdi, and M. F. Mokbel. TAREEQ: A MapReduce-Based Web Service for Extracting Spatial Data from OpenStreetMap. In *SIGMOD*, Utah, USA, June 2014.

[2] AllWebsiteStats. Statisitcs For Websites Usage. http://allwebsitestats.com/, June 2014.

[3] Balteanu Adrian and Jossé Gregor and Schubert Matthias. Mining driving preferences in multi-cost networks. In *SSTD*, pages 74–91, Munich, Germany, Aug. 2013.

[4] J. Bao, A. Magdy, M. Sarwat, and M. F. Mokbel. Minnesota Traffic Generator. URL:http://mntg.cs.umn.edu/traffic_requests/create_request/, Oct. 2013.

[5] M. Batty, K. W. Axhausen, F. Giannotti, A. Pozdnoukhov, A. Bazzani, M. Wachowicz, G. Ouzounis, , and Y. Portugali. Smart cities of the future. *The European Physical Journal Special Topics*, 214(1):481–518, 2012.

[6] G. Betsy, S. Kim, and S. Shekhar. Spatio-temporal network databases and routing algorithms: A summary of results. *Advances in Spatial and Temporal Databases*, 4605:460–477, 2007.

[7] U. C. Bureau. TIGER/Line and Shapefiles. http://www.census.gov/geo/maps-data/data/tiger-line.html, Oct. 2013.

[8] J.-P. Calbimonte, J. Eberle, and K. Aberer. Semantic Data Layers in Air Quality Monitoring for Smarter Cities. In *International Semantic Web Conference. Semantics for Smarter Cities*, Pennsylvania, USA, Oct. 2015.

[9] D. Delling, A. V. Goldberg, M. Goldszmidt, J. Krumm, K. Talwar, , and R. F. Werneck. Navigation made personal: Inferring driving preferences from gps traces. In *ACM SIGSPATIAL GIS*, Washington, USA, Nov. 2012.

[10] D. Edsger. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[11] G. Franz, K. Hans-Peter, R. Matthias, and S. Matthias. Mario: multi-attribute routing in open street map. In *SSTD*, pages 486–490, Minnesota, USA, Aug. 2011.

[12] Fujimura and Kikuo. Path planning with multiple objectives. *Automation Magazine*, 38(1):33–38, 19996.

[13] N. Giacomo, D. Daniel, L. Leo, and S. Dominik. Bidirectional A* search for time-dependent fast paths. *Experimental Algorithms*, pages 334–346, 2008.

[14] C. Y. Goh, J. Dauwels, N. Mitrovic, M. T. Asif, A. Oran, and P. Jaillet. Online map-matching based on hidden markov model for real-time traffic sensing applications. In *SIGMOD*, pages 776–781, Anchorage, USA, Sept. 2012.

[15] H. Gonzalez, J. Han, X. Li, M. Myslinska, and J. P. Sondag. Adaptive fastest path computation on a road network: A traffic mining approach. In *VLDB*, pages 794–805, Vienna, Austria, Sept. 2007.

[16] Google. Google Maps APIs. Traffic Layer. https://developers.google.com/maps/documentation/javascript/trafficlayer, July 2016.

[17] Guttman and Antonin. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 47–57, Massachusetts, USA, June 1984.

[18] S. Hanan, S. Jagan, and A. Houman. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, Vancouver, Canada, June 2008.

[19] P. E. Hart, N. J. Nils, and R. Bertram. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[20] W. Henan, G. Li, H. Hu, S. Chen, B. Shen, H. Wu, W.-S. Li, and K.-L. Tan. R3: a real-time route recommendation system. In *VLDB*, pages 1549–1552, Hangzhou, China, Sept. 2014.

[21] A. M. Hendawi, D. Hazel, A. Rustum, A. Teredesai, D. Oliver, M. Ali, A. A. Ahmadain, and J. A. Stankovic. Smart Routing User Study Results. URL:http://www.cs.virginia.edu/hendawi/UserStudyResults.pdf, Oct. 2016.

[22] A. M. Hendawi, M. Khalefa, H. Liu, M. Ali, and J. A. Stankovic. A vision for micro and macro location aware services. In *ACM SIGSPATIAL GIS*, page 12, California, USA, Oct. 2016.

[23] A. M. Hendawi, A. Rustum, A. A. Ahmadain, D. Oliver, D. Hazel, A. Teredesai, and M. Ali. Dynamic and Personalized Routing in PreGo. In *MDM*, Porto, Portugal, June 2016.

[24] A. M. Hendawi, E. Sturm, D. Oliver, and S. Shekhar. CrowdPath: a framework for next generation routing services using volunteered geographic information. In *SSTD*, pages 456–461, Munich, Germany, Aug. 2013.

[25] S. Jagan and S. Hanan. Query processing using distance oracles for spatial networks. *TKDE*, 22(8):1158–1175, 2010.

[26] X. Jiajie, G. Limin, D. Zhiming, S. Xiling, and L. Chengfei. Traffic aware route planning in dynamic road networks. In *DASFAA*, pages 576–591, Busan, South Korea, Apr. 2012.

[27] JOSM. An extensible editor for OpenStreetMap (OSM). http://josm.openstreetmap.de/wiki, Jan. 2016.

[28] L. Julia, J. Krumm, and E. Horvitz. Trip router with individualized preferences (trip): Incorporating personalization into route planning. *Proceedings of the National Conference on Artificial Intelligence*, 21(2):1795–1800, 2006.

[29] I. KB and S. V. Goal Directed Relative Skyline Queries in Time Dependent Road Networks. *arXiv preprint arXiv:1205.1853*, 2012.

[30] W. Ling-Yin, Y. Zheng, and W.-C. Peng. Constructing popular routes from uncertain trajectories. In *KDD*, pages 195–203, Beijin, China, Aug. 2012.

[31] NHTS. National Highway Traffic Safety Administration. Fatality Analysis Reporting System (FARS). ftp://ftp.nhtsa.dot.gov/FARS/, Jan. 2016.

[32] J. Ning, H. Yun-Wu, and R. E. A. Hierarchical optimization of optimal path finding for transportation applications. In *CIKM*, pages 261–268, Maryland, USA, Nov. 1996.

[33] M. Nirmesh, S. Madden, and A. Bhattacharya. A continuous query system for dynamic route planning. In *ICDE*, pages 792–803, Hannover, Germany, Apr. 2011.

[34] OSM. GPS Tracks. http://planet.openstreetmap.org/gps/, Jan. 2016.

[35] B. Panagiotis, S. Dimitris, D. Theodore, and S. Timos. Dynamic pickup and delivery with transfers. In *SSTD*, pages 112–129, Minnesota, USA, Aug. 2011.

[36] J. G. Rosario, S. Thambipillai, and Q. KH. Heuristic techniques for accelerating hierarchical routing on road networks. *IEEE Trans on Intelligent Transportation Systems*, 3(4):301–309, 2002.

[37] A. B. Rustum, A. M. Hendawi, and M. Ali. PreGo Source Code and Sample Data. URL:https://github.com/binrusas/PreGo, Oct. 2015.

[38] N. Saeed and M. R. Delavar. Multi-criteria, personalized route planning using quantifier-guided ordered weighted averaging operators. *International Journal of Applied Earth Observation and Geoinformation*, 13(1):322–335, 2011.

[39] C. Vaida and C. S. Jensen. Vehicle Routing with User-Generated Trajectory Data. In *MDM*, pages 14–23, Pennsylvania, USA, June 2015.

[40] L. Weifa, C. Baichen, and Y. J. Xu. Energy-efficient skyline query processing and maintenance in sensor networks. In *CIKM*, pages 1471–1472, California, USA, Oct. 2008.

[41] B. Yang, C. Guo, Y. Ma, and C. S. Jensen. Toward personalized, context-aware routing. *VLDB Journal*, 24(2):297–318, 2015.

[42] Y. Yang, H. Gao, J. X. Yu, and J. Li. Finding the CostOptimal Path with Time Constraint over TimeDependent Graphs. In *VLDB*, pages 673–684, Hangzhou, China, Sept. 2014.