City-level route planning with time-dependent networks

B. Anil Kumar[1], Rony Gracious[2], Chitrak Gangrade[2] and Lelitha Vanajakshi[2,*]

[1]Department of Civil and Environmental Engineering, IIT Patna, Bihta 801 103, India
[2]Department of Civil Engineering, Indian Institute of Technology Madras, Chennai 600 036, India

The computation of point-to-point shortest paths on time-dependent transportation networks has many practical applications. Finding the shortest path on transportation networks, taking into account prevailing dynamic traffic conditions, can help solve the problem of traffic congestion in urban areas. This study presents a framework for implementation of the shortest path algorithm on static as well as time-dependent city networks to identify the correct match between network complexity, computational requirements and scalability. Dijkstra, bidirectional A*, and A* with landmarks and triangle inequality (ALT) algorithms were selected and implemented based on their reported good performance in earlier studies. The algorithm implementation on both static and dynamic networks was tested on selected networks from Chennai city, India. Among the tested algorithms, ALT performed the best in terms of criteria used in this study. This algorithm is shown to be scalable and can be implemented for any other city network with ease, as demonstrated in this study. The study also discusses techniques for data extraction, cleaning and representation in addition to implementation and comparison of algorithms.

Keywords: Dynamic networks, shortest path algorithms, time-dependent city networks, transport planning, traffic engineering.

WITH the increasing number of vehicles on the roads, traffic congestion and associated delays have become a serious problem, especially in urban areas, all over the world. While online commercial solutions that can suggest best routes can help to an extent, they do not take into account time-dependent dynamic traffic conditions in a systematic way.

Shortest-path algorithms can be used to identify routes that cause least discomfort between origin and destination. There have been several studies on identifying the shortest path using static networks. However, a static transportation network does not exist, as traffic conditions keep changing on the roads. One way to address this is by considering traffic networks as time-dependent. Also, time-dependent road networks are usually represented as weighted graphs, where the weight of a link depends on travel time which changes continuously. Real-time data sourced through Global Positioning System (GPS), Bluetooth, image-processing through video cameras, etc. can be used to efficiently estimate point-to-point travel times for any two nodes in the network. With the advent of remote-sensing techniques such as GPS, collecting reliable real-time traffic data has become relatively easy. Using such real-time data, efficient time-dependent systems can be implemented to solve the problem of finding the best route for a trip dynamically.

This study is aimed at solving the shortest path problem in a city network. The methodology developed to implement shortest path algorithms can be applied to any traffic network, allowing it to be scalable and transferable. Different shortest path algorithms have been tested and compared in terms of their performance. The right mix of shortest path algorithms and speed-up techniques for real-time processing of a city-level network has been identified. The study also implements the selected algorithm on a real city-level network and proves the computational efficiency without compromising on accuracy. The main contribution of this study is identification and implementation of the correct combination of shortest path algorithm and speed-up technique for an actual city-level network under Indian traffic conditions. The study compares the performance of three algorithms, namely, A* with landmarks and triangle inequality (ALT) algorithm, bidirectional Dijkstra's algorithm and bidirectional A* algorithm. Comparison was made in terms of attributes such as ease of implementation, query times and accuracy of the solution. The performance of many popular algorithms depends on choice of the lower bound on the tentative distance between the node of interest and the destination. Many commercial car navigation systems use heuristic estimates such as layer concepts to speed-up processing times, which cannot guarantee reasonable solutions[1]. Such issues have been addressed in the present study by systematically identifying the lower bounds using ALT approach. This study also analysed the nature of relationship between complexity of the system and computational time, and showed that they are directly proportional and must be balanced to yield good results. The complexity of the system was measured in terms of the number of nodes touched while searching for a shortest

*For correspondence. (e-mail: lelitha@iitm.ac.in)

path result, i.e. it represents how an algorithm moves through the search space to obtain the desired result. Details of the proposed methodology are discussed below after a brief literature review of reported studies in this area.

Literature review

Studies on static networks

Shortest path algorithms have been extensively studied in the past, especially under static conditions. Dijkstra's Algorithm is a classic algorithm that maintains an array of the tentative distance to each node[2]. The algorithm visits the nodes in the order of their distance from the source node and is halted when the destination node is reached. These types of algorithms with a single source node are also called single-source problems. A simple improvement to Dijkstra's algorithm, known as the bidirectional search, executes the search simultaneously from the source and destination[3]. Dijkstra's algorithm cannot be used for large networks and is too slow for practical implementations. The A* algorithm is similar to the Dijkstra's search, except that the next node is selected based on the minimum tentative distance between the source and the destination[4]. The performance of the A* algorithm depends on the choice of the lower bounds on the tentative distance between the node and destination. In the bidirectional implementation, the route identified upon encountering a common node may not necessarily be the optimum one. Pohl[5] observed that with feasible lower bounds, the A* algorithm is synonymous with Dijkstra's algorithm, allowing the bidirectional implementation of A* and preserving optimality. These feasible lower bounds are calculated using reduced costs.

A commercially available solution for car navigation is based on heuristics and layer concepts. This heuristic search, however, does not guarantee reasonable solutions[1]. Various attempts at single-source problems have been made[6–9]. However, these fall short when applied on large datasets. Significant improvements have been reported using preprocessed data based on geometric information and hierarchical decomposition of the network. In addition, to expedite the process of finding the shortest path for a given origin and destination, speed-up techniques are used. Commonly used speed-up techniques include all pairs of shortest path (APSP)[10], reduced costs[5], hierarchical techniques[11–13], etc. Researchers have introduced the use of ALT algorithm, in which after selecting a small number of landmarks L, the distances between every node and L are pre-computed and stored[14–16]. These distances are used with the triangle inequality to produce better lower bounds for the A* algorithm. The ALT algorithm is reported to be highly effective and achieves significant speed-up. The advantage of using this algorithm is that it remains optimum even when the link weights increase and does not require pre-processing to be repeated for minor changes in the network.

Pre-computed cluster distances (PCD) partition the network into clusters and pre-compute the shortest distance between them[11]. While this approach exhibits good query performance, the APSP computation is an expensive step during pre-processing. Hierarchical algorithms reduce the nodes searched by dividing the underlying network into smaller segments and treating each segment as an individual node. Gutman[12] used an algorithm called reach-based routing, in which the minima of the distance of the node from the source and destination were used. The study also reported that a shortest path search can be pruned at nodes with a small reach. Due to inherent bidirectional nature of these algorithms, hierarchical algorithms can be applied only to static networks.

Highway hierarchies classify nodes and links based on their importance in the preprocessing step[13]. Highway hierarchies can work efficiently with large networks as they preserve optimality while the pre-processing can also be done efficiently. However, pre-processing of highway hierarchies is not trivial, and is not recommended for networks of moderate and large size. Transit node routing identifies a set of nodes that are always encountered when the shortest path is computed between two nodes that are sufficiently far apart[17]. Goldberg *et al.*[18] combined an advanced version of Reach algorithm[12] with landmark-based A* search[15]. SHARC (SHortcuts+ARC-flags) combined shortcuts and link flag methods to improve query and pre-processing times[19].

Many studies have reported faster way to find the shortest paths in large static networks with constant link weights. However, in the real-world scenario, the networks are rarely static and link weights need to be updated frequently. This motivates us to consider time-dependent network system and the related literature is presented below.

Studies on time-dependent networks

Traffic networks are dynamic in nature and hence it is more meaningful to use dynamic algorithms. Cooke and Halsey[20] developed a recursive formula to establish the minimum travel time to a given destination starting from a source at a particular time t. Dreyfus[21] attempted to generalize Dijkstra's algorithm for a time-dependent network[5]. The initial attempts relied on re-optimization techniques, and the effect of change in arc lengths on existing networks was studied[22,23]. Golden and Ball[24] developed a method to find the shortest path on a dynamic network with Euclidean distances. Dean[25] presented a label-correcting algorithm, similar to Dijkstra's algorithm[2], using cost functions instead of arc lengths. As this cost function was dependent on time, node labels could

be changed with time. Nannicini *et al.*[26] provided a heuristic implementation of the highway hierarchies on dynamic networks but the issues with this were that the pre-processing turned obsolete after a few changes in the network and re-computation of the pre-processed data was found to be computationally expensive. Contraction hierarchies are speed-up techniques for Dijkstra's algorithm, which use the concept of highway hierarchies to reduce the search space for bidirectional Dijsktra[27]. Batz *et al.*[28] have described an extension of the contraction hierarchies' algorithm in a time-dependent scenario. However, computational experiments were not conducted. Kim *et al.*[29] used three different algorithms (Dijkstra's algorithm with approximate buckets, Dijkstra's algorithm with double buckets and graph growth algorithm with two queues) to find the one-to-one time-dependent shortest path on real algorithms. Abdelghany *et al.*[30] introduced a parallel algorithm for the APSP problem with a network decomposition approach by decomposing the network into a set of independent augmented directed acyclic networks.

The A* algorithm can be applied effectively on a time-dependent network as long as the potential function is a valid lower bound on the distance between nodes at all times. While Euclidian distances are generally the first choice, Chabini and Lan[10] considered a network with arc lengths as the minimum cost of the arcs during a time period t and computed shortest paths from a node to all other nodes on this static network using them as lower bounds. Delling and Wagner[31] applied the ALT algorithm on a time-dependent network in a unidirectional manner. Nannicini *et al.*[32] provided a novel method to implement bidirectional ALT on a time-dependent network by initiating the backward search using lower bounds on arc costs to restrict the number of nodes to be explored by the forward search. Delling[33] advanced SHARC on the time-dependent scenario. It is one of the fastest algorithms for exact time-dependent shortest path computations on large networks.

Though this problem of time-dependent shortest path can be solved theoretically with the help of classical algorithms such as Djikstra's algorithm, such methods turn out to be computationally expensive for real-time field implementation. Also, the literature survey shows that there are multiple algorithms which can be implemented to solve the time-dependent shortest path problem. However, not many have implemented dynamic algorithms for time-dependent traffic networks using speed-up algorithms and pre-processing of data to make them computationally efficient for real-time implementation. The present study proposes a time-dependent algorithm, with application of pre-processing and speed-up techniques, to find the shortest path. The ALT algorithm, which can be implemented using real-time data, is used for this purpose. To summarize, this study focused on the specific case of solving the shortest path problem on a selected city network to identify the right mix of shortest path algorithm and speed-up technique for providing real-time routing information.

Pre-processing and cleaning of the raw network and representing it efficiently for optimum application of the shortest path algorithm was first carried out, as discussed below.

Data analysis

Network details

The road networks were sourced from OpenStreet Map (OSM)[34]. The extracted OSM raw data contained information about bus stops, traffic signals, etc. which are not relevant to this study. Hence, parser data were developed using imposm[35], a Python library that helps extract the required information from OSM data. The parser data were used to extract all the nodes, and store the node ID, and its latitude and longitude in a dictionary. The adjacent nodes were identified using the latitude and longitude, and the distance between them was computed using the Haversine formula[14].

Four test networks were extracted for the present study (Figure 1 *a–d*). Network shown in Figure 1 *a* is the entire Chennai city with all available routes, while the networks shown in Figure 1 *b–d* are sample sub-networks from city centre, which are in the order of decreasing size with increasing density of nodes (number of nodes per unit area). These were selected to understand the behaviour of the algorithm over long distances and over dense networks. Table 1 shows the properties of these test networks.

From Figure 1, it can be observed that the network contains unnecessary information which can be removed to improve the performance of the algorithm. For this, a data-cleaning operation was carried out as described below.

Data cleaning

The first technique (step 1) used for cleaning up the network identifies nodes with only two connected links and removes them from the network as explained below. Let A, B and C be three nodes connected in a series (Figure 2). As B is connected only to A and C, it is removed from the network, and A and C are connected with the distance between them as $d(A, B) + d(B, C)$. This is because the link characteristics cannot change without any other connections in between. However, it should be ensured that if A and C are also connected, the link with the minimum total distance is taken as the distance between A and C. This simple technique resulted in a reduction in the size of the network by about 60%. Table 2 shows the properties of the test networks after initial data cleaning, i.e. percentage reduction of network size (number of nodes in

Figure 1. Test networks: ***a***, Chennai city; ***b***, Chennai_1; ***c***, Chennai_2; ***d***, Chennai_3.

Table 1. Properties of the test networks

Network	No. of nodes	No. of links
Chennai city	184,208	199,803
Chennai_1	139,904	153,925
Chennai_2	99,282	110,324
Chennai_3	90,706	100,538

the network) after the first step of the data-cleaning process.

In the next step (step 2), nodes that are not accessible through roads and streets in the network are removed. These are present because OSM allows roads which are not part of the city road network also to be added to it. To clear them, an iterative search is run through the network, generating a tree of all the nodes connected to each other. The iterative search is continued until no more new nodes can be reached. The set of nodes that were touched during the search constitutes the new network. This methodology results in an additional network size reduction of about 10%. Table 3 shows the properties of the test networks after this step, i.e. cumulative reduction of size compared to the initial network (reduction percentage here is the decrease in percentage of nodes from the initial network size after both the steps).

From Tables 2 and 3 it can be seen that after removing non-essential information, the size of the network reduces significantly, which in turn improves the performance of

the shortest path algorithms on this network. As a sample case, Figure 3 *a* and *b* shows the complete Chennai network before and after data cleaning. The figure depicts the reduction in the number of nodes after the cleaning process, showing the efficacy of the process.

The networks once finalized were represented as adjacency array. The static network was extended to the dynamic representation by expanding the link weights across the time dimension. Different travel times were considered across a link at different times. As authentic real-time data were not available for time-dependent variations in travel time for the entire network, field data obtained for a corridor were extended to the entire network by assigning weight functions proportional to the link lengths, i.e. the present study considered only link length as the influencing factor for the network other than the corridor where data were available. We assume that the temporal influencing factors would be spread through the network weight. Dijkstra, bidirectional A* and ALT algorithms were selected and implemented based on their reported good performance in earlier studies. Here, we provide a brief description of these three methods along with modifications made and implementation details.

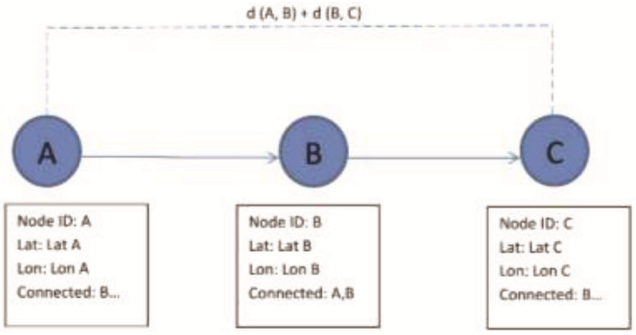

Figure 2. Node deletion based on the number of connected nodes.

Table 2. Properties of the test networks after initial data cleaning

Network	No. of nodes	No. of links	Percentage reduction in size
Chennai city	64,159	78,339	65.18
Chennai_1	55,857	68,701	60.08
Chennai_2	42,717	52,576	56.98
Chennai_3	38,801	47,447	57.23

Table 3. Properties of the test networks after step 2

Network	No. of nodes	No. of links	Percentage reduction in size
Chennai city	52,094	71,185	71.73
Chennai_1	45,713	62,657	67.33
Chennai_2	34,337	47,508	65.42
Chennai_3	30,677	42,535	66.18

Methodology

The present study compares the performance of ALT algorithm with a baseline approach, Dijkstra's algorithm and bidirectional A* algorithm. Though Dijkstra's algorithm has several limitations, it is one of the basic shortest path solutions and hence was used for comparison. Also, it is a known fact that Dijkstra's algorithm though time-consuming, guarantees optimal solution. These algorithms were tested across four networks as explained earlier. To begin with, the performance of the ALT algorithm was evaluated on a static network. The performance was measured across three parameters: time of computation (time taken), number of nodes reached before computing the shortest path (nodes touched) and length of the shortest path (distance). The ALT algorithm was used with four landmarks obtained from the farthest landmark selection method. The relationships among the three parameters are shown in Figure 4 *a–c* for the entire city network using ALT algorithm, where each data point corresponds to a different test case of the algorithm.

Figure 4 *a* shows the number of nodes the algorithm had to go through to get the shortest path against distance between the origin and destination and as expected, it is directly proportional. Figure 4 *b* shows the time taken to reach an optimal result versus distance between the origin and destination. As expected, since it has to go through more nodes to get the shortest path, time also increases with distance. Thus, the intuitive assumption that computation time increases with increase in distance between the nodes is verified. Figure 4 *c* shows the computation time versus nodes touched by the algorithm, which is shown to be directly proportional as expected. From Figure 4 *c*, it can be observed that there is a strong correlation between time of computation and the number of nodes the algorithm traverses before finding the shortest path. It can be concluded that the efficiency of the algorithm is proportional to the number of nodes it touches.

The performance of ALT algorithm was then compared with bidirectional Dijkstra's algorithm and bidirectional A* algorithm, the implementation of which is briefly explained below. These algorithms were tested across all four networks selected. The origin-destination (OD) pairs were selected randomly from the network. For all the algorithms, a sample set of 10,000 OD pairs was considered.

Dijkstra's algorithm

Starting from the source node *s* as root, Dijkstra's algorithm grows a shortest path tree that contains shortest path from *s* to all other nodes[2]. For the present study, binary heaps are used as priority queues for implementation of Dijsktra's algorithm[36]. A bidirectional version of Dijkstra's algorithm can be used to accelerate a shortest path query from a given node *s* to a given node *t*. For this,

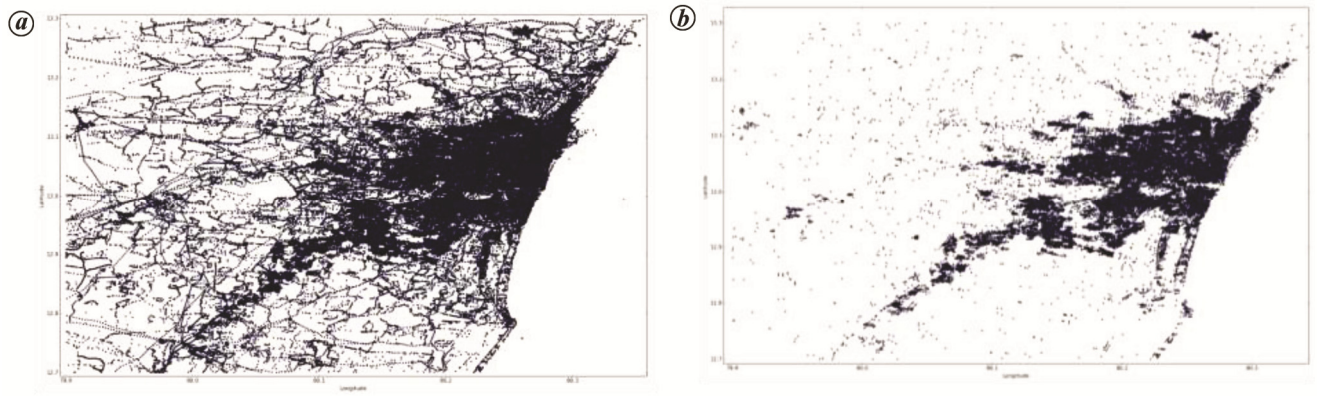

Figure 3. Representation of Chennai network (***a***) before and (***b***) after data cleaning.

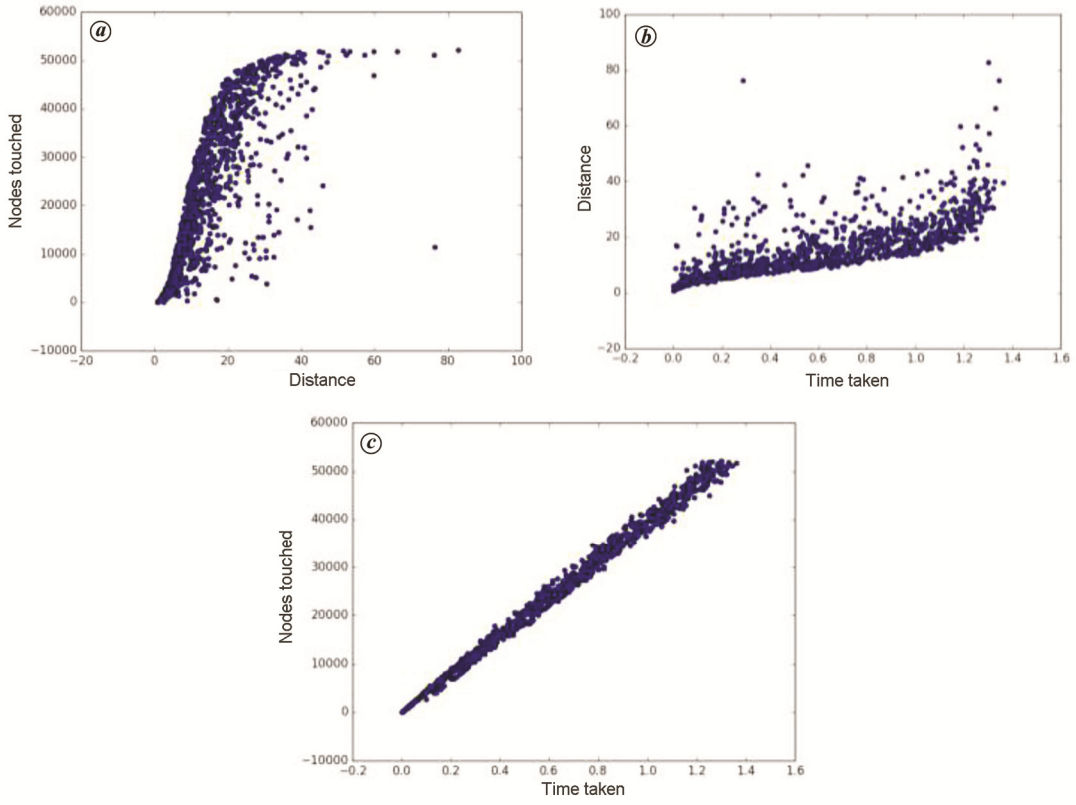

Figure 4. Relationship among three parameters for the ALT algorithm. ***a***, Distance versus nodes touched, ***b***, Time of computation versus distance, ***c***, Time of computation versus nodes touched.

two Dijkstra's searches are executed in parallel: one search from the source node s towards the destination node t, while the second search starts from the destination node t towards source node s. The bidirectional version significantly reduces the search space of the algorithm, halving it on an average. The algorithm can be extended to the time-dependent case on a first-in-first-out network by a simple modification in the arc relaxation procedure[37].

Let $l[u]$ be the distance label, i.e. the distance from the source to a node u. If t_0 is the departure time from the source node, for a link (u, v), we check if $l[v] > l[u] + d_{u,v}(t_0 + l[u])$. Here, $d_{ij}(t)$ is extracted from D, the set of time-dependent link travel times for nodes i, j. This bidirectional Dijkstra's algorithm was implemented for all the selected networks. Table 4 shows the performance of the algorithm.

From Table 4, it can be observed that the Chennai city network took an average computing time of 9.07 sec and a maximum of 74.24 sec with a standard deviation of 12.03 sec to identify a path. As evident from the table,

Table 4. Performance of bidirectional Dijkstra over four static test networks

Bidirectional Dijkstra		Chennai city	Chennai_1	Chennai_2	Chennai_3
Average performance	Time (s)	9.06	5.85	3.00	2.82
	Distance (km)	16.59	13.06	9.92	9.32
	Nodes touched	16,462	14,620	11,063	9,759
Standard deviation	Time (s)	12.03	7.47	3.70	3.43
Worst performance	Time (s)	74.24	41.21	20.34	19.07
	Distance (km)	60.49	27.23	19.16	19.24
	Nodes touched	44,316	38,622	26,926	24,308

Table 5. Performance of bidirectional A* with reduced costs over four static test networks

Bidirectional A* with reduced costs		Chennai city	Chennai_1	Chennai_2	Chennai_3
Average performance	Time (s)	2.83	2.38	1.37	0.83
	Distance (km)	16.84	13.25	10.04	9.46
	Nodes touched	3934	3501	2590	2238
Standard deviation	Time (s)	4.77	3.8429	2.20	1.21
Worst performance	Time (s)	42.81	35.25	24.59	13.37
	Distance (km)	101.20	40.90	44.60	20.90
	Nodes touched	18,324	16,082	13,573	10,567

query times for bidirectional Dijkstra are very high to be applied practically and hence the A* algorithm was used next.

A algorithm*

The efficiency of Dijkstra's algorithm can be improved by guiding the direction of the search towards the destination node. A* sorts the priority queue based on the function $F_u = L_u + \pi(u)$, where F_u is the sum of the (i) tentative distance between the source s and node u, i.e. L_u and (ii) the potential function, $\pi(u)$[38]. It can be seen that if the potential function is selected as $\pi(u) = 0$, F_u is the same as that of Dijkstra's algorithm and thus it is a special case of the A* algorithm. Furthermore, if $\pi(u) = d(u, t)$, the search only selects the nodes on the shortest path. Hence the efficiency of the algorithm depends on the accuracy with which the potential function can replicate $d(u, t)$.

In addition, a bidirectional search can be implemented to improve query times of A*. However, the identified solution when the forward search and backward search intersect, may not be the best possible shortest path. With the use of reduced costs, A* algorithm can be modelled as Dijkstra's algorithm, ensuring optimality in a bidirectional search while improving the performance of the algorithm. Bidirectional A* can be implemented with the use of reduced costs, translating the A* algorithm into Dijkstra's algorithm. The Dijkstra's algorithm using link length l' modified as in eq. (1) is equivalent to the A* algorithm using the original link length $d(u, v)$ and π as the potential function. This has been proven by Pohl[5].

$$l'(u, v) = d(u, v) + \frac{1}{2}(\pi_s(v) - \pi_s(u)) + \frac{1}{2}(\pi_l(u) - \pi_l(v)). \quad (1)$$

Table 5 presents the results obtained after implementation of the bidirectional A* using reduced costs and Euclidian distances as potential function. From the table it can be observed that bidirectional A* algorithm with reduced costs reduced the average computing time to 2.83 sec and a maximum of 42.82 sec with a standard deviation of 4.77 sec. Next, ALT algorithm was implemented as discussed below.

ALT algorithm

Very few speed-up techniques for route planning have been proven to work in a dynamic scenario. For most of these techniques, the pre-processed information must be updated every time the underlying graph is changed. However, goal-directed search based on landmarks (ALT algorithm) performs well as long as a link weight does not drop below its initial value[14–16].

One can solve the point to point problem only for a small portion of the graph. The ALT algorithm does this by pruning the search space of the A* algorithm with the use of landmarks. In bidirectional search, when the two searches intersect, it is not necessary that the intersecting node will lie on the shortest path. This problem can be solved by changing the termination criteria or using consistent potential functions. In this study, consistent approach was utilized for bidirectional ALT implementation.

The ALT algorithm obtains the potential function by computing the static shortest path distance of all the nodes from certain preselected landmarks in the network. Lower bounds are computed using these distances in combination with the triangle inequality[15]. Let L be a landmark and $d(L, u)$ be the distance of u from L. Then, by triangle inequality, $d(L, u) - d(L, v) \leq d(u, v)$. Thus,

the difference between the distances of the node from the landmark serves as a lower bound. For tighter lower bounds, one can take the maximum for all landmarks over these bounds. For time-dependent networks, for all $(i, j) \in A$, let $d_{ij}^{\min} = \min\{d_{ij}(t)\}$, i.e. based on historic data/ predicted link travel times, the lower bound can be taken as the minimum travel time on a link recorded over a time T. A virtual static network is created using $d_{ij}^{\min}$ as the link travel time. The lower bounds are then computed on this virtual network.

To illustrate the working of ALT algorithm, an example is detailed below. Figure 5 shows a sample network with nine nodes and 13 links. Here, S represents source, T represents terminal and E represents landmark. In Figure 5, values on links represent distance and values on the left side of the nodes represent pre-calculated distance from the landmark for each node. To start with, ALT algorithm calculates the potential function for any node as sum of distance from source and the absolute difference between distance from landmark and distance of destination from landmark. Table 6 shows the corresponding priority queue moves for the given network.

Here, proper landmark selection is important for quality for lower bounds. The preprocessing entails carefully choosing a small number of landmarks, then computing

Table 6. Priority queue moves for the sample network using ALT algorithm

Step	Queue
0	S0
1	A8, B10, C11
2	B10, C11, D17
3	C11, E16, D17
4	E16, D17, F19
5	G16, D17, F19
6	T16

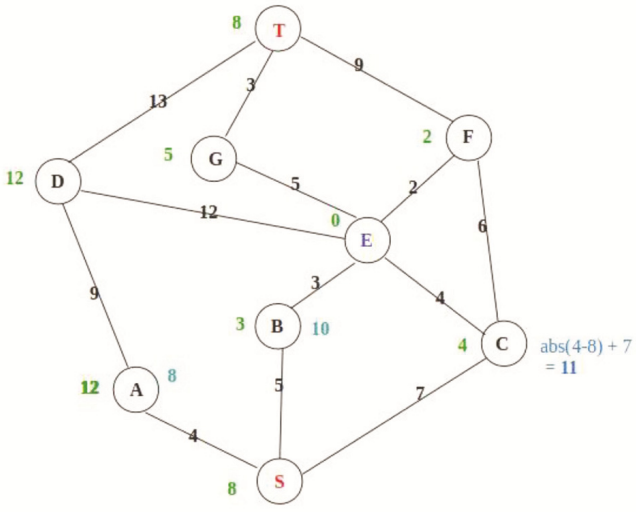

Figure 5. Working of ALT algorithm.

and storing shortest path distances between all vertices and each of these landmarks. The simplest way of choosing landmarks is to select landmark nodes at random. While this approach works reasonably well, better techniques can be devised. The farthest landmark selection approach is one such technique, which picks a vertex and finds the farthest vertex v_1 from it[15]. Then v_1 is added to the set of landmarks. This is repeated iteratively finding the next landmark farthest away from the rest of the selected landmarks. In this study, the farthest landmark selection strategy is used because of its small computation time.

For bidirectional implementation of ALT algorithm to solve time-dependent shortest path problem, backward search cannot be applied in a conventional sense as the arrival time is not known in advance. Hence, backward search is used to reduce the search space of the forward search. A backward search is run on a static network, weighted by the lower bounds computed using landmarks. Once the two searches intersect, the forward search runs only on the nodes explored by the backward search. Table 7 shows the performance of this algorithm for the four test networks.

From Table 7, it can be observed that the Chennai city network took an average computing time of 0.59 sec and a maximum of 1.43 sec with a standard deviation of 0.36 sec to identify a path on static networks, which is much faster and less varying than the previous two methods. In order to compare the performance of the above algorithms, a box plot of computation times for the static networks was made (Figure 6). From Figure 6, it can be observed that the unidirectional ALT algorithm has the least average and spread of computation time in all the four networks, whereas Dijkstra's algorithm has the highest average computing time and spread.

It is evident from Tables 4, 5 and 7, and Figure 6 that the ALT algorithm is far superior to the other algorithms tested. On an average, it can be seen that the ALT algorithm is twice as fast as the bidirectional A* algorithm and four times as fast as the bidirectional Dijkstra's algorithm. Another advantage of the ALT algorithm over the others is the low variability in performance metrics. The worst-case performance of ALT goes up to 1.45 sec (< three times the average), while for Dijkstra's it goes up to 12 times the average performance.

As ALT provides the best performance on the static network, analysis of performance of two variants of the ALT algorithm, unidirectional and bidirectional over four time-dependent networks was conducted. Both variants used four landmarks obtained from the farthest selection method. Table 8 shows the performance of these two algorithms for all the networks under consideration.

It can be seen that on the time-dependent network, the unidirectional ALT algorithm performs better than the bidirectional ALT algorithm. This is counterintuitive as the bidirectional algorithm is traditionally used to

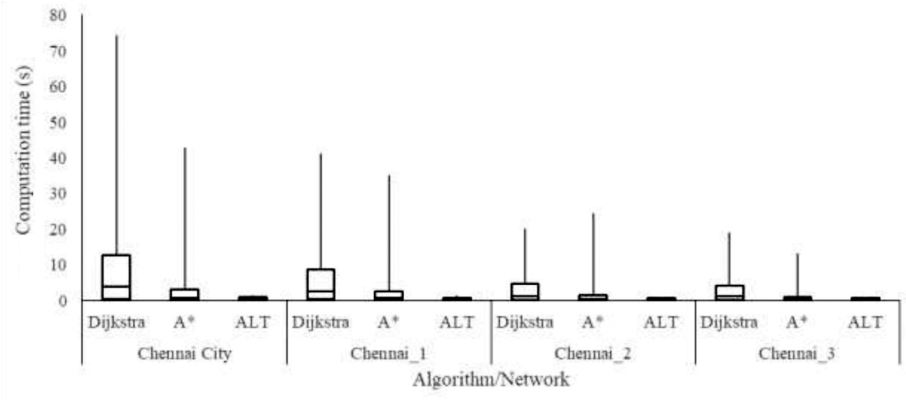

Figure 6. Box plots of computation time for three different algorithms over the four static study networks.

Table 7. Performance of unidirectional ALT algorithm over four static test networks

Unidirectional ALT algorithm		Chennai city	Chennai_1	Chennai_2	Chennai_3
Average performance	Time (s)	0.60	0.54	0.41	0.36
	Distance (km)	16.62	12.98	9.97	9.40
	Nodes touched	25,737	22,580	16,970	15,135
Standard deviation	Time (s)	0.37	0.32	0.24	0.21
Worst performance	Time (s)	1.43	1.42	0.94	0.86
	Distance (km)	31.66	20.44	15.12	19.45
	Nodes touched	50,287	43,592	34,071	30,238

Table 8. Performance of unidirectional and bidirectional ALT algorithms over four time-dependent test networks

Unidirectional ALT algorithm		Chennai city	Chennai_1	Chennai_2	Chennai_3
Average performance	Time (s)	0.34	0.31	0.24	0.21
	Distance (km)	33.89	26.87	21.62	20.90
	Nodes touched	24,495	21,267	16,292	14,430
Worst performance	Time (s)	1.20	1.10	0.85	0.74
	Distance (km)	31.00	22.38	18.29	35.17
	Nodes touched	51,411	43,042	32,531	30,292
Bidirectional ALT algorithm		Chennai city	Chennai_1	Chennai_2	Chennai_3
Average performance	Time (s)	0.73	0.61	0.45	0.38
	Distance (km)	19.78	13.54	10.04	9.92
	Nodes touched	29,842	24,210	17,293	15,533
Worst performance	Time (s)	1.38	1.45	0.95	0.86
	Distance (km)	30.43	21.32	17.32	20.58
	Nodes touched	48,793	44,930	35,247	31,367

improve computational efficiency. This anomaly may be due to the increasing network density, referring to the number of nodes per unit area. As the network under consideration is a city-level network, it is dense with thousands of nodes in a few square kilometres of area. As a result, the combined number of nodes touched by both the forward and backward searches is higher than the nodes touched by the unidirectional variant. The computational overhead caused by additional operations in the bidirectional search may be further adding to the computational complexity.

The drawback of using ALT is its high memory consumption for storing distances[39]. Once the landmarks are selected in ALT, the distance between landmarks and the nodes is computed, and then estimation of distances within the graph, i.e. between individual nodes, is carried out using triangle inequality. These two sets of distances have to be stored in order to find the shortest path in the network, leading to high memory consumption.

Summary

This study describes a technique to find the shortest path between two nodes on a large-scale time-dependent city-level network. It has identified the right mix of shortest

path algorithms and speed-up techniques for providing real-time routing information to users in a city. To reduce the size of the network, two techniques were implemented, which resulted in shrinking of the network by about 70%. The first technique reduced the size of the network by pruning the nodes that are connected to only two links. Such nodes were removed from the network resulting in a sparse intersection-to-intersection network. The second method involved removal of nodes that are not accessible from the main network. These nodes consisted of ways that are not part of the road network of the city and cannot be accessed by regular users.

The algorithms selected were first tested using a static network. The algorithms selected were bidirectional Dijkstra's algorithm, bidirectional A* algorithm, and ALT algorithm. A comparison showed better performance by the ALT approach than the other methods. Once the algorithm gave satisfactory results on the static network, it was extended to the time-dependent case. Two variations of the ALT algorithm, bidirectional and unidirectional, were tested. It was found that the unidirectional version was better suited for a city-level network than the bidirectional variant. This may be because city networks are dense and the bidirectional approach may not really improve on the search. The additional overhead for added operations also contributes to the computational pressure on the algorithm. On the other hand, the unidirectional variant was found to be space-efficient and showed an average query time of about 0.35 s.

The promising results show that the methodology proposed in this study can be directly used to implement real-time routing suggestions to end-users across any city. As the framework is flexible and does not depend on the underlying network, it can be scaled to other cities as well. This information can be relayed to the end-user through multiple channels ranging from SMS service to websites and mobile applications.

1. Lauther, U., An extremely fast, exact algorithm for finding shortest path in static networks with Geographical background. *Geoinformation und Mobilität–von der Forschung zur praktischen Anwendung* (eds Raubal, M., Sliwinski, A. and Kuhn, W.), 2004, vol. 22, pp. 219–230.
2. Dijkstra, E. W., A note on two problems in connection with graphs. *Num. Math.*, 1959, **1**(1), 269–271.
3. Luby, M. and Ragde, P., A bidirectional shortest-path algorithm with good average case behavior. *Lect. Notes Comput. Sci.*, 1985, **194**, 394–403.
4. Hart, P. E., Nilsson, N. J. and Raphael, B., A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybern.*, 1968, **4**(2), 100–107.
5. Pohl, I., Bi-directional search. *Mach. Intell.*, 1971, **6**, 124–140.
6. Cherkassky, B. V., Goldberg, A. V. and Radzik, T., Shortest path algorithms: theory and experimental evaluation. In Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM Press, Arlington, USA, 1971, pp. 516–525.
7. Gallo, G. and Pallottino, S., Shortest path algorithms. *Ann. Oper. Res.*, 1988, **13**, 3–79.
8. Goldberg, A. V., Shortest path algorithms: engineering aspects. *LNCS*, 2001, **2223**, 230–241.
9. Zhan, F. B. and Noon, C. E., Shortest path algorithms: an evaluation using real road networks. *Transp. Sci.*, 1998, **32**(1), 65–73.
10. Chabini, I. and Lan, S., Adaptations of the A* algorithm for the computation of fastest paths in deterministic discrete-time dynamic networks. *IEEE Trans. Intell. Transp. Syst.*, 2002, **3**(1), 60–74.
11. Maue, J., Sanders, P. and Matijevic, D., Goal directed shortest path queries using precomputed cluster distances. *LNCS*, 2006, **4007**, 316–327.
12. Gutman, R., Reach-based routing: a new approach to shortest path algorithms optimized for road networks. In Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX), Society for Industrial and Applied Mathematics, Philadelphia, USA, 2004, pp. 100–111.
13. Sanders, P. and Schultes, D., Highway hierarchies hasten exact shortest path queries. *LNCS*, 2005, **3669**, 568–579.
14. Goldberg, A. V. and Harrelson, C., Computing the shortest path: A* meets graph theory. Microsoft Research Technical Report, Vancouver, Canada, MSR-TR-2004-24, 2004.
15. Goldberg, A. V. and Harrelson, C., Computing the shortest path: A∗ meets graph theory. In Proceedings of 16th ACM-SIAM Symposium on Discrete Algorithms (SODA), Philadelphia, USA, 2005, pp. 156–165.
16. Goldberg, A. V. and Werneck, R. F., Computing point-to-point shortest paths from external memory. In Proceedings of the 7th workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, Philadelphia, USA, 2005, pp. 26–40.
17. Bast, H., Funke, S., Matijevic, D., Sanders, P. and Schultes, D., In transit to constant time shortest-path queries in road networks. In Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, Philadelphia, USA, 2007, pp. 46–59.
18. Goldberg, A. V., Kaplan, H. and Werneck, R. F., Better landmarks within reach. *Lect. Notes Comput. Sci.*, 2007, **4525**, 38–51.
19. Bauer, R. and Delling, D., SHARC: Fast and robust unidirectional routing. In Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, Philadelphia, USA, 2008, pp. 13–26.
20. Cooke, K. and Halsey, E., The shortest route through a network with time dependent inter-nodal transit times. *J. Math. Anal. Appl.*, 1966, **14**(3), 493–498.
21. Dreyfus, S., An appraisal of some shortest-path algorithms. *Oper. Res.*, 1969, **17**(3), 395–412.
22. Murchland, J. D., A fixed matrix method for all shortest distances in a directed graph and for the inverse problem, Ph D thesis, University of Karlsruhe, Germany, 1970.
23. Nepal, K. P., Park, D. and Choi, C. H., Upgrading arc median shortest path problem for an urban transportation network. *J. Transp. Eng.*, 2009, **135**(10), 783–790; doi:10.1061/(ASCE)0733-947X(2009)135:10(783)
24. Golden, B. L. and Ball, M., Shortest paths with Euclidean distances: an explanatory model. *Networks*, 1978, **8**, 297–314.
25. Dean, B. C., Continuous-time dynamic shortest path algorithms, Master's thesis, Massachusetts Institute of Technology, USA, 1999.
26. Nannicini, G., Baptiste, P., Barbier, G., Krob, D. and Liberti, L., Fast paths in large-scale dynamic road networks. *Comput. Optim. Appl.*, 2010, **45**(1), 143–158.
27. Geisberger, R., Sanders, P., Schultes, D. and Delling, D., Contraction hierarchies: faster and simpler hierarchical routing in road networks. *LNCS*, 2008, **5038**, 319–333.
28. Batz, V., Geisberger, R. and Sanders, P., Time dependent contraction hierarchies – basic algorithmic ideas, Technical report, ITI Sanders, Faculty of Informatics, Universitat Karlsruhe (TH), Germany, 2008; http://arxiv.org/pdf/0804.3947v1.pdf (accessed on 24 March 2017).

29. Kim, T., Haghani, A. and Kim, H., Algorithms for one-to-one time dependent shortest path on real networks. In 92nd Annual Meeting of the Transportation Research Board, Washington, DC, USA, 2004.
30. Abdelghany, K., Hashemi, H. and Alnawaiseh, A., Parallel all-pairs shortest path algorithm network decomposition approach. *Transp. Res. Rec.: J. Transp. Res. Board*, 2016, **2567**, 95–104.
31. Delling, D. and Wagner, D., Landmark-based routing in dynamic graphs. *LNCS*, 2007, **4525**, 52–65.
32. Nannicini, G., Delling, D., Schultes, D. and Liberti, L., Bidirectional A* search on time-dependent road networks. *Networks*, 2012, **59**(2), 240–251.
33. Delling, D., Time-dependent SHARC-routing. *Lect. Notes Comput. Sci.*, 2008, **5193**, 332–343.
34. OSM, The OpenStreet Map Project; https://www.openstreetmap.org/ (accessed on 24 March 2017).
35. Imposm.parser 1.0.7 documentation, Open Street Map XML/PBF parser for Python; http://imposm.org/docs/imposm.parser/latest/ (accessed on 24 March 2017).
36. Goldberg, A. V. and Tarjan, R. E., Expected performances of Dijkstra's shortest-paths algorithm, Technical Report 96-062, NEC Research Institute, Inc., Princeton University, Princeton, NJ, 1996.
37. Biswas, S. S., Alam, B. and Doja, M. N., Generalization of Dijkstra's algorithm for extraction of shortest paths in directed multigraphs. *J. Comput. Sci.*, 2013, **9**(3), 377–382.
38. Zeng, W., Church, R. L., Finding shortest paths on real road networks: the case for A*. *Int. J. Geogr. Inf. Sci*., 2009, **23**(4), 531–543.
39. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D. and Wagner, D., Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *LNCS*, 2008, **5038**, 303–318.

ACKNOWLEDGEMENTS. We thank the Ministry of Electronics and Information Technology, Government of India for supporting this research through the project 'InTranSE-II – Departure Time Planner using V2V and V2I Communication'.

Received 27 December 2018; revised accepted 10 June 2020

doi: 10.18520/cs/v119/i4/680-690