Hindawi
Journal of Advanced Transportation
Volume 2017, Article ID 5656323, 14 pages
https://doi.org/10.1155/2017/5656323

Research Article

A Gradual Approach for Multimodel Journey Planning: A Case Study in Izmir, Turkey

Ferištah Dalkılıç,[1] Yunus Doğan,[1] Derya Birant,[1] Recep Alp Kut,[1] and Reyat Yılmaz[2]

[1]*Department of Computer Engineering, Faculty of Engineering, Dokuz Eylül University, Izmir, Turkey*
[2]*Department of Electrical & Electronics Engineering, Faculty of Engineering, Dokuz Eylül University, Izmir, Turkey*

Correspondence should be addressed to Ferištah Dalkılıç; feristah@cs.deu.edu.tr

Received 29 June 2017; Revised 2 November 2017; Accepted 8 November 2017; Published 12 December 2017

Academic Editor: Emanuele Crisostomi

Planning a journey by integrating route and timetable information from diverse sources of transportation agencies such as bus, ferry, and train can be complicated. A user-friendly, informative journey planning system may simplify a plan by providing assistance in making better use of public transportation. In this study, we presented the service-oriented, multimodel Intelligent Journey Planning System, which we developed to assist travelers in journey planning. We selected Izmir, Turkey, as the pilot city for this system. The multicriteria problem is one of the well-known problems in transportation networks. Our study proposes a gradual path-finding algorithm to solve this problem by considering transfer count and travel time. The algorithm utilizes the techniques of efficient algorithms including round based public transit optimized router, transit node routing, and contraction hierarchies on transportation graph. We employed Dijkstra's algorithm after the first stage of the path-finding algorithm by applying stage specific rules to reduce search space and runtime. The experimental results show that our path-finding algorithm takes 0.63 seconds of processing time on average, which is acceptable for the user experience.

1. Introduction

Metropolitan areas have some prevalent transport problems such as traffic congestion, parking difficulties, and emission of pollutant and greenhouse gases. Public authorities support stronger use of public transportation systems for mitigating traffic congestion and reducing carbon emissions. Enormous development has been taking place in public transportation systems of metropolitan areas around the world since the last decades. Consequently, use of public transportation systems is getting complicated. Mostly, more than one of the transportation agencies are operating in metropolitan areas. Route and timetable information of these services is available on their websites, but there is no information about the connection of other forms of transportation. It is not easy to integrate route and timetable information from diverse sources of agencies to plan a journey as a whole.

The usability of public transportation systems can significantly be enhanced by assisting people in making better use of public transportation. User-friendly and informative journey planning systems have become important by providing suggestions for alternative routes. The goal of this study is to present an Intelligent Journey Planning System (IJPS), to assist commuters in planning their trips. Our motivation is to reduce dependence on personal automobiles and to encourage wider use of public transport. In consequence of this work, we aim to reduce noise and carbon dioxide emissions and avoid traffic jams and congestions.

Journey planner systems like TfL of London, RATP of Paris, AnachB of Austria, and so on are in use in many metropolitan cities throughout the world. Nevertheless, almost all of them are commercial projects. Despite that, there are also many academic studies focused on solving problems regarding transportation networks, including earliest-arrival problem and multicriteria problem in the literature. A significant part of these studies stands on graph-based techniques, one of which is Dijkstra's algorithm [1]. Multilabel correcting (MLC) algorithm [2], transfer patterns [3], contraction hierarchies (CH) [4], transit node routing (TNR) [5], and trip-based public transit routing (TB) [6] can exemplify this category.

Other studies, including round based public transit optimized router (RAPTOR) [7] and connection scan algorithm (CSA) [8], are not Dijkstra-based. They use arrays to organize timetable information or elementary timetable connections. Some studies perform precomputations to use precomputed data for table lookups in shortest-path queries [3–5], while others do not rely on preprocessing [7, 8].

In cities with insufficient subway infrastructure such as Izmir, bus lines meet a significant part of the public transportation needs. For instance, while maximum of 8 metro lines intersect in a transfer point in Paris, 55 bus lines intersect in a transfer point in Izmir. This increases the density and complexity of the transportation network. The main contribution of this study is a proposal of a novel gradual path-finding algorithm (GPFA) to solve the multicriteria problem in such dense networks. GPFA has a hybrid approach that utilizes the techniques of some efficient algorithms including round based public transit optimized router, transit node routing, and contraction hierarchies on transportation graph and modifies Dijkstra's algorithm for fast computation of the queries. Second, IJPS is introduced to indicate alternative routes by considering modal transfers and user preferences. The other intention is to inform the passengers about point of interests (POIs), car parks, and events (concerts, exhibitions, etc.) located on their routes. Thus, passengers will be encouraged by the social and cultural activities carried out in the city, thanks to this unique feature.

The structure of the paper is follows: Section 2 states the multicriteria problem. Section 3 discusses existing techniques that are applicable to solving this problem. Section 4 introduces GPFA, our main contribution. Section 5 gives an overview of the system architecture and provides the implementation details. Section 6 presents the scenario analysis results, and, finally, Section 7 concludes the paper with a summary of the work as well as future directions.

2. Problem Definition

In graph theory, a graph G is a set of vertices (nodes) V, and a set of edges E has the form (u, v), where $u, v \in V$ are ordered distinct vertices. A path p is a sequence of consecutive nodes and edges that are formulated as $v_1, e_1, v_2, e_2, \ldots, e_j, v_{(j+1)}$ for every i $(1 \leq i \leq j)$; the edge e_i connects v_i and $v_{(i+1)}$. The length of the path is equal to i, which is the number of edges traversed. A path is defined as the *shortest path* from the origin o to the destination d if the path starts with the node o and ends with the node d and has the shortest length among all paths from o to d.

Finding all possible paths between any two nodes in a graph with a large number of nodes is a classic example of problems in the field of computational complexity theory [9]. Such problems require very large numbers of computations and memory addresses to solve. Instead of finding all paths, dealing with just the best K path is generally the preferred technique [10, 11]. The K-shortest-path problem is an extension algorithm of the single-source shortest-path problem. The single-source shortest-path problem finds shortest paths from an origin node o to destination D (D is equalized to V, which is the set of all nodes in the graph). Yen's algorithm is one of the fundamental works dealing with the K-shortest-path problem [12]. It finds not only the shortest path, but also the second shortest, third shortest, and so on up to the Kth shortest path with increasing cost. It uses Dijkstra's algorithm [1], or any other shortest-path algorithm, to find the best K path. Hundreds of studies offer new solutions or improve the existing algorithms [13–15].

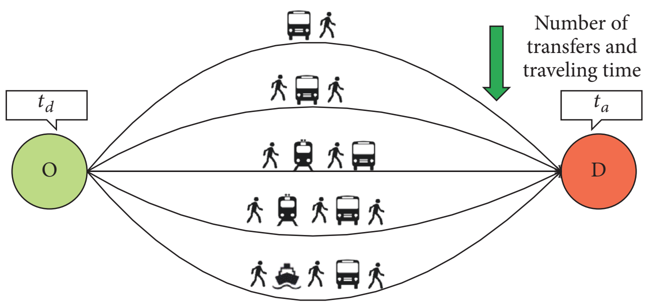


FIGURE 1: The scope of the problem.

For public transportation networks, there are two well-known problem definitions [16]. Earliest-arrival problem only considers the time as the optimization criterion. When the number of transfers or other criteria are considered, the problem is then referred to as the multicriteria problem. In this study, we want to answer the question illustrated in Figure 1: *"for a given departure time t_d, how can one get from origin o to destination d with the least number of transfers and the shortest traveling time by considering modal preferences and an acceptable walking distance?"* in an acceptable response time. In this case, the best path is not just the shortest one according to traveling time; here, transfer count and user preferences have to be considered, too.

The multicriteria problem has a host of challenges to overcome. Compared to static road networks, public transport networks are more difficult to model, because they are less hierarchically structured on large metropolitan areas [17]. Additionally, these networks are event-based and require modeling in a time-dependent approach. Many algorithms that perform well in road networks cannot perform well or even fail in transportation networks. While basic routing only considers static edge weights associated with travel times, transportation networks employ a travel time function depending on the chosen departure time. In multicriteria optimization, it is necessary to consider multiple criteria including the travel time and the number of transfers. Also depending on user preferences, acceptable walking distance, modal transfers, and vehicle type restrictions must be considered, too. Instead of suggesting only the best route, some favorable alternatives should be provided to the user. Furthermore, precomputing is much more complicated in public transportation routing and precomputed data must be filtered effectively according to variable query parameters.

3. Related Works

There have been many recent advances in journey planning on public transportation systems in recent years. Efficient algorithmic techniques (usually based on Dijkstra's algorithm) have been further developed for public transportation systems. Bast et al. gave an excellent survey of these techniques [16]. Mainly, time-expanded and time-dependent approaches have been used to model timetable information [2]. In time-expanded approach [18] every node models a departure or arrival event. The consecutive departure and arrival events are connected by connection edges. In time-dependent approach [19], every node models a station and an edge is assigned between two nodes if a direct connection between corresponding two stations exists. The weights of the edges are assigned "on the fly" associated with the travel time function. For the earliest-arrival problem, Pyrga et al. showed that time-expanded approach constructs large graphs and yields poor query performance compared to the time-dependent approach [2]. Therefore, in this study we focused on the time-dependent model.

One of the algorithms in time-dependent model is multilabel correcting (MLC) algorithm that aims to find all pareto-optimal solutions [2]. A path is called pareto-optimal if another path does not dominate it. MLC can handle multiple criteria that are modeled by multiple edge labels for each criteria in a generalized Dijkstra's algorithm. Another technique in time-dependent model is transfer patterns [3]. It precomputes and stores each sequence of intermediate transfer stations that can lead to an optimal route for every source station A to all stations B reachable from A. The set of these sequences of transfers is called transfer pattern that achieves very fast query times. Contraction hierarchies (CH) is also a time-dependent technique that performs precomputation for node contraction [4]. It gradually removes nodes from the graph and adds shortcuts to preserve the shortest paths. CH has been applied to the multimodal transportation networks [20] and the time-dependent model [21]. Algorithm maintains multiple labels per vertex to ensure minimum transfer counts. Transit node routing (TNR) is yet another technique that determines a small set of the transit nodes that every shortest path passes through at least one of these nodes [5]. Distances are precomputed from each node to its closest transit nodes (access nodes) to use for table lookups in shortest-path queries. TNR has been applied to public transit journey planning in [22]. Trip-based public transit routing (TB) is a different approach that uses a graph where nodes represent trips (not stops) and edges represent possible transfers between trips [6, 23]. A two-phase preprocessing step is required to compute all possible transfers between trips and discarding the superfluous transfers. TB determines the optimal routes in increasing order of transfers.

Round based public transit optimized router (RAPTOR) [7] is a different approach that is not Dijkstra-based. Instead of using a graph, it uses arrays of timetable information including stops, trips, and routes. It does not rely on preprocessing and operates in rounds by minimizing the transfer count and the arrival time. With round i, the journeys consisting of exactly i transfers are computed. Another non-graph-based algorithm is connection scan algorithm (CSA) [8, 24]. It is not based on trips and routes (as RAPTOR); instead elementary connections of the timetable are organized in an array that is sorted by departure time. This array is scanned once per query, which is very efficient in practice. Dib et al. proposed a heuristic approach whereby a Genetic Algorithm (GA) was combined with a Variable Neighborhood Search (VNS) to solve the multicriteria shortest-path problem in multimodal networks [25]. Experimental results show that the computational time is not prohibitive to integrate it within an online journey planning system.

We propose a gradual path-finding algorithm for public transportation networks that utilizes the published techniques of different algorithms. Like RAPTOR, our algorithm has a round base approach depending on transfer count, but based on graph-based Dijkstra algorithm. The transfer centers in our algorithm (the main items of Izmir transportation) correspond to the transit nodes in TNR technique. In preprocessing step that is explained in Section 4.1, we found and recorded the available paths that have up to two transfers among all the transfer centers. We inspired from contraction hierarchies to insert shortcuts between every transfer center pair, which is connected via a direct route.

4. Gradual Path-Finding Algorithm

In this study, two main parts of IJSP have been developed in a service-oriented model. These main components are the Update Service and the Journey Planning Web Service (JPWS). Update Service is a Windows service application that executes some preprocessing tasks every night. JPWS is a Windows Communication Foundation (WCF) web service and acts as a routing engine for clients.

Preprocessing tasks consist of four steps, namely, collecting and transforming transport data, determining nearby stops, accessible transfer centers, and route planning among all transfer center pairs. JPWS has a method that takes the journey parameters from user applications and produces alternative paths by using the GPFA. The algorithm schema of Update Service and JPWS is given in Figure 2. Sections 4.1 and 4.2 give detailed explanation of the preprocessing tasks and the GPFA, respectively.

4.1. Preprocessing Tasks. Standing on a common data format makes an application able to work in all transit systems for which open transit data has been released and the common data is available to any developer to use. General Transit Feed Specification (GTFS) is a common data format, which has six required feeds and seven optional feeds for representing the stops, routes, trips, and other schedule data of a transportation network [26]. During the last years, the GTFS has become the most popular format and there are 1000 transit agencies providing GTFS data as of June, 2017. Transportation agencies in Izmir store transportation data in different formats. Update Service collects the current data from transportation agencies, integrates them in accordance with GTFS format, and stores them into the database. All required feeds are generated for Izmir transportation network.

To make point-to-point queries in a transit network, some foot edges are required. This is so that any stage of the journey can be made on foot, or passengers may walk between stops while transferring between two lines. Foot edges also provide links between each part of the transportation network (bus,

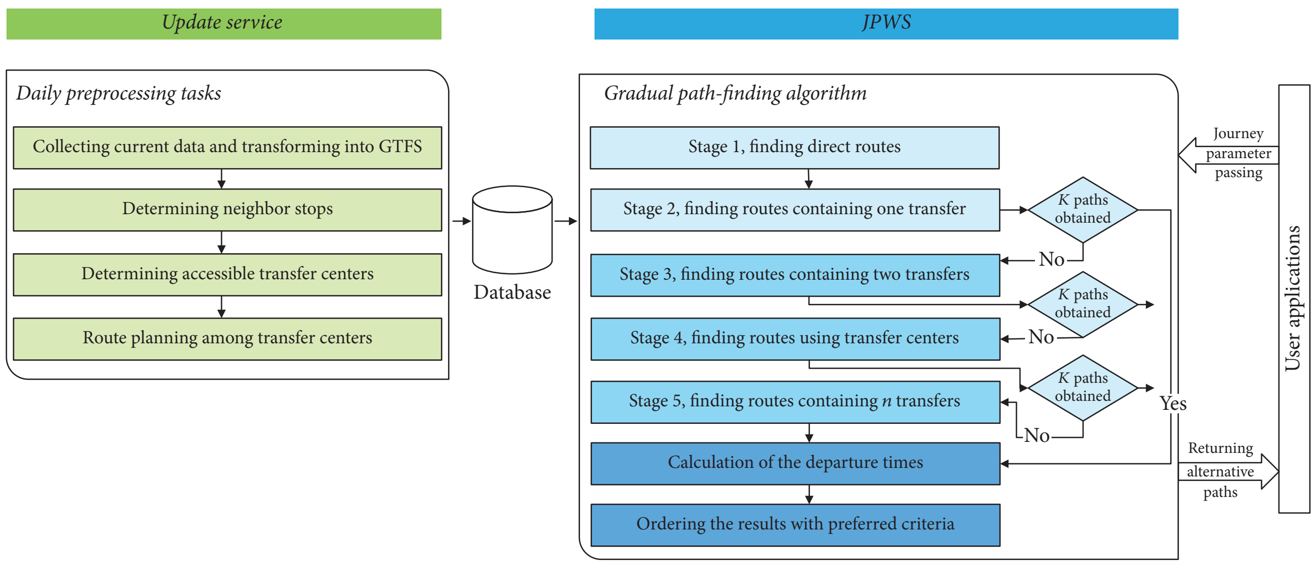

Update service
Daily preprocessing tasks
Collecting current data and transforming into GTFS
Determining neighbor stops
Determining accessible transfer centers
Route planning among transfer centers
Database
JPWS
Gradual path-finding algorithm
Stage 1, finding direct routes
Stage 2, finding routes containing one transfer
Stage 3, finding routes containing two transfers
Stage 4, finding routes using transfer centers
Stage 5, finding routes containing n transfers
Calculation of the departure times
Ordering the results with preferred criteria
K paths obtained
No
Yes
Journey parameter passing
Returning alternative paths
User applications

Figure 2: Algorithm schema of JPWS and Update Service procedures.

train, subway, and ferry) in the resulting multimodal network G. Two stops u and v are labeled as nearby stops by adding foot edges ($(u, v) \in E$ and $(v, u) \in E$) between them if distance (u, v) is less than the maximum walking distance. Nearby stops are computed using the Euclidean distance.

TNR technique precomputes the connections from each potential origin or destination to its access transit nodes and between all pairs of transit nodes [5]. In Izmir, the main transportation lines are train, subway, and ferry. Passengers using these lines can transfer to a bus line to reach their neighborhood. Currently, there are 20 transfer centers, located mostly around train, subway, and ferry stations, and each includes up to 34 stops. A passenger who travels between two remote points is likely to pass through one or more of these transfer centers. Our path-finding algorithm uses this logic by assuming the transfer centers to be *"access transit nodes"* in TNR technique. Thus, determination of accessible transfer centers for each stop is a preprocess, which runs after the update of transportation data. In this study, an accessible transfer center for a stop means that there is at least one path that has up to two transfers between the stop and transfer center.

The path-finding preprocess calculates and records the available paths that have up to two transfers among all the transfer centers. If we refer to the transportation graph as G, which consists of a set of transfer centers T, and each transfer center has a set of stops S, a set of paths $P \subseteq s_n \in t_i \times s_m \in t_j$ for every i and j ($i \neq j$ and $t_i, t_j \in T$) is calculated and stored in a database table. $p \in P$ is a path that has no or only one transfer. Notations describes the notations used in this paper.

4.2. Proposed Algorithm. In this study, each stop (bus, train, subway, and ferry) is represented as a node; a line connecting two consecutive stops in a particular direction is represented as a directed edge, thereby forming a transportation graph. If two stops are in walking range, they are connected to each other with foot edges. Transferring between different lines occurs through foot edges. The edge weights of the vehicle edges are represented as average travel times to cover the specified transport segment. The average travel time for each edge e is computed by a travel time function $f(e)$ using the statistical travel time of the transport segment at a given time interval (e.g., [0:00–0:59], [1:00–1:59], . . . , [23:00–23:59]). The edge weights of the foot edges are computed by taking the geographic length of the transport segment and assuming an average walking speed s_w of a pedestrian as 6 km/h. After the JPWS is launched, route and time table data, which are used to create a static transportation graph, are loaded into the memory once for use in the query execution. For each individual query, a new auxiliary graph is constructed from the static graph without any database queries. Then this auxiliary graph is used to compute alternative routes. Nodes and edges of user-unpreferred transportation modes are not included in auxiliary graph.

Transferring between two lines takes time, including walking between stops and waiting at the connecting line. Passengers naturally wish to reduce this waiting time [27]. Additionally, passengers who are disabled, older, or carrying heavy luggage or children usually prefer trips with fewer transfers. Therefore, we developed the GPFA, which produces a path primarily with fewer transfers and then according to other criteria. The path-finding operation is completed in at most five stages. If the necessary number of paths to achieve K-shortest paths cannot be identified, the operation proceeds to the next stage. Calculation of the departure times of the produced paths, filtering, and ordering of the results is then undertaken. The stages of the GPFA appear in Figure 2. If more than K paths are achieved by the last stage, only K journeys with highest score are shown to the user.

There may be multiple origin stops and multiple destination stops, which correspond to selected points. Therefore, the problem is reduced to the single-source shortest-path problem by running the modified Dijkstra's algorithms for each origin stop to all destination stops in Stages 2, 3, and 5. We modified Dijkstra's algorithm by determining edge weights on the fly to favor edges leading toward the destination. In the shortest-path calculation, the total travel time to cover the path forms the total cost of the path. So, we can obtain the fastest path which has the transfer count we currently work on. In our transportation graph, direct routes are used as shortcuts between transfer center nodes to bypass intermediate nodes in journey planning. This approach is an adaption of contraction hierarchies [21].

We have illustrated an example scenario in Figure 3 to explain and clarify some definitions. A passenger wants to arrive at the Dokuz Eylül University Faculty of Engineering from the Faculty of Medicine, which is located in a different district. He is ready at the origin at 09:30 AM. To keep the drawing simple, we only display the transfer stops. For each transportation connection in the itinerary, we give the traveling time in minutes and time of arrival at the next transfer stop (e.g., (20, 10:07)). The GPFA suggests the route alternatives by considering the transfer count and travel time. In the first stage, the algorithm checks if a common line exists between L_{O} and L_{D}, which implies a direct route between the origin and the destination points. In this scenario, there is no direct line that is connecting the origin and the destination. In the second stage of the algorithm, routes containing only one transfer are produced. In this case, one of the origin lines, namely, Bus-554, is connected to one of the destination lines, namely, Bus-304, by a walk. The passenger waits 5 minutes to get on the first service of the Bus-554, when he arrived at the origin stop. After he travels 32 min by the Bus-554, he gets off at the stop s_2 and walks 1 min to arrive at the stop s_3. There, he waits 1 min to get on the Bus-304. After 40 min of travel on the Bus-304, he then arrives at the destination at 10:49 AM. In the third stage of the algorithm, routes containing two transfers are discovered. According to one of the suggested paths in this stage, the passenger can start the journey by the Bus-554, and then transfers to the Metro-31 and the Bus-490, respectively, to arrive at the destination at 10:53 AM. In the next stage, the GPFA finds the paths that have up to two transfers, passing through at least one transfer center. In this scenario, the suggested path includes the Bus-554, Metro-31, Train-22, and the Bus-878, respectively. The last stage of the algorithm that finds n transfer paths does not run in this example, since K paths (K is assumed to be 3) are obtained already.

Some of the origin lines (e.g., Bus-950) are night-only services which operate during the late night hours. These lines do not take a part in any of the results for the chosen departure time. The algorithm eliminates all the paths that contain lines causing unacceptable waiting times. Thus, timetable restricts the route selection from the viewpoint of time or space as it is in [28, 29]. In the following subsections, the stages of GPFA will be explained in detail.

4.2.1. Finding Direct Routes. Direct paths, which have no transfers, are calculated by using origin lines L_{O} and destination lines L_{D}. For a line l, if $l \in L_{\mathrm{O}}$ and $l \in L_{\mathrm{D}}$, we can say that l is a line that reaches the destination from the origin with no transfers. A *Path* obtained at this stage has only one *SubPath*, which contains line l. Each line that forms a direct path is stored in the used-line list L_{U} to prevent using it in calculating paths with transfers. Direct paths are obtained by using static data without any database queries and shortest-path algorithms.

4.2.2. Finding Routes Containing One Transfer. In this stage of the algorithm, the aim is finding a path that starts with an origin line and ends with a destination line. The origin and destination lines must be connected by a transfer stop or there must be a walk between two lines. A modified version of Dijkstra's algorithm is developed to obtain paths that start with an origin line and end with a destination line. The algorithm allows starting with an origin line, transferring to a destination line, and arriving at the destination stop using that line (Figure 4) according to the given restrictions below:

(i) $l_1 \in L_{\mathrm{O}}, l_2 \in L_{\mathrm{D}}$.

(ii) $l_1, l_2 \notin L_{\mathrm{U}}$.

(iii) $s_1 \in S_{\mathrm{O}}, s_2 \in S_{\mathrm{D}}$, and $s_3, s_4 \notin S_{\mathrm{O}}, S_{\mathrm{D}}$.

In our algorithm, each node (stop) has a weight vector with the size of its outbound line count. With our approach, in contrast to a conventional graph, edge weights are not known at the beginning of the process. Weight values are assigned according to restrictions when running the algorithm. The weights of edges that do not meet these requirements remain as infinity, while the other weights are identified by the travel time function. The minimum-cost path that corresponds to the rules is obtained as a result and returned as a *Path* object.

The modified version of Dijkstra's algorithm runs for each origin line l. We use two cost values C_{T} and C_{W} in the algorithm. C_{T} is used to find paths with fewer stops when C_{W} is employed to identify paths with less walking. After finding a path with one transfer, new paths are generated by replacing the second line l_2 with *parallel lines* using static data. A parallel line is a line that passes through both the first stop and the last stop of l_2 in the produced *SubPath*. Parallel lines are then added to L_{U}. If the necessary path count cannot be found, the algorithm runs again for l_1 to search for an alternative path without using l_2 and its parallel lines. This stage continues until finding K paths, or a path with one transfer can no longer be found for any line l_1. Pseudocode of the modified Dijkstra's algorithm for routes that contain one transfer is given in Algorithm 1.

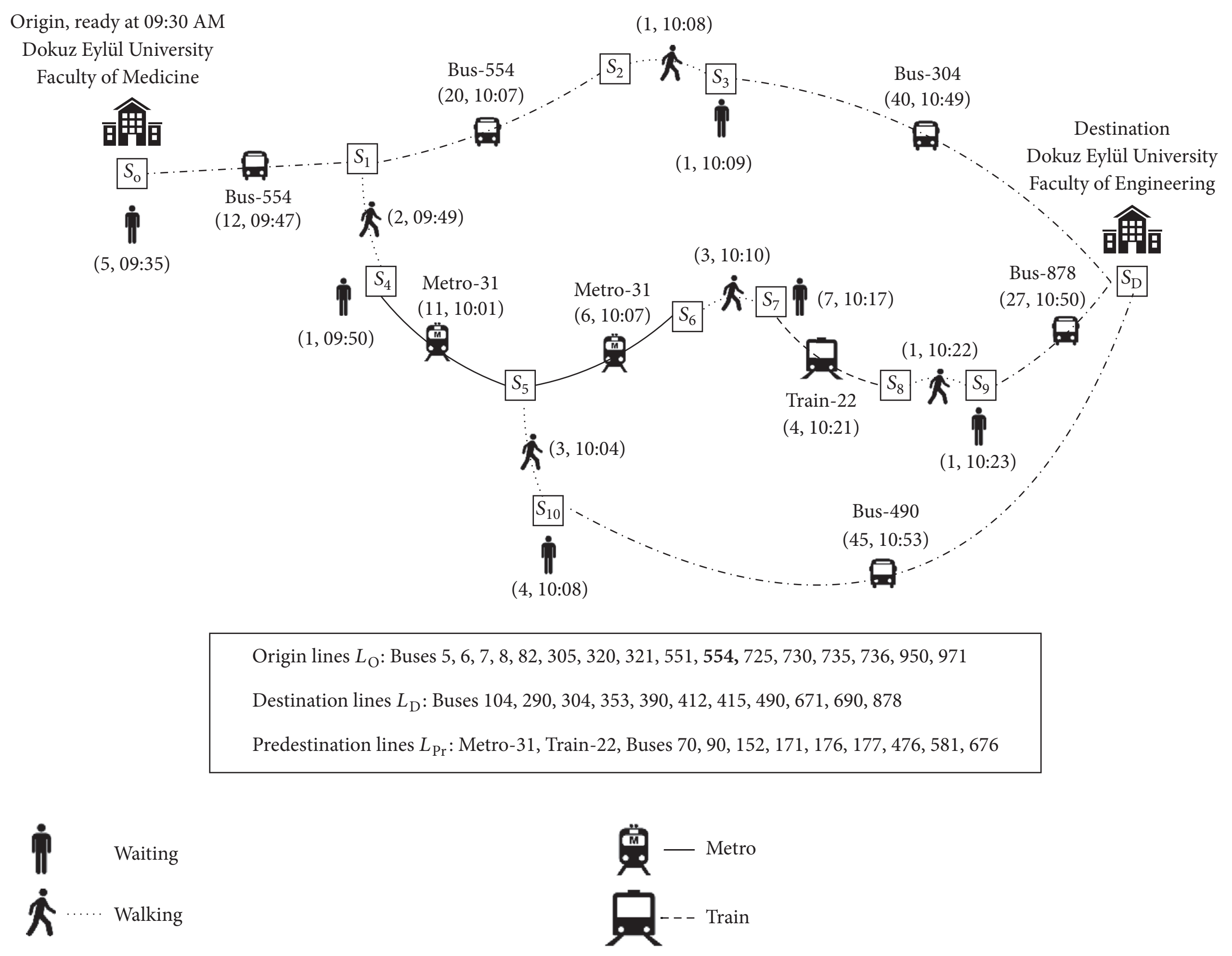
4.2.3. Finding Routes Containing Two Transfers. If the necessary path count (K paths) cannot be found in Stages 1 and 2 (Figure 2), the process continues by identifying routes that contain two transfers. At the beginning of this stage, predestination lines L_{Pr}, which intersect with a destination line at a stop directly or via a walk, are determined. The algorithm finds the paths that start with an origin line, continue with a predestination line, and end with a destination line. Arbitrary lines can be connected at a transfer stop or there may be a walk between two lines (Figure 5) according to the given restrictions:

(i) $L_{\mathrm{Pr}} \cap L_{\mathrm{O}} = \varnothing, L_{\mathrm{Pr}} \cap L_{\mathrm{D}} = \varnothing$.

(ii) $l_1 \in L_{\mathrm{O}}, l_2 \in L_{\mathrm{Pr}}, l_3 \in L_{\mathrm{D}}$.

(iii) $l_1, l_2, l_3 \notin L_{\mathrm{U}}, s_1 \in S_{\mathrm{O}}, s_2 \in S_{\mathrm{D}}$.

(iv) $s_3, s_4, s_5, s_6 \notin S_{\mathrm{O}}, S_{\mathrm{D}}$.



Origin lines L_O: Buses 5, 6, 7, 8, 82, 305, 320, 321, 551, **554,** 725, 730, 735, 736, 950, 971

Destination lines L_D: Buses 104, 290, 304, 353, 390, 412, 415, 490, 671, 690, 878

Predestination lines L_{Pr}: Metro-31, Train-22, Buses 70, 90, 152, 171, 176, 177, 476, 581, 676

Figure 3: An example transportation network.

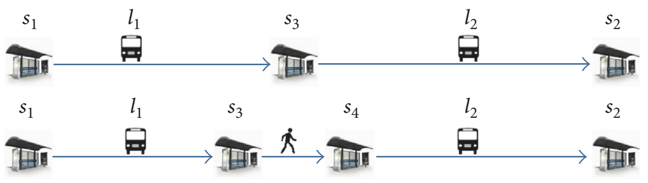


Figure 4: Illustration of routes with one transfer.

Pseudocode of the modified Dijkstra's algorithm for routes that contain two transfers is given in Algorithm 2. In this version of Dijkstra's algorithm, each node has a *transfer count* property in addition to *distance* property. Transfer count property is used to hold the transfer count of the minimum-cost path of each node. After a path is found with two transfers, new paths are generated by replacing the second line l_2 with its parallel lines and then by replacing the third line l_3 with its parallel lines. l_2, l_3, and their parallel lines are then added to L_U. If the necessary count of paths still cannot be found, the algorithm runs again for l_1 to find an alternative path without using lines that are in L_U. This stage continues until producing K paths, or a path with two transfers can no longer be found with any l_1.

4.2.4. Finding Routes Using Transfer Centers. If the necessary path count with zero, one, or two transfers cannot be found between the origin and destination stops, a search must be conducted for paths over two transfers. Considering the traffic infrastructure in Izmir, a person making a trip between two distant points is likely to pass through a transfer center. At this stage, the aim is finding paths with over two transfers and passing through at least one transfer center. As noted above, determining the accessible transfer centers for all stops and the path-finding preprocess between each transfer center are accomplished by Update Service.

A path-finding process using transfer centers can occur in two cases as illustrated in Figure 6. In the first case, origin and destination stops have common accessible transfer centers (Figure 6(a)). In this case, paths with zero transfers and one transfer are calculated between the origin stop and common transfer center by the techniques noted for Stages 1 and 2. If no path is found, paths with two transfers are calculated. The

```
Inputs: origin, origin line l, $L_U$, $L_D$
for each node n in Graph:
-n.dist ≔ infinity;
-n.previous ≔ undefined;

origin.dist ≔ 0;
Q ≔ Priority queue according to distance;
enqueue origin into Q;

while Q.isEmpty != true:
-u ≔ node with min distance in Q;
-remove u from Q;

-for each outbound edge e of node u:
--v ≔ node reachable from u with edge e;
--e.weight ≔ infinity;
--if e.Line $\notin L_U$:
---if u.previous != null: prev_e ≔ edge used for reaching to u
----if prev_e.Line $\in L_D$:
-----if e.Line = prev_e.Line: e.weigth ≔ $f(e)$;
----else
-----if e is a line:
------if e reaches a destination stop: e.weigth ≔ $f(e)$;
------else if e.Line = prev_e.Line: e.weigth ≔ $f(e) + C_T$;
-----else if e is foot-edge: e.weigth ≔ $f(e) + C_W$;
---else//u.previous = null:
----if e.Line is l: e.weigth ≔ $f(e)$;

-for each outbound edge e of node u:
--v ≔ node reachable from u with edge e;
--dist_v ≔ u.dist + e.weigth;
--if dist_v < v.dist:
---dequeue v from Q with key v.dist;
---v.dist ≔ dist_v;
---v.previous ≔ u;
---enqueue v into Q with key v.dist;

S ≔ empty sequence
u ≔ target
while u.previous is not null:
-insert u into S;
-u ≔ u.previous;
```

Algorithm 1: Finding routes containing one transfer.

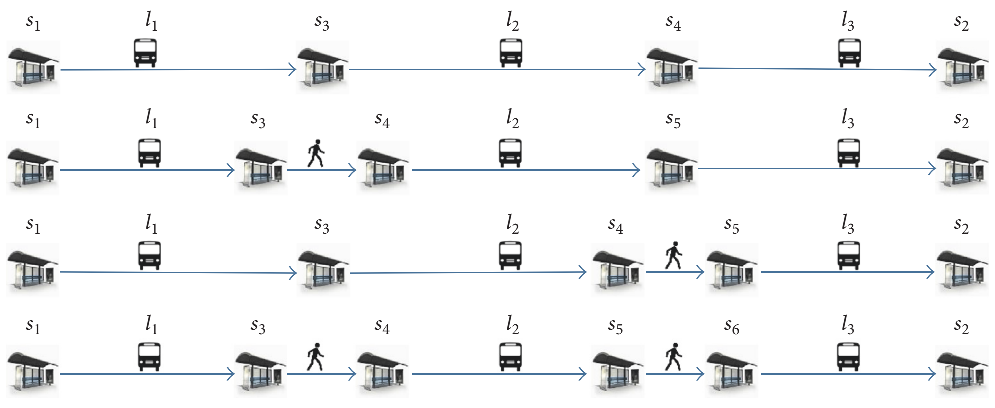

s_1 l_1 s_3 l_2 s_4 l_3 s_2
s_1 l_1 s_3 s_4 l_2 s_5 l_3 s_2
s_1 l_1 s_3 l_2 s_4 s_5 l_3 s_2
s_1 l_1 s_3 s_4 l_2 s_5 s_6 l_3 s_2

Figure 5: Illustration of routes with two transfers.

```
Inputs: origin, origin line l, $L_U$, $L_D$, $L_{Pr}$
for each node n in Graph:
-n.dist ≔ infinity;
-n.trCnt ≔ 0;  // Transfer Count
-n.previous ≔ undefined;

origin.dist ≔ 0;
Q ≔ Priority queue according to distance;
enqueue origin into Q;

while Q.isEmpty != true:
-u ≔ node with min distance in Q;
-remove u from Q;

-for each outbound edge e of node u:
--v ≔ node reachable from u with edge e;
--e.weight ≔ infinity;
--if e.Line $\notin L_U$:
---if u.previous != null: prev_e ≔ edge used for reaching to u;
----if e is a line:
-----if ((u.trCnt = 2 and e.Line = prev_e.Line) or
         (u.trCnt = 1 and e.Line != prev_e.Line and e.Line $\in L_D$) or
         (u.trCnt = 1 and e.Line = prev_e.Line) or
         (u.trCnt = 0 and e.Line != prev_e.Line and e.Line $\in L_{Pr}$) or
         (u.trCnt = 0 and e.Line = prev_e.Line)): e.weigth ≔ $f(e)$;
----else // e is a foot-edge
-----if ((u.trCnt = 0 and prev_e is a line) or
-----     (u.trCnt = 1 and prev_e is a line)): e.weigth ≔ $f(e)$;
-----else e.weigth ≔ infinity;
---else // u.previous = null:
----if e.Line is l: e.weigth ≔ $f(e)$;

-for each outbound edge e of node u:
--v ≔ node reachable from u with edge e;
--prev_e ≔ edge used for reaching to u;
--dist_v ≔ u.dist + e.weigth;
--if dist_v < v.dist:
---dequeue v from Q with key v.dist;
---v.dist ≔ dist_v;
---v.previous ≔ u;
---v.trCnt ≔ u.trCnt;
---if e.Line != prev_e.Line: v.trCnt ≔ u.trCnt + 1;
---enqueue v into Q with key dist_v;

S ≔ empty sequence;
u ≔ target;
while u.previous is not null:
-insert u into S;
-u ≔ u.previous;
```

ALGORITHM 2: Finding routes containing two transfers.

same process is undertaken between the common transfer center and destination stop. P_1 is the set of paths found between the origin and transfer center; P_2 is the set of paths found between the transfer center and destination. The result set P is obtained by cross production of P_1 and P_2, where $p \in P$ and $p \subset P_1 \times P_2$.

In the second case, origin and destination stops do not have a common transfer center (Figure 6(b)). Paths with zero transfers and one transfer are calculated between the origin stop and accessible transfer center T_1; paths with two transfers are calculated if necessary. The same procedure is conducted for transfer center T_2 and destination stops. Paths between the two transfer centers are calculated and stored into the database by Update Service. P_1 is the set of paths between the origin and transfer center T_1; P_2 is the set of paths between the transfer center T_1 and transfer center T_2 stored in the

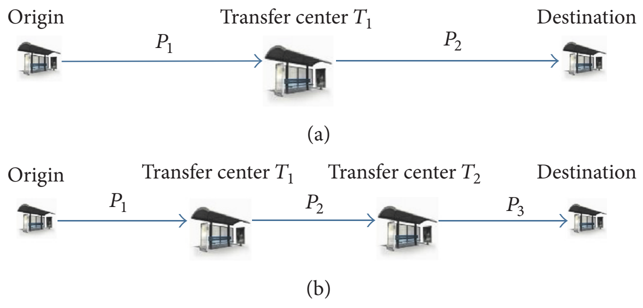


Figure 6: Path-finding process between two stops through one transfer center (a) and two stops through two transfer centers (b).

database; P_3 is the set of paths between the transfer center T_2 and destination. In this case, the result set P is obtained by cross production of P_1, P_2, and P_3, where $p \in P$ and $p \subset P_1 \times P_2 \times P_3$.

During cross production, if the destination stop of P_1 differs from the start stop of P_2, a walk is inserted between them. Conversely, if the last line of P_1 is the same as the first line of P_2, the two parts are combined. The same procedure is performed for P_2 and P_3.

4.2.5. Finding Routes Containing n Transfers. If the K paths still cannot be produced, we use another modified version of Dijkstra's algorithm to find minimum-cost paths that have over two transfers. We employ some constants to change the behavior of the algorithm. C_{Tr} is the constant value added onto edge weight at each line change. If the transferred line is not a destination line but a predestination line, *coefficient1* is added to edge weight. If the transferred line is in neither L_{D} nor L_{Pr}, *coefficient2* is added to edge weight. *coefficient2* > *coefficient1*, and these values are determined according to edge weights in the graph. C_{W} is added to weights of foot edges, so a path with less walking also has lower cost. Pseudocode of the modified Dijkstra's algorithm for routes that contain n transfers is given in Algorithm 3.

4.2.6. Algorithm Complexity. In our implementation, all unscanned reached nodes are stored using their tentative distance values as keys in a priority queue implemented by *SortedDictionary* < *TKey, TValue* > class, which has $O(\log n)$ search time complexity. Every node reachable from o is inserted and removed from the queue exactly once. Each node is scanned once at most, and each edge is relaxed once at most. Hence, the worst-case complexities of our modified Dijkstra's algorithms are $O((V + E) \times \log V)$. Average-case complexity is much better than the worst-case complexity thanks to our restrictions while visiting the nodes and relaxing the edges.

5. Case Studies on Izmir Public Transportation

With a population of over 4 million, Izmir is the third-largest city in Turkey. Each year, approximately 200,000 tourists visit Izmir, where four public transport agencies currently operate. Route and timetable information from these agencies is available on their websites, but there is no information about connections with other forms of transport.

In this study, we present the IJPS that stands on a service-oriented architecture (Figure 7) to assist domestic and foreign visitors, as well as locals, in efficient use of urban transportation. We selected Izmir, Turkey, as the pilot city for this system. IJPS indicates alternatives routes, transfer details, and departure and arrival times according to user preferences for any origin or destination point. The system provides optimal route choices according to multiple criteria, such as desired means of transport, maximum walking distance, least number of changes, and shortest traveling time. IJPS runs on different platforms to provide a wide range of usage anytime and anywhere. Applications are available in both English and Turkish for mobile web and desktop web portals, at kiosks, and for Android, iPhone, and Windows Phone platforms. Sample screens are shown in Figure 8, which display mobile Web and Web applications. IJPS supplies information including weather, traffic and road conditions, approximate taxi fares, activity centers, and events along the route.

At the beginning of the journey plan, origin and destination points are determined. Three methods are used in determining the origin and destination points: making a selection from a pool of predefined POIs, making selection from stop or station names, and marking the location on the map. Other parameters are the desired means of transport, preference regarding walking between transfers, maximum acceptable walking distance, and sorting criteria for the results. According to user preferences, event centers and POIs on the route can be visualized on the map. POIs along the route are presented as a list. Another list offered includes cultural centers, such as museums and exhibitions, and event centers, such as cinemas, theaters, and concert halls near the route. If the user selects a center from the list, events that are (or will be) taking place around the date and time of the journey are displayed.

Table 1: Visited edge counts for modified and pure Dijkstra's algorithms.

Stages	Visited edge count	Ratio%
2	364	1.32
3	2,169	7.89
4	5,833	21.21
5	21,288	77.39
Pure Dijkstra's algorithm	27,464	99.85

6. Scenario Analysis

Here, we present an experimental study, investigating the reduction in search space achieved by the GPFA. A sample query for the origin-destination pair "Evka 3 Transfer Center" and "Buca City Hall" has been executed to evaluate search spaces for the stages of the GPFA. Table 1 shows the visited edge counts among total 27,506 edges present in graph for the modified Dijkstra's algorithms which were used in Stages 2–5.

The visited edge counts increase over the upper stages of the algorithm. We reduced the search space to 1.32%, 7.89%,

```
Inputs: origin, origin line l, $L_U$, $L_D$, $L_{Pr}$
for each node n in Graph:
-n.dist ≔ infinity;
-n.previous ≔ undefined;

Q ≔ Priority queue according to distance;
for each origin stop s in $S_S$:
-s.dist ≔ 0;
-enqueue s into Q;

while Q.isEmpty != true:
-u ≔ node with min distance in Q;
-remove u from Q;
-for each outbound edge e of node u:
--v ≔ node reachable from u with edge e;
--e.weight ≔ infinity;
--if e.Line $\notin L_U$:
---if u.previous != null:
----prev_e ≔ edge used for reaching to u
----if e is a line:
-----if e.Line != prev_e.Line:
------if e.Line $\in L_D$: e.weight ≔ $f(e) + C_{Tr}$;
------else if e.Line $\in L_{Pr}$: e.weight ≔ $f(e) + C_{Tr}$ + coefficient1;
------else e.weight ≔ $f(e) + C_{Tr}$ + coefficient2;
-----else e.weight ≔ $f(e)$; // e.Line = prev_e.Line
----else if e is foot-edge:  e.weight ≔ $f(e) + C_W$;
---else e.weight ≔ $f(e)$; // u.previous = null

-for each outbound edge e of node u:
--v ≔ node reachable from u with edge e;
--dist_v ≔ u.dist + e.weigth;
--if dist_v < v.dist:
---dequeue v from Q with key v.dist;
---v.dist ≔ dist_v;
---v.previous ≔ u;
---decrease-key v in Q;
---enqueue v into Q with key dist_v;

S ≔ empty sequence;
u ≔ target;
while u.previous is not null:
-insert u into S;
-u ≔ u.previous;
```

Algorithm 3: Finding routes containing *n* transfers.

21.21%, and 77.39% for Stages 2–5, respectively; the pure Dijkstra's algorithm visits 99.85% of all edges. Reducing the search space to this degree underlines the good performance of the algorithm. Visualizations of the search spaces corresponding to the sample query and optimal paths obtained for Stages 2–5 appear in Figure 9.

To verify the accuracy, reliability, and consistency of the IJPS, we created a test data table; we executed 96,107 sample queries from the first stop of all available bus, train, subway, and ferry lines to the lines' last stops. The statistical details obtained from the test queries are shown in Table 2. Stages 1 and 2 were executed for 100% of all queries independently from the result count. The operation continued with Stage 3 for 69.2% of queries because the K-shortest path (K was assumed to be 5) could not be found in the first two stages. After Stage 3, 30.1% of queries were processed in Stage 4 to achieve the K-shortest path (K was here also assumed to be 5). Only 0.8% of queries that could not produce at least three paths continued to Stage 5. Approximately 30% of all queries ended in the first two stages; around 70% ended in Stage 3. In the first four stages, we observed average runtimes under 1 sec. But, in Stage 5, the average runtime was 8.47 sec because the rules applied at this stage did not restrict the search space as the other stages. This experimental results show that our path-finding algorithm takes 0.63 sec of processing time on average. It should be noted that all the runtimes contained

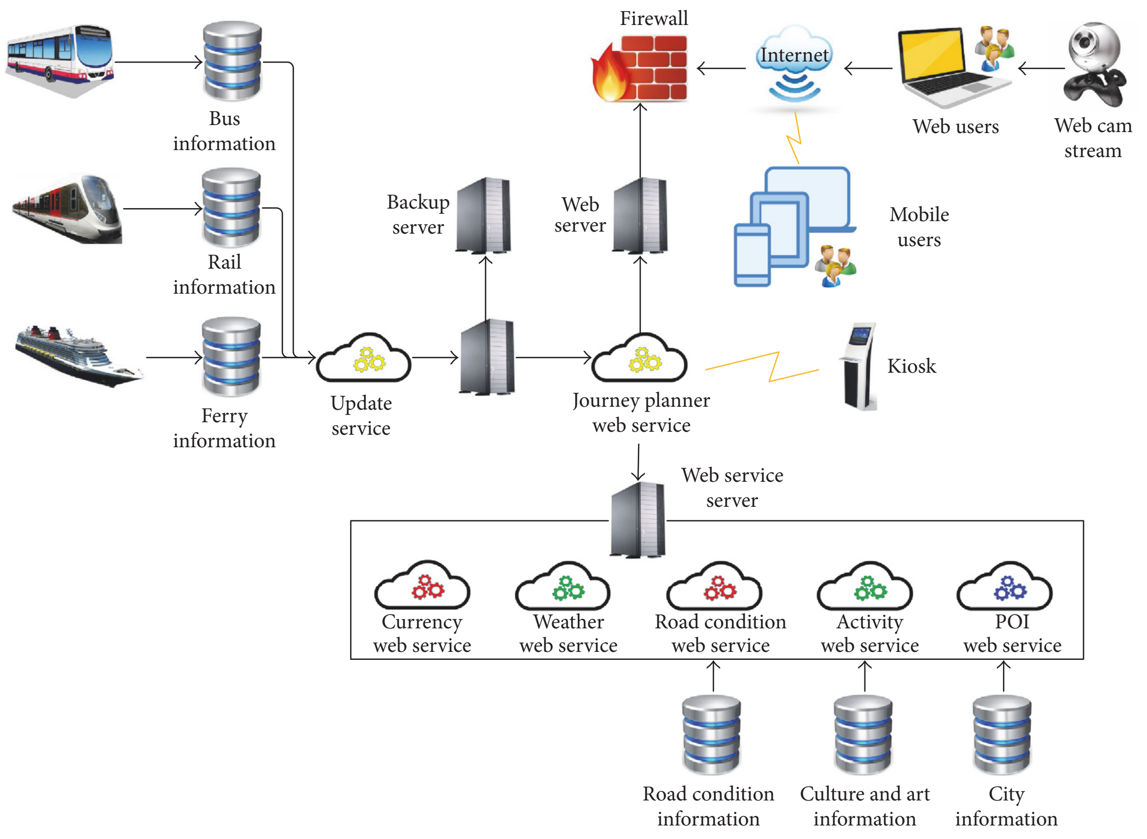
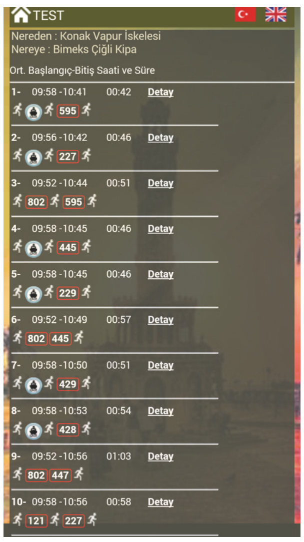


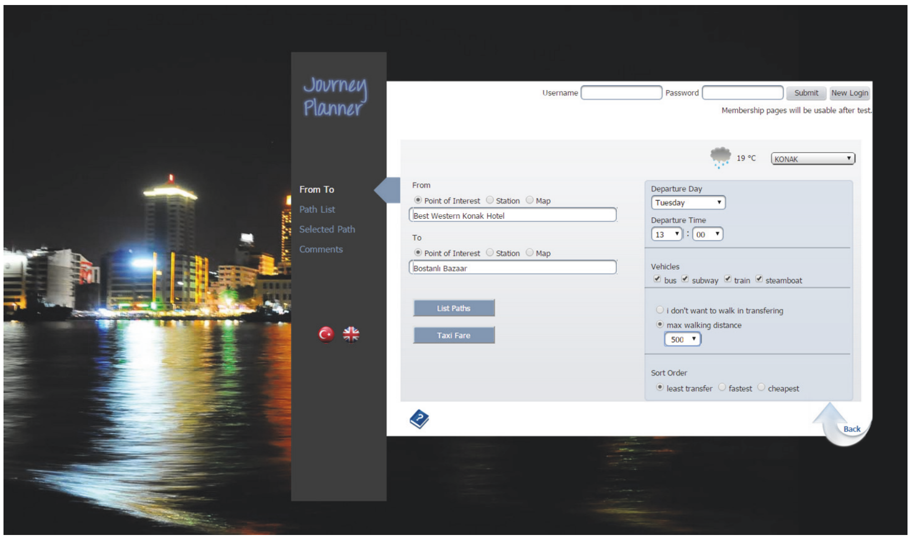
Figure 7: System architecture.


TEST

Nereden : Konak Vapur İskelesi
Nereye : Bimeks Çiğli Kipa

Ort. Başlangıç-Bitiş Saati ve Süre

1-	09:58 -10:41	00:42	Detay
	595		
2-	09:56 -10:42	00:46	Detay
	227		
3-	09:52 -10:44	00:51	Detay
	802 595		
4-	09:58 -10:45	00:46	Detay
	445		
5-	09:58 -10:45	00:46	Detay
	229		
6-	09:52 -10:49	00:57	Detay
	802 445		
7-	09:58 -10:50	00:51	Detay
	429		
8-	09:58 -10:53	00:54	Detay
	428		
9-	09:52 -10:56	01:03	Detay
	802 447		
10-	09:58 -10:56	00:58	Detay
	121 227		



Membership pages will be usable after test.

19 °C KONAK

From: [✓ Point of Interest] Station Map
Best Western Konak Hotel

To: [✓ Point of Interest] Station Map
Bostanlı Bazaar

Departure Day: Tuesday

Departure Time: 13 : 00

Vehicles: ✓ bus ✓ subway ✓ train ✓ steamboat

i don't want to walk in transfering
[✓ max walking distance] 500

Sort Order: [✓ least transfer] fastest cheapest

(a) (b)

Figure 8: Sample screens from IJPS applications for (a) mobile web and (b) web.

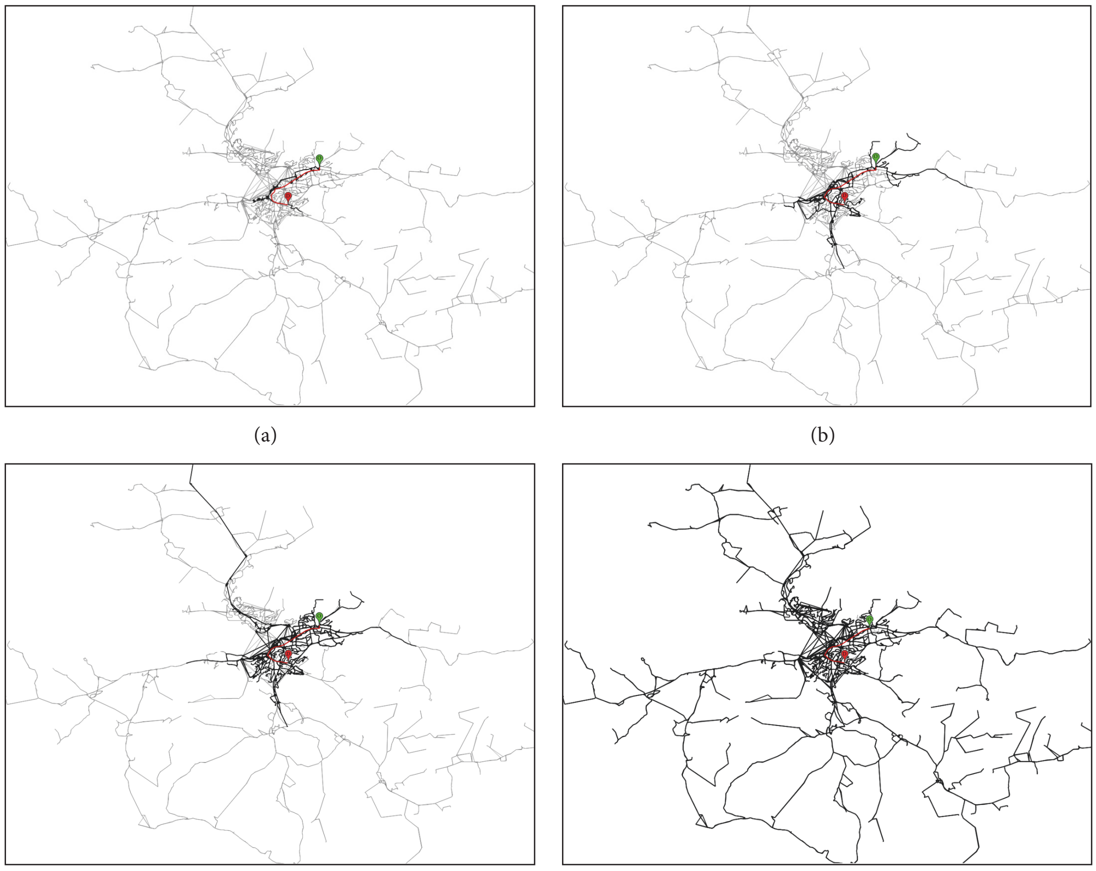

(a) (b) (c) (d)

Figure 9: Visualization of search spaces for Stages 2–5. The black lines indicate the traversed edges; the red lines show the optimal path.

Table 2: Average runtime (sec) and result count for stages of GPFA algorithm.

Stage	# of queries executed (%)	Avg. runtime (sec)	# of queries that generated result (%)	Avg. result count
1	96,107 (100.0)	0.00005	15,738 (16.4)	2.24
2	96,107 (100.0)	0.08743	47,319 (49.2)	20.46
3	66,500 (69.2)	0.37313	47,434 (71.3)	28.92
4	28,905 (30.1)	0.77419	27,392 (94.8)	949.27
5	777 (0.8)	8.47079	91 (11.7)	2.38
Overall		0.62618		295.00

several runs of modified Dijkstra's algorithms. Overall, GPFA produces accurate results for all queries with an acceptable response time for system users.

7. Conclusion

In this study, we designed and implemented the service-oriented, multimodel IJPS to assist travelers in journey planning. We proposed a gradual approach to path finding in transportation networks with the GPFA. Our proposed algorithm employed the modified versions of Dijkstra's algorithm in several stages. The modified Dijkstra's algorithms ran several times at each stage depending on the stop and line count. We have presented our results obtained by running the IJPS on the city of Izmir for journey planning. We assigned edge weights on the fly associated with the travel time function. The search space was reduced by decreasing the number of visited nodes and edges in modified Dijkstra's algorithms. Our results showed that the number of visited edges is increased over the upper stages of the algorithm. Increases in the number of visited edges also resulted in increasing the algorithm runtime. We observed that the average response time of GPFA was 0.63 sec.

IJPS is a flexible system that can be applied to all means of public transport. Prospective means of transport to be operated in Izmir can be integrated into the system, too. IJPS has the potential to be applied in any city. The performance of IJPS can be enhanced by combination with other multimodel speed-up techniques.

Notations

o, d:	Origin node, destination node
p:	Path: the obtained routes are kept as Path objects that store route details
sp:	SubPath: Path object has a *SubPath* list, which relates to each part of a route
s, t:	Stop or station, transfer center
S_O and S_D:	Origin stops and destination stops
l:	Transportation line (bus, ferry, subway, or train)
L_O:	Origin lines: outbound transitions of origin stops
L_D:	Destination lines: inbound transitions of destination stops
L_U:	Used-line list: the lines which form a direct path
L_{Pr}:	Predestination lines: the lines that intersect with a destination line at a stop directly or with a walk
C_T, C_W, and C_{Tr}:	Transition cost, walk cost, and transfer cost.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was performed in participation with Ünibel A.Ş. and was supported by the Scientific and Technological Research Council of Turkey under Grant 3110231.

References

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[2] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis, "Efficient models for timetable information in public transportation systems," *ACM Journal of Experimental Algorithmics*, vol. 12, pp. 2–4, 2008.

[3] H. Bast, E. Carlsson, A. Eigenwillig et al., "Fast routing in very large public transportation networks using transfer patterns," in *Proceedings of the Algorithms - ESA 2010, 18th Annual European Symposium. Proceedings, Part I*, pp. 290–301, Springer, Berlin, Germany, 2010.

[4] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transportation Science*, vol. 46, no. 3, pp. 388–404, 2012.

[5] H. Bast, S. Funke, and D. Matijevic, "Ultrafast shortest-path queries via transit nodes. In The Shortest Path Problem," in *DIMACS Book*, vol. 74, pp. 175–192, 2009.

[6] S. Witt, "Trip-based public transit routing," in *Algorithms–ESA*, vol. 9294 of *Lecture Notes in Computer Science*, pp. 1025–1036, Springer, Heidelberg, Germany, 2015.

[7] D. Delling, T. Pajor, and R. F. Werneck, "Round-based public transit routing," *Transportation Science*, vol. 49, no. 3, pp. 591–604, 2015.

[8] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, "Intriguingly simple and fast transit routing," in *Proceedings of the International Symposium on Experimental Algorithms*, vol. 7933 of *Lecture Notes in Computer Science*, pp. 43–54, Springer, Berlin, Germany, 2013.

[9] M. J. Atallah and M. Blanton, "VLSI layout algorithms," in *Algorithms and Theory of Computation Handbook*, Chapman and Hall/CRC, London, UK, Second edition, 2009.

[10] Y. Urayama and T. Tachibana, "Virtual network construction with K-shortest path algorithm and optimization problems for robust physical networks," *International Journal of Communication Systems*, 2015.

[11] T. Akiba, T. Hayashi, N. Nori, Y. Iwata, and Y. Yoshida, "Efficient Top-k Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling," in *Proceedings of the AAAI'15*, January 2015.

[12] J. Y. Yen, "Finding the *K* shortest loopless paths in a network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[13] J. Gao, J. Yu, H. Qiu, X. Jiang, T. Wang, and D. Yang, "Holistic Top-k simple shortest path join in graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 4, pp. 665–677, 2012.

[14] A. Frieder and L. Roditty, "An experimental study on approximating k shortest simple paths," *ACM Journal of Experimental Algorithmics*, vol. 19, no. 1, article 5, 2015.

[15] B. Y. Chen, Q. Li, and W. H. K. Lam, "Finding the k reliable shortest paths under travel time uncertainty," *Transportation Research Part B: Methodological*, vol. 94, pp. 189–203, 2016.

[16] H. Bast, D. Delling, A. Goldberg et al., "Route planning in transportation networks," in *Algorithm engineering*, vol. 9220 of *Lecture Notes in Computer Science*, pp. 19–80, Springer, Berlin, Germany, 2016.

[17] R. Geisberger, *Advanced Route Planning in Transportation Networks*, Karlsruhe Institute of Technology, Karlsruhe, Germany, 2011.

[18] F. Schulz, D. Wagner, and K. Weihe, "Dijkstra's algorithm online: an empirical case study from public railroad transport," *ACM Journal of Experimental Algorithmics*, vol. 5, no. 12, pp. 1–23, 2000.

[19] G. S. Brodal and R. Jacob, "Time-dependent networks as models to achieve fast exact time-table queries," *Electronic Notes in Theoretical Computer Science*, vol. 92, pp. 3–15, 2004.

[20] J. Dibbelt, T. Pajor, and D. Wagner, "User-constrained multimodal route planning," *ACM Journal of Experimental Algorithmics*, vol. 19, no. 3.2, 2015.

[21] G. V. Batz, R. Geisberger, P. Sanders, and C. Vetter, "Minimum time-dependent travel times with contraction hierarchies," *ACM Journal of Experimental Algorithmics*, vol. 18, no. 1.4, pp. 1–43, 2013.

[22] L. Antsfeld and T. Walsh, "Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm," in *Proceedings of the ITS World Congress*, October 2012.

[23] S. Witt, "Trip-based public transit routing using condensed search trees," https://arxiv.org/abs/1607.01299.

[24] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, "Connection scan algorithm," https://arxiv.org/abs/1703.05997.

[25] O. Dib, L. Moalic, M.-A. Manier, and A. Caminada, "An advanced GA–VNS combination for multicriteria route planning in public transit networks," *Expert Systems with Applications*, vol. 72, pp. 67–82, 2017.

[26] A. Antrim and S. J. Barbeau, “The many uses of GTFS data–Opening the door to transit and multimodal applications,” in *Proceedings of the ITS America'13*, April 2013.

[27] R. Clever, *Airport and station accessibility as a determinant of mode choice [Ph.D. thesis]*, University of California, Berkeley, USA, 2006.

[28] Y. Sun and M. Lang, “Modeling the multicommodity multimodal routing problem with schedule-based services and carbon dioxide emission costs,” *Mathematical Problems in Engineering*, vol. 2015, Article ID 406218, 21 pages, 2015.

[29] B. Ayar and H. Yaman, “An intermodal multicommodity routing problem with scheduled services,” *Computational optimization and applications*, vol. 53, no. 1, pp. 131–153, 2012.