Continuously Monitoring Alternative Shortest Paths on Road Networks

Lingxiao Li‡, Muhammad Aamir Cheema‡, Mohammed Eunus Ali§, Hua Lu†, David Taniar‡
‡Faculty of Information Technology, Monash University, Australia
§Bangladesh University of Engineering and Technology, Bangladesh
†Department of People and Technology, Roskilde University, Denmark
‡{lingxiao.li, aamir.cheema, david.taniar}@monash.edu, §eunus@cse.buet.ac.bd, †luhua@ruc.dk

ABSTRACT

Modern navigation systems do not only provide shortest paths but also some alternative paths to provide more options to the users. This paper is the first to study the problem of continuously reporting alternative paths for a user traveling along a given path. Specifically, given a path P on which a user is traveling, we continuously report to the user k paths from the user's current location on P to the target t. We present several algorithms each improving on the previous based on non-trivial observations and novel optimisations. The proposed algorithms maintain and exploit the previously computed useful information to efficiently update the k alternative paths as the user moves. We provide space and time complexity analysis for each proposed algorithm. We conduct a comprehensive experimental study on large real-world road networks. The results demonstrate that the proposed algorithms are up to several orders of magnitude faster than the straightforward algorithms.

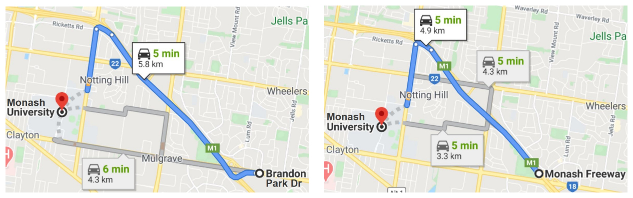
PVLDB Reference Format:
L. Li, M. A. Cheema, M. E. Ali, H. Lu, D. Taniar. Continuously Monitoring Alternative Shortest Paths on Road Networks. *PVLDB*, 13(11): 2243-2255, 2020.
DOI: https://doi.org/10.14778/3407790.3407822

1. INTRODUCTION

Given a source s and a target t, a shortest path query [28, 31, 3, 36, 7] returns the path from s to t with the minimum cost (e.g., distance, travel time). Shortest path queries are of fundamental importance to a wide variety of map-based systems. However, a shortest path may not always match a user's traveling choices and, therefore, modern map-based systems often provide several alternative paths from which the user can choose a path of their choice to travel along.

A *snapshot* alternative paths query returns top-k alternative paths (including the shortest path) from s to t. In contrast, for a user moving on a path P towards the target t, a *continuous* alternative paths query continuously reports top-k alternative paths from the user's current location s_i to t. A large body of research has focused on snapshot alternative paths queries [19, 5, 13, 6, 15, 14, 24]. However, to the best of our knowledge, we are the first to study continuous alternative paths queries.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407822



(a) (b)

Figure 1: Alternative paths to Monash University from the user's current location: (a) Brandon Park Drive; (b) Monash Freeway.

Motivation. A snapshot alternative paths query provides fixed choices at the start of a user's journey. In contrast, a continuous query goes one step further and continuously reports the alternative paths in real-time as the user travels towards the target. Note that, as the user moves, new interesting alternative paths may appear that were not reported to the user previously and thus giving new options in real-time. Consider the example in Fig. 1 where a user is traveling to Monash University. Fig. 1(a) shows three paths to Monash University when the user is at Brandon Park Drive. Assume that the user is travelling on the shortest path in terms of traveling time (the blue path that goes through M1 – Monash Freeway). As the user enters the Monash Freeway (see Fig. 1(b)), two new alternative paths can be found that have travel time similar to the user's current path. Note that these alternatives did not appear when the user was at Brandon Park Drive. Presenting these new paths gives the user more options to make a decision. Motivated by this, some commercial navigation systems (e.g., Google Maps) continuously report alternative paths to the users as they are moving towards their target.

In addition to enhancing navigation experience, continuously maintaining alternative paths in real time may also help autonomous vehicles, e.g., to reduce/avoid potential congestion [9] and to ensure flexible and safe driving. Limiting the alternative choices results in a constrained search space, forcing an autonomous vehicle to operate less flexibly. For example, if alternative paths are not updated continuously and timely, it may reduce an autonomous vehicle's maneuver time and thus affects its driving safety.

What are "good" alternative paths? A shortest path is well-defined as it corresponds to the path with the minimum cost. In contrast, a "good" alternative path is not well-defined. Intuitively, a set of k alternative paths is "good" if a user finds these paths to be *meaningful/natural* (e.g., a path does not entail unnecessary detours) and *significantly* different from each other. Since "goodness" of alternative paths is mostly subjective, existing studies for snapshot alternative paths use different approaches to define and compute top-k alternative paths using different intuitive ideas. However, due to the subjective nature of "goodness", there is no consensus on which of these techniques generates paths *perceived* better by users. Therefore, before we can propose techniques to continuously monitor alternative paths, we must choose one of the existing approaches to define the top-k alternative paths.

As to be detailed in Section 2.2, existing techniques for snapshot alternative paths fall into three broad categories: 1) plateaus-based techniques [19, 5, 20, 29, 17]; 2) penalty-based techniques [13, 6, 8, 21]; and 3) dissimilarity-based techniques [15, 14, 24]. Many commercial systems (e.g., Google Maps and TomTom) also generate alternative paths. However, these techniques are proprietary/confidential. Consequently, the research community and open-source community are unable to use these techniques in their implementation/systems. A recent user study [22] conducted on the Melbourne road network shows that the above mentioned techniques (including Google Maps) all return results with comparable path quality. Therefore, any of the existing techniques can be used to define top-k alternative paths. In this work, we choose to use the plateaus-based approach to generate alternative paths for the following reasons: 1) among the published techniques, the plateaus-based approach is arguably the most popular approach and has been used in several commercial and open-source systems (e.g., Cotares Limited[1] and GraphHopper[2,3]); 2) the paths generated using plateaus-based approach are guaranteed to be *local optimal* [5]; and 3) the computational cost to generate alternative paths using plateaus-based techniques is significantly smaller than the other techniques (e.g., generating dissimilarity-based alternative paths is NP-hard [14]).

Challenges and Contributions. As to be detailed in Section 2.2, the plateaus-based approach needs to create a forward shortest path tree T_f (rooted at the user's current location) and a backward shortest path tree T_b (rooted at target t), and to join the two trees. A straightforward solution to continuously report k alternative paths is to recompute T_f whenever the user moves. However, this is computationally expensive especially for large road networks. Also, it is non-trivial to efficiently update T_f instead of recomputing it. Even if the tree can be efficiently updated, joining T_f and T_b is not cheap since it requires traversing the two trees. Thus, there is a need to design techniques that can materialize the information computed earlier and can exploit this to efficiently update the k alternative paths without the need to update or join the two trees. We summarize our contributions below.

- To the best of our knowledge, we are the first to study the problem of continuously monitoring k alternative paths.
- We design novel techniques that exploit the fact that the backward tree T_b does not change as the user moves on the path. We categorise the vertices of T_b into familiar and unfamiliar vertices and use these to define *maxdepth* for each vertex which is an upper bound on the maximum length of a plateau. Based on careful observations, we present techniques to efficiently update *maxdepth* of the affected vertices. Finally, we present a novel idea to accessing the vertices in descending order of their *maxdepth* values to efficiently compute the results.
- We provide an extensive experimental study on several real-world road networks which demonstrates that our techniques are up to several orders of magnitude faster than the straightforward approaches. We also demonstrate the effectiveness of our proposed optimisations.

[1]`http://www.cotares.com/routing/`

[2]GraphHopper is an open-source routing library and has joined OpenStreetMap Foundation as a gold member.

[3]`https://www.graphhopper.com/blog/2016/01/19/alternative-roads-to-rome/`

2. PRELIMINARIES

2.1 Problem Definition

Similar to many existing works [2, 37, 1], we assume that the edges in the road network are bidirectional. However, our techniques can be trivially extended for directed networks. Let $G = (\mathcal{V}, \mathcal{E})$ be a graph representing the road network where $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ is the set of edges. Each edge $e = (u, v)$ in $\mathcal{E}$ is assigned a weight $w(e) > 0$, e.g., distance or travel time etc. Given two vertices s, $t \in \mathcal{V}$, a shortest path $sp(s, t)$ is a sequence of edges $(e_1, e_2, \ldots, e_n)$ that connects s to t such that $\sum_{i=1}^{n} w(e_i)$ is minimized. The shortest distance between s and t, denoted as $d(s, t)$, is the total weight of $sp(s,t)$. $d(s,u,t)$ denotes $d(s,u)+d(u,t)$ and $sp(s,u,t)$ denotes $sp(s,u) \oplus sp(u,t)$ where $\oplus$ is the concatenation operation. Table 1 lists the frequently used symbols throughout the paper.

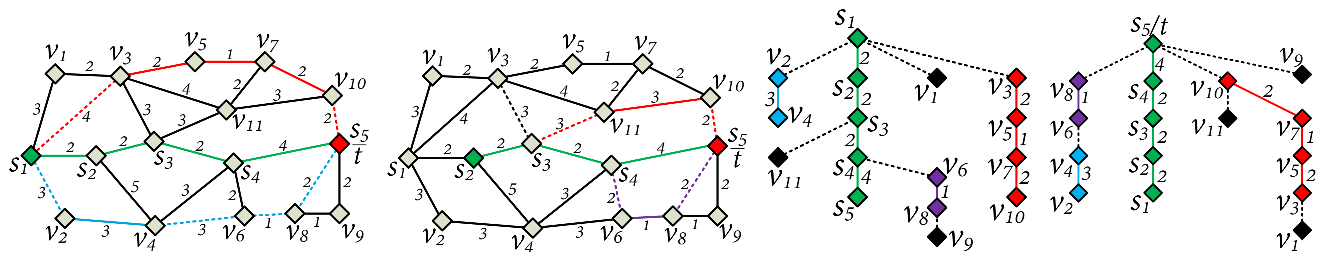
Table 1: Frequently used notations

Notation	Description
$e(u, v)$	an edge between vertices u and v
$w(u, v)$	weight of the edge $e(u, v)$
T_f	forward shortest path tree rooted at s
T_b	backward shortest path tree rooted at t
$pl(u, v)$	plateau with source and target ends u, v
$sp(u, v)$	the shortest path between s and t
$d^E(u, v)$	Euclidean distance between u and v
$d(u, v)$	shortest distance between s and t
$d(u, v, w)$	$d(u,v) + d(v,w)$
P	a user's traveling path $\langle s_1, s_2, s_3, ..., s_{\lvert p \rvert} \rangle$
$d^P(s_i, s_j)$	length of the segment $\langle s_i, \cdots, s_j \rangle$ of P
L^k	length of the k-th longest plateau
$dv_v^{s_i}$	deviation vertex of v when user is at s_i
$v.depth[u]$	depth of v w.r.t. u
$v.maxdepth$	maximum depth of a vertex v

2.1.1 Plateaus-based Alternate Paths

Given a source s and a target t, plateaus are generated by joining the forward shortest path tree T_f and the backward shortest path tree T_b where T_f is rooted at the source vertex s and T_b is rooted at the target vertex t. For simplicity of presentation, similar to many existing works [4, 10], we assume that the shortest path between any two vertices is unique implying that each shortest path tree is unique. The common branches[4] of these trees are called *plateaus*. We use $pl(u,v)$ to denote a plateau where v is closer to the target t (called the *target end*) and u is closer to s (called the *source end*). The length of a plateau $pl(u,v)$ is $d(u,v)$, the shortest distance between u and v. In Fig. 2(a), assume that source is s_1 and target is s_5. Fig. 2(c) and Fig. 2(d) show the forward and backward shortest path trees, respectively. The plateaus (common branches) are shown using same colors in

[4]Let u and v be two nodes such that v is an ancestor of u in a tree. We use branch to refer to a chain of edges connecting u and v.



(a) user is at s_1 (b) user is at s_2 (c) T_f with root s_1 (d) T_b with root s_5

Figure 2: A sample road network where user's path $P = \langle s_1, s_2, \cdots, s_5 \rangle$. Top-3 plateaus when user is at s_1 are $pl(s_1, s_5)$, $pl(v_3, v_{10})$ and $pl(v_2, v_4)$.

both trees except the black vertices (that represent common branches/plateaus with length 0, i.e., no edge containing such a vertex is common in the two trees). For example $\langle v_3, v_5, v_7, v_{10} \rangle$ is a plateau (represented as $pl(v_3, v_{10})$) with length 5 (shown in red). v_3 is the source end and v_{10} is the target end of this plateau. Other plateaus are $pl(s_1, s_5)$ with length 10 (shown green), $pl(v_2, v_4)$ with length 3 (shown blue) and $pl(v_6, v_8)$ with length 1 (shown purple).

Given a plateau $pl(u, v)$, the alternative path from s to t based on $pl(u, v)$ is $sp(s, u) \oplus pl(u, v) \oplus sp(v, t)$. In Fig. 2(a), the alternative path using $pl(v_3, v_{10})$ is $\langle s_1, v_3, v_5, v_7, v_{10}, t \rangle$. Note that $pl(u, v)$ represents the shortest path from u to v, i.e., $sp(u, v)$. Next, we briefly describe some properties of the plateaus. Proofs are straightforward and omitted.

Property 1. *A plateau $pl(u, v)$ is the overlap of $sp(s, v)$ and $sp(u, t)$.*

Property 2. *Let u be a descendent of v in the backward tree T_b. Vertices u and v are on the same plateau iff $d(s, u) + d(u, v) = d(s, v)$.*

Property 3. *For any plateau $pl(u, v)$, we have $d(s, u) + d(u, t) = d(s, v) + d(v, t)$, i.e., $d(s, u, t) = d(s, v, t)$.*

Property 4. *Two plateaus cannot intersect each other (because each shortest path tree is unique).*

In Fig. 2(a) where source is s_1 and target is s_5, $pl(v_3, v_{10})$ is the overlap of $sp(s_1, v_{10})$ and $sp(v_3, s_5)$ (Property 1). Also, $d(s_1, v_3) + d(v_3, v_{10}) = d(s_1, v_{10})$ (Property 2) and $d(s_1, v_3) + d(v_3, s_5) = d(s_1, v_{10}) + d(v_{10}, s_5)$ (Property 3).

2.1.2 *Continuous Alternative Paths*

Let $P = \langle s_1, s_2, \cdots, s_{|P|} \rangle$ be a simple path on which a user is travelling where $s_{|P|}$ is the target vertex, i.e., $t = s_{|P|}$. Hereafter, we use $s_{|P|}$ and t interchangeably. We use $P(s_i, s_j)$ to denote the user's path from s_i to s_j for $i < j \leq |P|$, i.e., $P(s_i, s_j) = \langle s_i, \cdots, s_j \rangle$. Furthermore, we use $d(s_i, s_j)$ to denote the shortest distance between s_i and s_j and $d^P(s_i, s_j)$ as the path distance from s_i to s_j as the user travels along the path P, i.e., $d^P(s_i, s_j) = \sum_{k=i}^{j-1} w(s_k, s_{k+1})$. In this paper, we continuously report k alternative paths for each location of the user $s_i \in P$. In real world applications, the users may not want an alternative path which is significantly larger than the path length from s_i to the target $s_{|P|}$. This can be controlled using a user defined parameter $\epsilon \geq 1$ such that any path reported to the user must have length at most equal to $d^P(s_i, s_{|P|}) \times \epsilon$. To formalize this, we define *valid* plateaus.

Definition 1. **Valid plateau**. Let $P = \langle s_1, \cdots, s_{|P|} \rangle$ be the path a user is traveling on and s_i be the user's current location. Given a user-defined parameter $\epsilon \geq 1$, a plateau $pl(u, v)$ is valid if the alternative path generated by it has length at most $d^P(s_i, t) \times \epsilon$, i.e., $d(s_i, v, t) \leq d^P(s_i, t) \times \epsilon$.

Since each plateau is unique and generates a unique alternative path, for simplicity, we use plateau to refer to an alternative path whenever clear by context. Our techniques report k longest plateaus (top-k plateaus) at each location s_i and these plateaus can be used to easily generate k alternative paths.

Definition 2. **Top-k Plateaus (TKP) query**. Given the current location s_i of a user on the path P, an integer k and a distance upper bound parameter $\epsilon \geq 1$, the TKP query $Q = (s_i, P, k, \epsilon)$ returns k longest valid plateaus w.r.t. s_i and t.

Definition 3. **Continuous Top-k Plateaus (CTKP) query**. Given a positive integer k, a distance upper bound parameter $\epsilon \geq 1$ and the user's path $P = \langle s_1, s_2, s_3, ..., s_{|p|} \rangle$, the CTKP query $Q = (k, \epsilon, P)$ query continuously returns the top-k valid plateaus R_i for each vertex s_i on P.

Whenever clear by context, hereafter, we use plateau to refer to a valid plateau. Consider the example in Fig. 2(a). Suppose that $\epsilon = 1.3$ and the user is traveling on the shortest path from s_1 to s_5/t (shown green). When the user is at s_1, the top-3 plateaus are $pl(s_1, s_5)$, $pl(v_3, v_{10})$ and $pl(v_2, v_4)$ with lengths 10, 5 and 3, generating three paths shown in green, red and blue lines in Fig. 2(a), respectively. When the user is at s_2, the top-3 plateaus can be obtained by joining the forward shortest path tree T_f rooted at s_2 (see Fig. 3(a)) and T_b (Fig. 2(d)). The top-3 valid plateaus at s_2 are $pl(s_2, s_5)$, $pl(v_{11}, v_{10})$ and $pl(v_6, v_8)$ with lengths 8, 3 and 1, respectively, generating three alternative paths shown in green, red and purple lines in Fig. 2(b). Note that $pl(v_3, v_5)$ is also a plateau with length 2. However, since $\epsilon = 1.3$, the plateau $pl(v_3, v_5)$ is invalid because the path generated by it $\langle s_2, s_3, v_3, v_5, v_7, v_{10}, s_5 \rangle$ has length $12 > (d^P(s_2, t) \times \epsilon = 8 \times 1.3 = 10.4)$. Thus, this plateau (and consequently the path generated by it) is ignored.

Some existing studies aim to further improve the quality of reported alternative paths by employing additional filtering/ranking criteria (e.g., uniformly bounded stretch [5]). To this end, our techniques can be immediately used to efficiently monitor top-m ($m > k$) plateaus and the paths generated using these m plateaus can be refined to obtain the top-k alternative paths satisfying the additional criteria.

2.2 Related Work

While various other types of continuous queries have received significant attention [12, 11, 26, 30], to the best of our knowledge, we are the first to study continuous top-k alternative paths queries. For the snapshot alternative paths, other than the plateaus-based techniques, there are two broad categories of the existing techniques. **Penalty-based techniques** [6, 13] iteratively compute shortest paths and, after each iteration, apply a penalty to each edge on the shortest path found in the previous iteration by increasing its edge weight. Further constraints can be applied to prune the retrieved shortest paths if they do not meet a user's requirement (e.g., are longer than a given threshold). The algorithm terminates when k paths satisfying the constraints are found. While the k paths returned by the penalty-based method may be quite similar to each other, the plateaus-based paths are naturally dissimilar to each other (as explained in [22]). **Dissimilarity-based techniques** [15, 14, 24] specifically aim to select dissimilar paths by defining a function to measure the dissimilarity between a path p and a given set of paths P. The goal is to iteratively add paths p in ascending order of their lengths to the current result set P if the dissimilarity between p and P is greater than a threshold. The problem is shown to be NP-hard [14] and the existing studies mainly focus on approximate solutions. A recent user study [22] discusses more details of the existing studies on computing snapshot alternative paths and provides a user study comparing the path quality of these approaches.

3. TECHNIQUES

We first briefly describe a straightforward algorithm based on tree joins (TJ) (Section 3.1). Then, we propose an algorithm based on a top down (TD) traversal of the backward tree (Section 3.2). Finally, built on two novel optimisations, we present our main algorithms named: depth-aware top down (DA-TD) algorithm (Section 3.3) and depth-aware best first (DA-BF) algorithm (Section 3.4). Initially (when user is at s_1), all algorithms compute the top-k plateaus by joining T_f and T_b. We focus on how each algorithm updates the results for each subsequent location s_i of the user.

3.1 Tree-Join Algorithm

For each location of the user s_i on the path P, the top-k plateaus can be obtained by joining the forward shortest path tree T_f rooted at s_i and the backward shortest path tree T_b rooted at the target $s_{|P|} = t$. Tree-Join algorithm exploits the fact that the T_b does not change because the target vertex remains the same. Therefore, for each location s_i, the algorithm creates T_f and joins it with T_b to obtain the top-k plateaus. The tree T_f is created using Dijkstra's algorithm [16]. The two trees T_f and T_b can be joined in time linear to the size of the two trees using backward pointers [19]. Next, we present a pruning rule to prune the subtree rooted at v.

Pruning Rule 1. *Let s_i be the user's current location and v be a node in the backward tree T_b such that $d(s_i, v) + d(v, t) > d^P(s_i, t) \times \epsilon$. The subtree rooted at v cannot have any valid plateau for the user's location s_i.*

Proof. Let u be *any* descendant node of v in T_b. We show that u cannot be on any valid plateau by showing $d(s_i, u) + d(u, t) > d^P(s_i, t) \times \epsilon$. Due to the triangle inequality, $d(s_i, u) \geq d(s_i, v) - d(u, v)$. Also, since u is a descendant of v in T_b with root t, we have $d(u, t) = d(u, v) + d(v, t)$. Thus, $d(s_i, u) + d(u, t) \geq d(s_i, v) - d(u, v) + d(u, v) + d(v, t)$. Hence, $d(s_i, u) + d(u, t) \geq d(s_i, v) + d(v, t) > d^P(s_i, t) \times \epsilon$. $\square$

Consider our running example in Fig. 2 where $\epsilon = 1.3$. Fig. 3 shows T_f and T_b when the user is at s_2 and $d^P(s_2, t) = 8$. The subtree rooted at v_7 in Fig. 3(b) can be pruned because $d(s_2, v_7) + d(v_7, t) = 7 + 4 = 11$ is greater than $d^P(s_2, t) \times \epsilon = 8 \times 1.3 = 10.4$.

During the construction of T_f, Dijkstra's algorithm prunes a vertex v that satisfies this condition. Note that $d(s_i, v)$ is known to the Dijkstra's algorithm and $d(v, t)$ can be obtained in $O(1)$ using the backward tree T_b.

Complexity analysis: We use E to denote the number of edges in the graph. As is typically the case, we assume that the road network is a connected graph, i.e., the space taken by input graph is $O(E)$. Thus, the space complexity is $O(E)$ because the size of each tree is bounded by $O(E)$. The construction of the shortest path tree by Dijkstra'a algorithm is bounded by $O(E \log V)$ where V is the number of vertices in the tree. The two trees can be joined in time linear to the number of vertices in the two trees [19]. Thus, the total time complexity is $O(E \log V)$.

3.2 Top Down Tree Traversal Algorithm

One possible approach to improve the performance of the tree-join algorithm is to design techniques to efficiently update T_f as the user moves, instead of recomputing T_f. Firstly, this is non-trivial and, to the best of our knowledge, there does not exist any technique that can efficiently update a shortest path tree when the source node is changed. Furthermore, even if a technique is designed to efficiently update T_f, this tree must still be joined with T_b to obtain the plateaus, which is inefficient. Therefore, instead of trying to efficiently update or partially update T_f, we look at the problem from a fresh perspective and propose an algorithm that traverses the backward tree T_b in a top down fashion to compute the top-k plateaus. A key strength of this tree traversal algorithm is that we can exploit the computed information for the user's future locations.

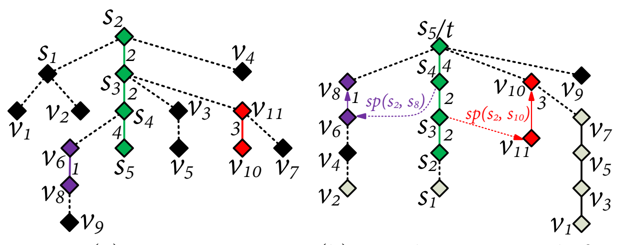
The key idea is to traverse the backward tree T_b in a top down fashion. For each accessed vertex v, the algorithm finds the plateau $pl(u, v)$ for which v is the target end. Recall that a plateau $pl(u, v)$ is the overlap of $sp(s_i, v)$ and $sp(u, t)$ (Property 1). Therefore, to find the plateau with the target end v, we get $sp(s_i, v)$ by issuing a shortest path query from s_i to v. Then, we check the overlap of $sp(s_i, v)$ with the subtree rooted at v in T_b to obtain the plateau. We illustrate this using an example.

Consider the example in Fig. 3(b) that shows the backward tree T_b for our running example where the user's current location is s_2. Suppose that the top-down traversal has accessed the vertex v_8 and we want to find the plateau with v_8 as the target end, i.e., $pl(u, v_8)$. The algorithm issues a shortest path query from s_2 to v_8 which returns $sp(s_2, v_8) = \langle s_2, s_3, s_4, v_6, v_8 \rangle$. The overlap of this path with the subtree of v_8 is $\langle v_6, v_8 \rangle$ which is a plateau with length 1 (the purple vertices in Fig. 3).

To efficiently obtain the plateau (i.e., overlap of $sp(s_i, v)$ and the subtree rooted at v), we simply need to find the first vertex u on $sp(s_i, v)$ such that v is the ancestor of u in T_b. At the first timestamp (i.e., when user is at s_1), we use the interval-based labelling scheme [32, 33] to label all vertices in T_b, which allows checking ancestor-descendent relationship for any two vertices in $O(1)$. Since T_b does not change, the labelling remains valid for all future locations.

3.2.1 Pruning Rules

Recall that Pruning Rule 1 prunes the subtree rooted at v for the current location s_i. The next pruning rule shows that the subtree can, in fact, be pruned for all future locations of the user on the path P.



(a) T_f at s_2 (b) Top down traversal of T_b

Figure 3: Illustration of the tree-join and top-down algorithms with user at s_2. Grey vertices are pruned. Vertices with the same color form a plateau.

Pruning Rule 2. *Let v be a node in the backward tree T_b such that $d(s_i, v) + d(v, t) > d^P(s_i, t) \times \epsilon$. The subtree rooted at v cannot have any valid plateau for any s_j on P where $j \geq i$.*

Proof. Let u be *any* descendant node of v in T_b. The Pruning Rule 1 shows that $d(s_i, u) + d(u, t) > d^P(s_i, t) \times \epsilon$. We now show that for any s_j on P s.t. $j \geq i$, $d(s_j, u) + d(u, t) > d^P(s_j, t) \times \epsilon$, i.e., u cannot be on any valid plateau for s_j. Let $d^P(s_i, s_j)$ be the distance between s_i and s_j along the path P. Note that $d^P(s_i, t) = d^P(s_i, s_j) + d^P(s_j, t)$. Thus $d(s_i, u) + d(u, t) > d^P(s_i, t) \times \epsilon$ can be written as $d(s_i, u) + d(u, t) > \epsilon \times (d^P(s_i, s_j) + d^P(s_j, t))$. Subtracting $d^P(s_i, s_j)$ on both sides gives us $d(s_i, u) - d^P(s_i, s_j) + d(u, t) > (\epsilon - 1) \times d^P(s_i, s_j) + \epsilon \times d^P(s_j, t)$. Since $d^P(s_i, s_j) \geq d(s_i, s_j)$, due to the triangle inequality, we have $d(s_j, u) \geq d(s_i, u) - d(s_i, s_j) \geq d(s_i, u) - d^P(s_i, s_j)$. Thus, we have $d(s_j, u) + d(u, t) > (\epsilon - 1) \times d^P(s_i, s_j) + \epsilon \times d^P(s_j, t)$. Since $\epsilon \geq 1$, the right side of the above inequality is at least $d^P(s_j, t) \times \epsilon$. Thus, we have $d(s_j, u) + d(u, t) > \epsilon \times d^P(s_j, t)$. □

At the beginning (i.e., at s_1) when the backward tree T_b is computed, we store $d(v, t)$ for each vertex v in T_b. Thus, $d(v, t)$ can be obtained in $O(1)$. However, $d(s_i, v)$ may not be known when the user moves to s_i. Instead, we use a lower bound (e.g., Euclidean distance $d^E(s_i, v)$) to prune the subtree as stated in the next pruning rule.

Pruning Rule 3. *Let v be a node in T_b where $d^E(s_i, v) + d(v, t) > d^P(s_i, t) \times \epsilon$. The subtree of v cannot have any valid plateau for every s_j on P where $j \geq i$.*

The proof is straightforward and is omitted. The next pruning rule prunes the subtree of v by using the length of k-th longest plateau found so far by the algorithm.

Pruning Rule 4. *Let s_i be a user's current location and L^k be the length of the k-th longest plateau found so far by the algorithm. The subtree rooted at v cannot contain any top-k plateau for s_i if $L^k + d(v, t) > d^P(s_i, t) \times \epsilon$.*

Proof. Assume that a plateau $pl(u, x)$ is longer than L^k where both u and x are in the subtree rooted at v. We show that the plateau is invalid, i.e., $d(s, x) + d(x, t) > d^P(s_i, t) \times \epsilon$. Since x is a node in the subtree of v in the backward tree T_b rooted at t, we have $d(x, t) \geq d(v, t)$. So, we have $d(s, x) + d(x, t) \geq d(s, x) + d(v, t)$. Also, if $pl(u, x)$ is longer than L^k, we have $d(s, x) > L^k$ (because $pl(u, x) \subseteq sp(s, x)$ according to Property 1). Thus, we have $d(s, x) + d(x, t) > L^k + d(v, t) > d^P(s_i, t) \times \epsilon$. □

Let *maxleafdist*(v) be the distance from a vertex v to the furthest leaf in the subtree rooted at v. The next pruning rule prunes the subtree using *maxleafdist*(v).

Algorithm 1: Top Down Algorithm

```
   Input: $s_i$: user's current location on $P$
          $T_b$: backward shortest path tree
   Output: $R_i$: top-$k$ plateaus for $s_i$
1  use plateaus from $s_{i-1}$ to initialize $R_i$ and $L^k$;
2  initialize a queue $S$ with root of $T_b$;
3  while $S$ is not empty do
4      pop a vertex $v$ from $S$;
5      if $d^E(s_i, v) + d(v, t) > d^P(s_i, t) \times \epsilon$ then
6          remove the subtree rooted at $v$ from $T_b$;
7          continue;
8      if $v$ is not pruned then // Pruning Rules 4&5
9          insert children of $v$ in $S$;
10         if $v$ is not marked visited then
11             obtain $pl(u, v)$ using $sp(s_i, v)$;
12             update top-$k$ plateaus and $L^k$ if needed;
13             mark vertices on $pl(u, v)$ as visited;
14 return $R_i$
```

Pruning Rule 5. *The subtree rooted at v cannot contain a top-k plateau if maxleafdist$(v) < L^k$.*

Proof. Recall that each plateau $pl(u, v)$ is the overlap of $sp(s_i, v)$ and the subtree rooted at v. Hence, the length of any plateau $pl(u, v)$ is bounded by *maxleafdist*(v). Thus, the length of $pl(u, v)$ is smaller than L^k. □

3.2.2 Algorithm

Algorithm 1 details how to update the top-k plateaus when a user moves from s_{i-1} to s_i. First, we use the top-k plateaus for s_{i-1} and check which of these are still plateaus for the new location s_i. Specifically, u and v are on the same plateau iff $d(s_i, u) + d(u, t) = d(s_i, v) + d(v, t)$ (Property 2). Hence, we issue two shortest path queries from s_i to both u and v to check if $pl(u, v)$ is still a plateau or not. Then, we initialize the set of top-k plateaus R_i by inserting the valid plateaus in R_i and update L^k accordingly (set to 0 if R_i contains less than k plateaus). Note that we cannot terminate the algorithm even if all k plateaus from s_{i-1} are still the plateaus for s_i. This is because the length of any of these plateaus or some other plateaus may have increased thus the top-k plateaus may have changed.

The algorithm accesses the tree in a top-down fashion by iteratively accessing nodes from a queue S. We apply the pruning rules presented in Section 3.2.1. Specifically, we remove the whole subtree of v from T_b if $d^E(s_i, v) + d(v, t) > d^P(s_i, t) \times \epsilon$ (line 5) because the subtree cannot contain any valid plateau for all future locations of the user. If v cannot be pruned using the pruning rules, we insert its children in S to be accessed in future iterations (lines 8 and 9). We issue a shortest path query from s_i to v to obtain the plateau $pl(u, v)$ and update top-k plateaus and L^k as needed (lines 11 and 12). In our implementation, we use pruned highway labeling (PHL) [7] to obtain the shortest paths. Each vertex v' on $pl(u, v)$ is marked visited (line 13). In each iteration, a vertex which is already visited is ignored (line 10). This ensures that the algorithm does not issue an unnecessary shortest path query to obtain a plateau $pl(u, v')$ which is contained by a $pl(u, v)$ already found. The algorithm terminates when the queue S becomes empty.

Fig. 3 shows execution of the top-down algorithm when the user is at s_2. The algorithm first issues a shortest path query to s_5 and obtains the plateau $pl(s_2, s_5)$ with length 8. When v_8 and v_{10} are accessed, two plateaus $pl(v_6, v_8)$ and $pl(v_{11}, v_{10})$ are found with length 1 and 3, respectively.

The vertices on these plateaus are marked visited and not processed by the algorithm. Plateaus for v_4 and v_9 contain only single nodes, i.e., $pl(v_4, v_4)$ and $pl(v_9, v_9)$ have length 0. The grey vertices are pruned by the pruning rules. Hence, the algorithm returns $pl(s_2, s_5)$ and $pl(v_{11}, v_{10})$ as the top-2 plateaus (assuming $k = 2$).

Complexity Analysis: The space complexity is $O(E + SP_{index})$ where $O(E)$ is the space for the input graph and SP_{index} is the size of the shortest path index (depending on which shortest path algorithm is used). Next, we analyse the time complexity. Let V_u^{TD} be the total number of unpruned and umarked vertices by the Top-Down (TD) algorithm and SP_{cost} be the cost of a single shortest path query. Applying each pruning rule takes $O(1)$. Since the algorithm uses a queue, push and pop operations also take $O(1)$. The algorithm processes $O(V)$ vertices in total and issues V_u^{TD} shortest path queries. Therefore, the total cost is $O(V + V_u^{TD} \times (SP_{cost} + \log k))$ where $O(\log k)$ is the cost to update the top-k plateaus whenever a new plateau is found.

3.3 Depth-Aware Top Down Algorithm

In this section, we present a novel approach that assigns each node v with a *maximum depth* which guarantees that $pl(u, v)$ cannot be longer than the maximum depth. Furthermore, the depths do not necessarily become invalid as the user moves from s_i to s_j, which allows reusing the previous computations for the user's new locations.

3.3.1 Observations

Definition 4. **Deviation vertex & overlapping path**. Let s_i be the user's current location. The deviation vertex of a vertex v w.r.t. s_i is the first vertex after which the shortest path $sp(s_i, v)$ deviates from the user's path $P(s_i, t)$. Formally, the deviation vertex of v is the first vertex s_j on $P(s_i, t)$ such that $P(s_i, s_{j+1})$ is not a sub-path of $sp(s_i, v)$. The path $P(s_i, s_j)$ is called the overlapping path of v because this represents the overlap of $sp(s_i, v)$ and $P(s_i, t)$.

The deviation vertex of v w.r.t. the user's location s_i is denoted as $dv_v^{s_i}$ and the overlapping path is denoted as $P(s_i, dv_v^{s_i})$. Consider the example in Fig. 2(a) where the path is $P = \langle s_1, s_2, s_3, s_4, s_5 \rangle$ and the current location of user is s_1. By the definition, s_4 is the deviation vertex for vertex v_6 because the shortest path from s_1 to v_6 is $\langle s_1, s_2, s_3, s_4, v_6 \rangle$ which deviates from the user's path P at vertex s_4. The overlapping path of v_6 is $\langle s_1, s_2, s_3, s_4 \rangle$. The deviation vertex and overlapping path for v_8 are the same as those of v_6. The deviation vertex of v_{11} is s_3.

The deviation vertex of v can be computed by issuing a shortest path query from s_i to v. Specifically, when we issue a shortest path query from s_i to a vertex v in Algorithm 1 (line 11), for each vertex x on this shortest path $sp(s_i, v)$, we update the deviation vertex of x. Next, we show that the deviation vertex of v does not change as long as the user is on the overlapping path of v (i.e., as long as the user has not crossed the deviation vertex). E.g., in Fig. 2(a), the deviation vertex of v_{11} remains s_3 as long as the user is on the overlapping path $P(s_1, s_3)$, i.e., the user has not crossed s_3. The next lemma formalizes this.

LEMMA 1. *Let $dv_v^{s_i}$ be the deviation vertex of v for the user at location s_i. For every $s_j \in P(s_i, dv_v^{s_i})$, $dv_v^{s_i} = dv_v^{s_j}$.*

PROOF. By definition of the deviation vertex, each $s_j \in P(s_i, dv_v^{s_i})$ lies on the shortest path $sp(s_i, v)$. Therefore, $sp(s_i, v) = sp(s_i, s_j) \oplus sp(s_j, v)$ where $\oplus$ is the concatenation operation. Also, $P(s_i, t) = P(s_i, s_j) \oplus P(s_j, t)$. Since s_j is on the overlapping path, $P(s_i, s_j) = sp(s_i, s_j)$ which implies $P(s_i, t) = sp(s_i, s_j) \oplus P(s_j, t)$. Thus, $sp(s_j, v)$ and the path $P(s_j, t)$ deviate from each other at the same vertex as $sp(s_i, v)$ and the path $P(s_i, t)$ deviate from each other, i.e., $dv_v^{s_i} = dv_v^{s_j}$ □

Hereafter, whenever clear by context, we use dv_v to denote the deviation vertex of v w.r.t. the user's current location.

We have established that the deviation vertex of v does not change unless the user crosses it. When the user crosses it, the algorithm may not know the deviation vertex of v unless it issues another shortest path query which passes through v in which case its new deviation vertex is updated and is known to the algorithm. For example, in Fig. 2, when the user crosses s_3, the deviation vertex of v_{11} is not known to the algorithm unless a shortest path query involving v_{11} is issued. Next, we classify each vertex based on whether its deviation vertex is known to the algorithm or not.

Familiar/unfamiliar vertices. A vertex v for which the algorithm knows its deviation vertex is called a *familiar* vertex. In contrast, a vertex is called an *unfamiliar* vertex if our algorithm does not know its deviation vertex. For each familiar vertex v, in addition to its deviation vertex $dv_v^{s_i}$, we also store $d(s_i, v)$. As we show shortly (Lemma 2), this may help obtaining the shortest distances $d(s_j, v)$ for other vertices s_j on the path in constant time.

Initially (at location s_1), the algorithm knows the deviation vertex for each vertex in the graph (which is recorded during the construction of T_f). So, all vertices are familiar initially. In Fig. 2(a), the deviation vertex for v_5 is s_1 and the deviation vertex for both v_6 and v_8 is s_4. When the user crosses s_1 and reaches s_2, the deviation vertex of v_5 is unknown whereas the deviation vertex of v_6 and v_8 remain s_4. Thus, at s_2, both v_6 and v_8 are familiar vertices whereas v_5 is an unfamiliar vertex. Assume that, at s_2, our algorithm needs to issue a shortest path query $sp(s_2, v_5)$ which returns the path $\langle s_2, s_3, v_3, v_5 \rangle$. As a result, the deviation vertex for all the unfamiliar vertices on this path (i.e., v_3, and v_5) is updated to be s_3 and these vertices become familiar vertices.

Next, we show that the shortest distance to any familiar vertex v can be computed in $O(1)$ as long as the user is on its overlapping path.

LEMMA 2. *Let v be a vertex which became familiar to the algorithm at location s_i, i.e., the algorithm records $d(s_i, v)$ for v. For every vertex s_j on the overlapping path $P(s_i, dv_v^{s_i})$, $d(s_j, v) = d(s_i, v)$ - $d(s_i, s_j)$.*

PROOF. By definition of deviation vertex and overlapping path, s_j is on $sp(s_i, v)$, which implies that $d(s_i, v) = d(s_i, s_j) + d(s_j, v)$. Hence, $d(s_j, v) = d(s_i, v)$ - $d(s_i, s_j)$. □

Using the above lemma, $d(s_j, v)$ can be computed in $O(1)$ because $d(s_i, v)$ is stored by the algorithm and $d(s_i, s_j)$ can also be obtained in $O(1)$. Specifically, at the beginning, for each s_k on the path P, we record $d^P(s_1, s_k)$. Hence, $d^P(s_i, s_j) = d(s_i, s_j) = d^P(s_1, s_j) - d^P(s_1, s_i)$. Note that $d^P(s_i, s_j) = d(s_i, s_j)$ because s_j is on the overlapping path.

Next, we present several lemmas to identify the cases when a vertex v and its child u in T_b *cannot* be on the same plateau. Afterwards, we will present techniques that exploit these to update top-k plateaus efficiently.

LEMMA 3. *Let v be a vertex that became familiar to the algorithm at location s_i. Let u be a child of v in T_b which is an unfamiliar vertex at s_i. Vertices u and v cannot be on the same plateau for the user's location s_i.*

PROOF. We prove this by contradiction. Assume u and v are on the same plateau at s_i. This implies that the shortest path from s_i to v must pass through u, i.e., $u \in sp(s_i, v)$. Recall that when our algorithm issues a shortest path query $sp(s_i, x)$, each vertex on $sp(s_i, x)$ becomes a familiar vertex. Since v became familiar at s_i, this implies $v \in sp(s_i, x)$ for some vertex x for which our algorithm had issued a shortest path query. This implies that $sp(s_i, v) \subseteq sp(s_i, x)$. Since $u \in sp(s_i, v)$, $u \in sp(s_i, x)$. Thus, u must also be a familiar vertex which is not the case. $\square$

Note that the above lemma implies that u and v cannot be on the same plateau if u is unfamiliar but v is familiar. However, this does not imply that u and v must be on the same plateau if u is also familiar. If both u and v are familiar vertices, u and v may or may not be on the same plateau.

LEMMA 4. *Let u be a child of v in T_b. If u is an unfamiliar vertex and v is a familiar vertex, u and v cannot be on the same plateau as long as the user is on the overlapping path of v, i.e., user is on $s_j \in P(s_i, dv_v^{s_i})$.*

PROOF. If u and v are on the same plateau for a location s_j, then $sp(s_j, v)$ must pass through u. We prove that, for each $s_j \in P(s_i, dv_v^{s_i})$, $sp(s_j, v)$ does not pass through u. Lemma 3 implies that u and v are *not* on the same plateau when the user is at s_i, i.e., $sp(s_i, v)$ does not pass through u. By definition of deviation vertex, for each $s_j \in P(s_i, dv_v^{s_i})$, $sp(s_i, v) = sp(s_i, s_j) \oplus sp(s_j, v)$. Since $sp(s_i, v)$ does not pass through u, $sp(s_j, v)$ also cannot pass through u. $\square$

According to the above lemmas, a familiar vertex v and its unfamiliar child u in T_b cannot make a plateau as long as the user has not crossed the deviation vertex of v. Consider the example in Fig. 4 where a backward tree T_b is shown (not related to the running example). Assume that green vertices are familiar and red vertices are unfamiliar. The vertex v_2 and its child v_5 cannot be on the same plateau. Likewise, v_4 and its child v_6 cannot be on the same plateau.

LEMMA 5. *Let u be a child of v in T_b. Let u and v **both** be familiar vertices at a user's location s_i. If u and v are not on the same plateau when the user is at s_i, they cannot be on the same plateau as long as the user is on the overlapping path of v, i.e., $s_j \in P(s_i, dv_v^{s_i})$.*

PROOF. Since u and v are *not* on the same plateau when the user is at s_i, $sp(s_i, v)$ does not pass through u. By definition of deviation vertex, for each $s_j \in P(s_i, dv_v^{s_i})$, $sp(s_i, v) = sp(s_i, s_j) \oplus sp(s_j, v)$. Since $sp(s_i, v)$ does not pass through u, $sp(s_j, v)$ also cannot pass through u. $\square$

In Fig. 4, assume that v_{13} and v_{18} are not on the same plateau at a location s_i. They cannot be on the same plateau as long as the user is on the overlapping path of v_{13}. In Fig. 4, we use the dashed edges to show all cases where it is guaranteed (by Lemmas 4 and 5) that v and its child u cannot be on the same plateau for each location on the overlapping path of v. For the solid edges, the pair of vertices may or may not be on the same plateau.

3.3.2 Depth and Maximum Depth

The lemmas presented in Section 3.3.1 can be used to infer the maximum possible length of any plateau $pl(x, v)$ with v as its *target end*. Below, we introduce the concept of *depth* and *maximum depth* to formalize this.

Definition 5. **Depth.** Let u be a child of v in the backward tree T_b. The depth of v w.r.t. u, denoted as $v.depth[u]$, is the maximum possible length of a plateau $pl(x, v)$ that contains both u and v with v as the target end.

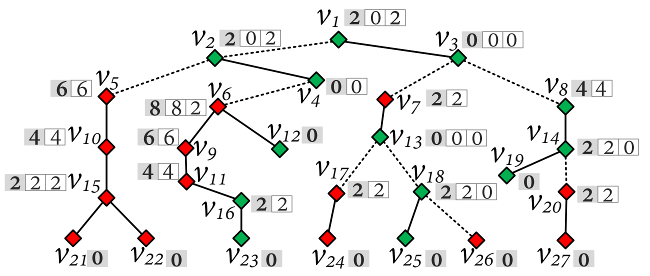


Figure 4: T_b with depth arrays and maximum depths. Red vertices are unfamiliar and green vertices are familiar. The dashed edges show that the two vertices cannot be on the same plateau.

Definition 6. **Maximum depth.** Maximum depth of a node v, denoted as $v.maxdepth$, is the maximum possible length of any plateau $pl(x, v)$ with v as the target end. In other words, $v.maxdepth = max\{v.depth[u] \mid u \in C\}$ where C denotes all children of v in the backward tree T_b.

For each non-leaf node v in T_b, we store its maximum depth along with a depth array that records depths for each of its children. In Fig. 4, depth array for each node is shown along with the maximum depth shown with grey background besides the array. In this example, when the depth array of a node contains two values, the first (resp. second) value in it corresponds to the depth of the node w.r.t. the left (resp. right) child. For simplicity, we assume that each edge has weight 2. As stated earlier, if the algorithm does not yet know whether two nodes are on the same plateau or not, we show the edge in solid line. For v_{14}, its depth w.r.t. right child v_{20} is $v_{14}.depth[v_{20}] = 0$ because v_{14} and v_{20} cannot be on the same plateau (Lemma 4). The depth of v_{14} w.r.t. v_{19} is $v_{14}.depth[v_{19}] = 2$ because v_{14} and v_{19} may or may not be on the same plateau but we set depth by assuming that they are on the same plateau (recall depth corresponds to the upper bound on the length of the plateau). The maximum depth of v_{14} is $max(v_{14}.depth[v_{20}], v_{14}.depth[v_{19}]) = 2$. The depth of v_3 w.r.t. its both children is 0 and its maximum depth is also 0.

Computing maximum depth and depth array. Before we show how to initialize and continuously update depth arrays and maximum depths of each vertex, we first show how to compute the depth array and the maximum depth of a single vertex v *assuming* that the maximum depth for each of its children u has been correctly computed, i.e., $u.maxdepth$ is known for each child u of v. First, we summarize the observations presented in Section 3.3.1.

Case 1: v is familiar but u is unfamiliar. u and v cannot be on the same plateau (Lemma 4), hence, $v.depth[u] = 0$. E.g., $v_{14}.depth[v_{20}] = 0$ and $v_3.depth[v_7] = 0$ in Fig. 4.

Case 2: v and u both are familiar. In this case, u and v may or may not be on the same plateau. We determine whether u and v are on the same plateau or not. **Case 2a:** If u and v are not on the same plateau then $v.depth[u] = 0$ (Lemma 5), e.g., $v_3.depth[v_8] = 0$. **Case 2b:** If u and v are on the same plateau then $v.depth[u] = u.maxdepth + d(u, v)$, e.g., $v_8.depth[v_{14}] = v_{14}.maxdepth + d(v_{14}, v_8) = 2 + 2 = 4$.

Case 3: v is unfamiliar. Note that the lemmas presented in Section 3.3.1 do not apply when v is an unfamiliar vertex (regardless of whether u is a familiar vertex or not). Since $v.depth[u]$ is an upper bound (i.e., maximum length of $pl(x, v)$ with both u and v on it), we assume that v and u are on the same plateau and set $v.depth[u] = u.maxdepth + d(u, v)$. E.g., $v_6.depth[v_9] = v_9.maxdepth + d(v_9, v_6) =$

Algorithm 2: computeDepths(v)

```
1 foreach child $u$ of $v$ do
2     $v.depth[u] = u.maxdepth + d(u, v)$;
3     if $v$ is a familiar node then // Case 1 and 2
4         if $u$ is an unfamiliar node then // Case 1
5             $v.depth[u] = 0$;
6         else // Case 2
7             if $d(s_i, v) \neq d(s_i, u) + d(u, v)$ then
                  // Case 2a
8                 $v.depth[u] = 0$;
9 $v.maxdepth \leftarrow$ maximum value in $v$'s depth array;
```

$6+2=8$ and $v_{11}.depth[v_{16}] = v_{16}.maxdepth + d(v_{16}, v_{11}) = 2+2=4$ in Fig. 4.

Now, we present Algorithm 2 that details how to compute the depth array and maximum depth of v assuming that the maximum depth of each of its children u is known. For each child u, we initialize $v.depth[u] = u.maxdepth + d(u, v)$ assuming Case 2b and Case 3 (line 2). Later, we update $v.depth[u] = 0$ if v and u correspond to Case 1 or Case 2a. Specifically, if v is familiar and u is unfamiliar (Case 1), we set $v.depth = 0$ (line 5). If v and u are both familiar, we check whether u and v are on the same plateau. Recall that u and v are on the same plateau iff $d(s_i, v) = d(s_i, u) + d(u, v)$ (Property 1). Note that $d(u, v)$ is the edge weight and $d(s_i, x)$ for any familiar node x can be computed in $O(1)$ (Lemma 2). Therefore, since u and v are both familiar, we can check in $O(1)$ whether u and v are on the same plateau or not. Otherwise, (Case 2a), we update $v.depth[u] = 0$ (lines 7 and 8). The maximum depth of v is set to be the maximum of all values in the depth array (line 9).

3.3.3 Initializing and Maintaining Depth Arrays

First, we explain how to update depth arrays and maximum depths of affected vertices when a user moves from s_{i-1} to s_i. Afterwards, we explain how to initialize depth arrays and maximum depths.

Maintaining Depth Arrays. As implied by Lemmas presented in Section 3.3.1, the depth arrays and maximum depths remain valid unless at least one familiar (resp. unfamiliar) vertex becomes unfamiliar (resp. familiar). We handle these two cases as follows.

Case 1: Some familiar vertices become unfamiliar. When the user moves from s_{i-1} to s_i, the familiar vertices for which the deviation vertex was s_{i-1} become unfamiliar. Let L be the set of such vertices. We update the depth arrays and maximum depths using Algorithm 3. The algorithm updates the depth arrays and maximum depths in a bottom up approach, i.e., the vertices are processed in descending order of $level(v)$ where level of a vertex v is the number of edges between v and the root of the backward tree T_b. For each vertex v, we call Algorithm 2 to recompute its depth arrays and maximum depth. If the maximum depth of v changes, the maximum depth of its parent may also need to be updated (because the depth array of the parent depends on the maximum depths of its children). In that case, we insert the parent of v in L unless it is already in L.

Since the algorithm processes the affected vertices in a bottom up fashion, this guarantees that the maximum depth of each child u of v is correctly updated before the depth array of v is computed by Algorithm 2. Note that, in the worst case, the above algorithm requires updating the depth array and maximum depth for each vertex at most once. However, in practice, the total number of vertices for which depth arrays need to be updated is much smaller.

Algorithm 3: depthUpdate(L)

```
  Input: $L$: List of affected vertices
1 foreach $v \in L$ in descending order of $level(v)$ do
2     computeDepths($v$);                // Algorithm 2
3     if $v.maxdepth$ changes then
4         insert parent of $v$ in $L$ if not already present
```

Finally, we describe how to efficiently obtain L, the list of vertices that become unfamiliar when a user moves from s_{i-1} to s_i. This can be efficiently done by maintaining inverted lists for each vertex s_i on the path P. Specifically, the inverted list for s_i contains all vertices for which s_i is a deviation vertex. These lists are initialized at s_1 and are maintained as the algorithm finds the new deviation vertices. Thus, the list of affected vertices when the user moves from s_{i-1} to s_i corresponds to the inverted list of s_{i-1}.

Case 2: Some unfamiliar vertices become familiar. As stated earlier, when our query processing algorithm issues a shortest path query from s_i to a vertex v (e.g., line 11 in Algorithm 1), each vertex $x \in sp(s_i, v)$ becomes a familiar vertex and the deviation vertex of x is updated. To update the depth arrays, we set L by inserting in it each vertex $x \in sp(s_i, v)$ which was not already a familiar vertex. Then, Algorithm 3 is called with L as the input to update the depth arrays and maximum depths.

Initialization. When the user is at s_1, we initialize depth arrays and maximum depths of all vertices by calling Algorithm 3 where the input L contains all vertices in T_b.

3.3.4 Algorithm

The Depth-Aware Top Down Algorithm is the same as Algorithm 1 except the following three important differences.

1. Since $v.maxdepth$ is an upper bound on the maximum length of a plateau $pl(x, v)$, we do not need to issue a shortest path query from s_i to v to obtain the plateau if $v.maxdepth$ is smaller than L^k, the length of k-th longest plateau found so far. Specifically, lines 11 to 13 in Algorithm 1 are not processed if $v.maxdepth < L^k$.

2. If v is a familiar vertex, Pruning Rule 2 is used at line 5 instead of Pruning Rule 3, i.e., we use $d(s_i, v)$ instead of the Euclideadn distance $d^E(s_i, v)$ because $d(s_i, v)$ can be computed in $O(1)$ for each familiar vertex (Lemma 2).

3. If $v.maxdepth \geq L^k$, one option to obtain the plateau $pl(x, v)$ is to issue the shortest path query to v. However, if v is a familiar vertex, we can get the plateau $pl(x, v)$ without issuing a shortest path query. Specifically, v and its child u can be on the same plateau only if both u and v are familiar. So, we only need to traverse down the branch with a familiar child u such that u and v are on the same plateau which can be confirmed in $O(1)$ (by checking whether $d(s_i, v) = d(s_i, u) + d(u, v)$ as discussed in Section 3.3.2).

Complexity Analysis: The space used by all the depth arrays is $O(V) \leq O(E)$. This is because for each vertex in T_b, constant space is used in its parent for the depth array. Also, other bookkeeping information (e.g., $dist(s_i, v)$ and familiar/unfamiliar status) uses constant space per vertex. Thus, the total space complexity is $O(E + Sp_{index})$, the same as for the basic top down algorithm.

To analyse time complexity, we first analyse the cost of updating maxdepths. Cost for `computeDepths` (Algorithm 2) is $O(1)$ since the size of depth array is bounded by maximum out-degree which is a small constant in road networks. Cost of `depthUpdate(L)` (Algorithm 3) is $O(V_L)$ where $V_L \leq V$ is the number of vertices with updated maxdepths. Thus, the total cost to maintain maxdepths is

$O(V_L)$ which is bounded by $O(V)$. Each pruning rule takes O(1). Let V_u^{DA} be the number of unpruned and umarked vertices for the depth-aware (DA) algorithm. The algorithm traverses the whole tree, which takes $O(V)$, and issues V_u^{DA} shortest path queries. The cost to update top-k plateaus is $O(\log k)$ per new plateau found. So, the total cost is $O(V + V_u^{DA} \times (SP_{cost} + \log k))$.

3.4 Depth-Aware Best First Algorithm

The depth-aware top down (DA-TD) algorithm processes the backward tree T_b in a top down fashion which is sub-optimal because many of the top-k plateaus usually appear deeper in the tree. A better approach would access the vertices in the descending order of their *maxdepth* values. However, *maxdepth* values of vertices change throughout the algorithm and maintaining a list of vertices sorted on *maxdepth* is expensive, e.g., for large road networks, the backward tree T_b may contain hundreds of thousands of vertices. Next, we propose a novel solution which uses *root lists* to allow accessing the vertices in the best first order without requiring expensive maintenance.

3.4.1 Root Lists

We conceptually partition the backward tree T_b into a set of disjoint subtrees. Fig. 5 shows such a conceptual partitioning of the tree T_b shown in Fig. 4. Based on the observations presented in Section 3.3.1, we partition T_b using two rules: 1) A familiar vertex v and its unfamiliar child u in T_b cannot be in the same conceptual partitions (i.e., subtrees) because they cannot be on the same plateau (Lemma 4); 2) Two familiar vertices u and v cannot be in the same subtree if u and v are not on the same plateau (Lemma 5). These two rules can be summarized as follows. Vertex v and its child u are in two disjoint subtrees if $v.depth[u] = 0$. Such instances are represented in Fig. 5 using dashed edges and, note that, each dashed edge connects two disjoint subtrees,

Given the tree T_b which is conceptually partitioned into disjoint subtrees as described above, we keep the roots of these subtrees in two separate lists: 1) a sorted root list (denoted as RL_S); and 2) an unsorted root list (denoted as RL_U). Specifically, we use a threshold η and insert every root node v with $v.maxdepth > \eta$ in RL_S which is kept sorted in descending order of the *maxdepth* values. Every other root node is pushed to unsorted RL_U. Consider the example in Fig. 5 and assume that the threshold is $\eta = 3$. The sorted root list is $RL_S = \{v_6, v_5, v_8\}$ (sorted on maximum depths 8, 6 and 4, respectively). The unsorted root list RL_U contains all other root nodes not necessarily sorted, i.e., $v_1, v_2, v_{26}, v_{20}, v_{17}, v_{18}$ and v_7. Note that the total number of vertices in the backward tree T_b is 27 whereas the root list contains 10 roots (equal to the number of conceptual subtree). In our experiments (see Fig. 9(b)), we show that the number of nodes in the sorted root list RL_S is around 100 for all road networks even when the backward tree contains up to 1 million nodes. Fig. 9(b) also shows that the unsorted list RL_U contains less than 10% of the nodes in T_b. Furthermore, in majority of cases, we are able to compute the top-k plateaus by using only the root nodes in RL_S whereas RL_U is rarely accessed during query processing.

3.4.2 Maintaining Root Lists

Recall that a vertex v and its child u are in two different subtrees iff $v.depth[u] = 0$. Therefore, we only need to handle the situations when $v.depth[u]$ changes to zero from a non-zero value and when it changes from zero to non-zero.

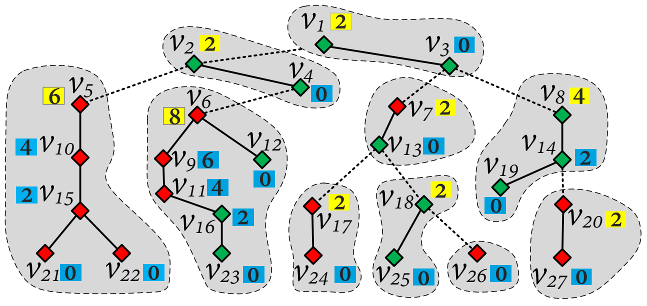


Figure 5: T_b with conceptual partitions. Only *maxdepth* values are shown – depth arrays are omitted for brevity.

<u>*Case 1: $v.depth[u]$ becomes zero.*</u> If $v.depth[u]$ becomes 0, this means u and v are now in two different sub-trees (and u is the root of the new subtree). In this case, u is inserted in the root list with $u.maxdepth$.
<u>*Case 2: $v.depth[u]$ becomes non-zero.*</u> In this case, u and v were in two different subtrees but now the subtree rooted at u is merged with the subtree that contains v. So, u is removed from the root list.
<u>*Case 3: $v.maxdepth$ is updated.*</u> If v is a root node, its *maxdepth* is updated in the root list. Furthermore, it is moved from RL_S to RL_U or vice versa depending on whether $v.maxdepth$ is bigger than the threshold η or not.

For each vertex v in T_b, we maintain pointers to their locations in RL_S or RL_U (the pointer is set to null if the vertex is not in any of the two lists). These pointers allow efficiently updating the root lists.

Note that the above cases can be easily detected and handled in Algorithm 3. In Fig. 5, assume that v_1 becomes unfamiliar and v_{10} becomes familiar. We call Algorithm 3 by passing to it v_{10} and v_1. Since these vertices are processed based on level, v_{10} is processed first. As v_{10} is now familiar, `computeDepths` (Algorithm 2) sets $v_{10}.depth[v_{15}] = 0$. This implies that v_{15} is the root of a new subtree. So, v_{15} is inserted in the root list with its maximum depth, i.e., $v_{15}.maxdepth = 2$. Since $v_{10}.maxdepth$ has changed (to 0) and, as per Algorithm 3, its parent v_5 is inserted in L. In the next iteration, v_5 is processed. Both $v_5.depth[v_{10}]$ and $v_5.maxdepth$ are updated to $v_{10}.maxdepth + d(v_{10}, v_5) = 0 + 2 = 2$. Since $v_5.depth[v_{10}]$ was non-zero and is still non-zero, this does not indicate a subtree split. However, $v_5.maxdepth$ is updated from 6 to 2. Since v_5 is a root node, its *maxdepth* is updated in the root list. Assuming $\eta = 3$, v_5 is moved from RL_S to RL_U. As $v_5.maxdepth$ has changed, its parent v_2 is inserted in L and processed but its depth array and *maxdepth* do not change. Finally, node v_1 is processed. As v_1 has now become an unfamiliar node, $v_1.depth[v_2] = v_2.maxdepth + d(v_2, v_1) = 2 + 2 = 4$. Since $v_1.depth[v_2]$ was zero and now has become non-zero (Case 2), the subtree rooted at v_2 is merged with the subtree that contains v_1. Hence, v_2 is removed from the corresponding root list. Finally, $v_1.maxdepth = 4$ which is updated to 4 in the root list and moved from RL_U to RL_S.

3.4.3 Algorithm

The key idea behind our Depth-Aware Best First (DA-BF) Algorithm (Algorithm 4) is that it traverses each subtree in the sorted root list in the descending order of the maxdepths of their root nodes (lines 2 and 3). Each traversal (line 3) is similar to the DA-TD algorithm except that here it does not traverse the subtree of v if $v.maxdepth \leq L^k$.

Table 2: Complexity analysis. V is # of vertices in the shortest path tree. V_u^{TD} (resp. V_u^{DA}) is # of shortest path queries issued by the top-down algorithm (resp. the two depth-aware algorithms). V_L is # of vertices for which maxdepth changes at the timestamp and $V^>$ is # of vertices in T_b with maxdepth greater than L^k. SP_{index} and SP_{cost} are the index size and cost of the shortest path algorithm used. E is # of edges in the graph.

	Tree-Join	**Top-Down**	**Depth-Aware Top-Down**	**Depth-Aware Best-First**
Space	$O(E)$	$O(E + SP_{index})$	$O(E + SP_{index})$	$O(E + SP_{index})$
Time	$O(E \log V)$	$O(V + V_u^{TD}(SP_{cost} + \log k))$	$O(V + V_u^{DA}(SP_{cost} + \log k))$	$O(V_L + V^> + V_u^{DA}(SP_{cost} + \log k))$

Note that it is possible that the unsorted root list RL_U contains a top-k plateau if a root node in RL_U has maximum depth greater than L^k. During the update of root lists, we maintain $maxdepth(RL_U)$ which denotes the maximum maxdepth in RL_U. If $maxdepth(RL_U) > L^k$, the algorithm processes the entries in it (line 4). Each entry e is accessed and if $e.maxdepth > \eta$, the entry is moved to RL_S (line 6). Also, if $e.maxdepth > L^k$, the subtree rooted at e is traversed as described above to update the top-k plateaus and L^k (line 8). The algorithm also updates $maxdepth(RL_U)$ (line 9) which may have been changed as some entries are moved to RL_S. Note that the algorithm accesses each unpruned vertex v with maxdepth greater than L^k which ensures its correctness.

Algorithm 4: Depth-Aware Best First Algorithm

```
1  use plateaus from $s_{i-1}$ to initialize $R_i$ and $L^k$;
2  foreach $e$ in $RL_S$ with $e.maxdepth > L^k$ do
3      traverse tree rooted at $e$ to update $R_i$ and $L^k$;
4  if $maxdepth(RL_U) > L^k$ then
5      foreach $e$ in $RL_U$ with $e.maxdepth > \eta$ do
6          move $e$ from $RL_U$ to $RL_S$;
7          if $e.maxdepth > R^L$ then
8              traverse subtree of $e$ to update $R_i$&$L^k$;
9      update $maxdepth(RL_U)$;
10 $\eta = L^k/\alpha$;
11 return $R_i$
```

Just before returning the top-k plateaus, we also update $\eta = L^k/\alpha$ (line 10), where $\alpha > 1$ and helps setting the threshold relative to L^k. Our experimental study shows that $\alpha = 5$ is a reasonable choice. The intuition behind setting η several times smaller than L^k is to ensure that, in most cases, the top-k plateaus can be found from RL_S without the need to process RL_U. Also, since L^k varies throughout the algorithm, η also changes. However, unless a root node is implicitly accessed in the root list, we do not move it from RL_S to RL_U or vice versa. This does not affect the correctness of the algorithm.

Complexity Analysis: Compared to the DA-TD algorithm, the only additional data structure needed by the DA-BF algorithm consists of the two root lists with total size $O(V) \leq O(E)$. Thus, the space complexity is $O(E + SP_{index})$.

Now, we analyse time complexity. As earlier, the cost to maintain maxdepths is $O(V_L)$ where V_L is the number of vertices with updated maxdepths. We analyse the cost for maintaining the root lists. Insertion/deletion cost for sorted (resp. unsorted) root list is $O(\log \beta)$ (resp. $O(1)$) where β is the size of the sorted root list. Note that at most V_L entries are inserted/removed in/from the two lists. Let m (resp. $V_L - m$) be the number of entries inserted/removed in/from the sorted (resp. unsorted) root list. The total maintenance cost is $O(V_L + m \log \beta + (V_L - m)) = O(V_L + m \log \beta)$.

Let $V^>$ be the number of vertices in T_b with maxdepth larger than L^k. The algorithm traverses at most $V^>$ vertices during the traversal of all subtrees in the two root lists. Let γ be the number of entries in the unsorted list. In the worst case, the algorithm may also access all entries in the unsorted root list (when the results cannot be computed using the sorted list). So, the algorithm accesses $V^> + \gamma$ vertices in the worst case. The algorithm issues V_u^{DA} shortest path queries where V_u^{DA} is the number of unpruned and unmarked vertices (the same as the DA-TD algorithm). Thus, the cost of Algorithm 4 is $O(V^> + \gamma + V_u^{DA} \times (SP_{cost} + \log k))$. The total cost including the maintenance of maxdepths and root lists is $O(V_L + m \log \beta + V^> + \gamma + V_u^{DA} \times (SP_{cost} + \log k))$.

Comparison of complexities: Table 2 summarizes the space and time complexities of all the algorithms. For the DA-BF algorithm, we simplify the complexity by removing β, γ and m because our experimental study shows that $\beta \approx 100$ for all different road networks (Fig. 9(b)). Furthermore, Fig. 9(b) also shows that $\gamma \ll V$ for all road networks. Also, although not shown in the experimental study due to the space limitations, $m < 10$ and the unsorted root list, containing γ nodes, is rarely accessed by DA-BF (around 1–2 times out of 100). Thus, the cost is simplified to $O(V_L + V^> + V_u^{DA} \times (SP_{cost} + \log k))$. As shown in Fig. 9(c), V_u^{DA} is less than 2 on average, which is significantly smaller than V_u^{TD}. This explains why the two depth-aware algorithms are significantly faster than the basic top-down (TD) algorithm. Finally, Fig. 9(c) also shows that $V_L + V^>$ is (around 30) which is up to 2 orders of magnitude smaller than V. This explains why the DA-BF algorithm outperforms the DA-TD algorithm.

4. EXPERIMENTS

4.1 Settings

Algorithms and environment. We compare four algorithms that continuously monitor top-k alternative paths, namely the TJ algorithm (Section 3.1), the TD algorithm (Section 3.2), the DA-TD algorithm (Section 3.3), and the DA-BF algorithm (Section 3.4). For the DA-BF algorithm, we studied the effect of $\eta = L^k/\alpha$ by using different values of α and observe that the best performance is achieved for $\alpha = 5$ which is used as the default value for DA-BF.

Datasets: Table 3 shows details of the eight real road networks in the USA that we use in the experiments (downloaded from DIMACS[5]). Similar to some existing works (e.g., see [35]), we fix some issues such as removing self-loops and unconnected components from the graphs.

Parameters: We investigate the effect of varying k, length of path P, upper bound parameter ϵ, and the road network size. We vary k from 2 to 16 where the default value is 6. Note that $k = 6$ means the shortest path and 5 other alternative paths are reported to the user. Similar to previous

[5]http://www.dis.uniroma1.it/challenge9

Table 3: Road Network Datasets

Name	Region	#Vertices	#Edges
NY	New York City	264,346	733,846
COL	Colorado	435,666	1,042,400
NW	North-west US	1,207,945	2,545,844
NE	North-east US	1,524,453	3,897,636
E-US	Eastern US	3,598,623	8,708,058
W-US	Western US	6,262,104	15,119,284
C-US	Central US	14,081,816	33,866,826
USA	United States	23,947,347	57,708,624

works [5, 25] on the snapshot queries, ϵ is varied from 1.05 to 1.25 and 1.15 is the default value. Alternative paths are continuously reported for each location s_i on the user's path and, unless mentioned otherwise, the results report *average* CPU time for each location, i.e., the total CPU time divided by the number of vertices on the path. For each experiment, we run 1000 queries and present the *average* cost.

4.2 Performance Evaluation

Varying Query Path Length. In this experiment, we vary the distance between source and target of the CTKP query. More specifically, similar to several existing studies on the shortest path queries (e.g., see [34, 27]), we generate eight groups of queries $Q_1, Q_2, \ldots, Q_8$ as follows: we set l_{min} to 10 km and set l_{max} to the maximum distance of any pair of vertices in the road network. Let $z = (l_{max}/l_{min})^{1/8}$. For each $1 \leq i \leq 8$, we generate 1,000 queries to form Q_i, in which the distance between the source and target vertices for each query pair fall in the range $(l_{min} \times z^{i-1}, l_{min} \times z^i]$. Note that the distances of queries increase exponentially in each group from Q_1 to Q_8.

The experimental results for CTKP queries in each group from Q_1 to Q_8 are shown in Fig. 6 for four different road networks. As expected, the query processing time increases with the increase of the path length, which is mainly because the search space increases with the path length. The TD algorithm performs worst, mainly because, despite pruning, it still requires issuing a large number of shortest path queries. We were not able to run TD for bigger road networks and, hereafter, we omit TD from the comparisons. The two depth-aware algorithms outperform the TJ and TD algorithms by up to several orders of magnitude, especially for bigger road networks. This shows the effectiveness of the ideas proposed in Sections 3.3 and 3.4. Furthermore, DA-BF is several times faster than DA-TD showing the effectiveness of maintaining root lists and accessing the tree in the best first order. Even for the largest road network (USA), DA-BF can continuously monitor k alternative paths in around 1 ms on average per location s_i.

Varying k. We vary k and show the results in Fig. 7. The cost of TJ does not change with k because it computes T_f and joins it with T_b regardless of the value of k. The cost of our algorithms increases with increasing k because the algorithms need to compute more plateaus which increases the space searched by our algorithms. Our best algorithm DA-BF is more than two orders of magnitude faster than the TJ algorithm even for the largest k. Also, DA-BF is 3-6 times faster than DA-TD, which shows the effectiveness of the proposed best first traversal.

Varying ϵ. In this experiment, we study the effect of the upper bound distance parameter ϵ and report the results in Fig. 8. As expected, the cost of each algorithm increases with the increase in ϵ because the search space (i.e., T_b) becomes larger for a larger ϵ. However, DA-BF outperforms TJ by more than two orders of magnitude for all values of ϵ on bigger road networks. Also, DA-BF is several times faster than DA-TD.

Effectiveness of optimisations. Fig. 9 shows the effectiveness of different optimisations presented in the paper. Specifically, Fig. 9(a) shows the effectiveness of different pruning rules. The No-UB algorithm is the same as DA-TD except that it does not use the pruning rules that prune the search space using the upper bound distance $d^P(s_i, t) \times \epsilon$. Recall that TD uses the Euclidean distance $d^E(s_i, v)$ to apply the pruning rules. However, DA-TD uses the exact distance $d(s_i, v)$ if v is a familiar vertex because $d(s_i, v)$ can be obtained in $O(1)$ for the familiar vertices. The No-FUB uses the upper bound pruning rules but does not use this optimisation for the familiar vertices, i.e., it uses Euclidean distance instead of the exact shortest distance for the familiar vertices. As shown in Fig. 9(a), each proposed optimisation significantly improves the performance of the algorithm for different groups of queries Q_1 to Q_8.

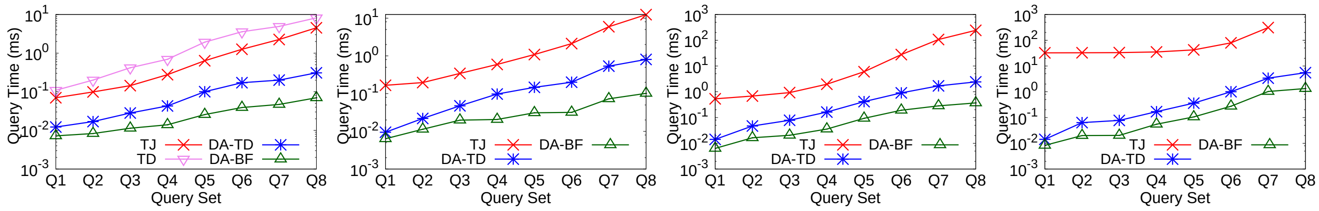
Fig. 9(b) shows the size of two root lists RL_S and RL_U, the total size of two root lists RL and the total number of valid nodes in T_b for different road networks. Note that the size of RL_S is around 100 for all road networks. This shows the effectiveness of the root list which maintains only around 100 nodes in sorted order (and the algorithm can find the top-k plateaus using these nodes in most of the cases). Also, the total size of RL is 10 times smaller than the size of T_b.

Fig. 9(c) shows different variables used in the complexity analysis (Table 2). Recall that V_u^{TD} (resp. V_u^{DA}) correspond to the number of shortest path queries issued by TD (resp. DA-TD and DA-BF) per location s_i. Fig. 9(c) shows that DA-TD and DA-BF need to issue less than 2 shortest path queries per location of s_i on average. This shows the effectiveness of maintaining *maxdepth* values. TD requires issuing up to 1000 shortest path queries on average for each location, which explains the reason why it is slow. Nevertheless, it is able to prune a lot of vertices in T_b, i.e., V_u^{TD} is up to two orders of magnitude smaller than V, the size of T_b. Also, $V_L + V^>$ is significantly smaller than V, which explains why DA-BF outperforms DA-TD.

Finally, Fig. 9(d) shows the average number of familiar nodes, unfamiliar nodes and total nodes in T_b for each location s_i as the user moves on P towards the target. Most of the nodes are familiar, which explains the effectiveness of the depth-aware algorithms. Also, as expected, the size of T_b decreases as the user gets closer to the target.

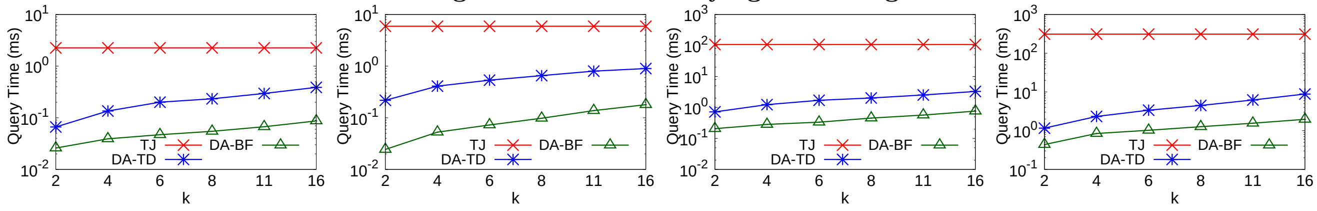
Different road networks (CPU time and memory). Fig. 10 shows the CPU time and memory used by each algorithm for all road networks shown in Table 3. The x-axis shows the road networks in increasing order of their sizes. As expected, Fig. 10(a) shows that the running time of each algorithm increases with the increase in the size of the road network. DA-BF consistently outperforms TJ and DA-TD for all data sets.

The memory used by each algorithm is shown in Fig. 10(b). "Graph" corresponds to the size of the input graph (road network). As expected, DA-BF and DA-TD consume more memory than TJ and TD due to the additional data structures employed (e.g., depth arrays, root lists). However, the memory used by each algorithm increases linearly with the graph size. Recall that the algorithms also need a shortest path index to efficiently compute shortest paths. However, in Fig. 10(b), we do not include the memory used by the shortest path index due to two main reasons: 1) the memory usage depends on the specific shortest path algorithm employed; 2) almost all navigation systems already have a shortest path index which



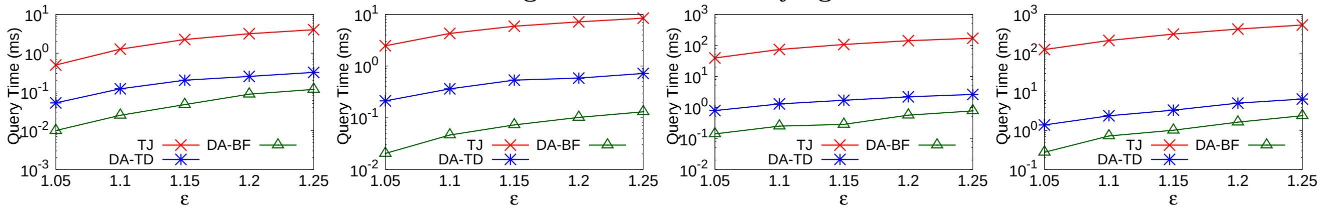
(a) NY (b) NW (c) E-US (d) USA

Figure 6: Effect of Varying Path Length



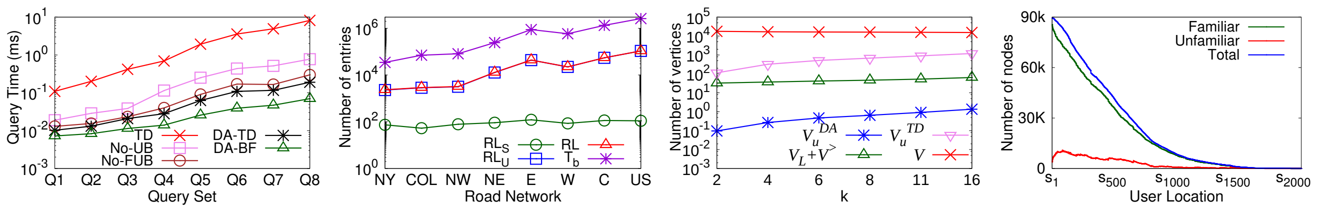
(a) NY (b) NW (c) E-US (d) USA

Figure 7: Effect of Varying k



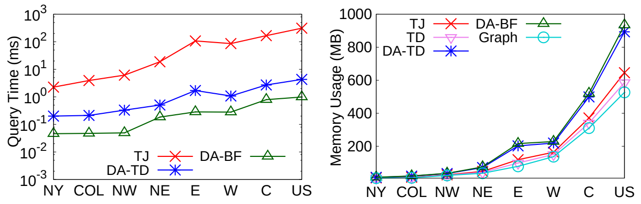
(a) NY (b) NW (c) E-US (d) USA

Figure 8: Effect of Varying ϵ



(a) Prune rules (b) Root list size (c) Comp. analysis variables (d) # of familiar nodes

Figure 9: Effectiveness of optimisations



(a) Running time (b) Memory usage

Figure 10: All road networks

our algorithms can directly use (thus Fig. 10(b) represents the memory overhead for each of our algorithms). We used pruned highway labelling (PHL) [7] in our implementation with index size around 16 GB for the US road network as reported in [7] (which dominates the overall memory used by our algorithms). If the index size is an issue, one can use other shortest path algorithms such as Contraction Hierarchies (CH) [18] or Arterial Hierarchy (AH) [37], which have an order of magnitude smaller indexes but provide comparable or better running time for the shortest path queries [23].

5. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, this is the first paper to study the problem of continuously monitoring k alternative paths as the user is moving on a path towards the target. Due to the popularity of plateaus-based alternative paths, we also continuously report alternative paths based on plateaus. We present several algorithms each improving on the previous using some non-trivial observations and novel optimisations. We provide complexity analysis of our algorithms and a comprehensive experimental study using real-world road networks and demonstrate the efficiency of the proposed algorithms. An important direction for future work is to design techniques for snapshot and continuous top-k alternative path queries for dynamic road networks and time-dependent road networks.

Acknowledgements. Muhammad Aamir Cheema is supported by ARC FT180100140 and DP180103411. Hua Lu's work was partly conducted when he was employed at Aalborg University, Denmark.

6. REFERENCES

[1] T. Abeywickrama, M. A. Cheema, and A. Khan. K-SPIN: Efficiently processing spatial keyword queries on road networks. *IEEE TKDE*, 32(5):983–997, 2019.

[2] T. Abeywickrama, M. A. Cheema, and D. Taniar. K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *PVLDB*, 9(6):492–503, 2016.

[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*. Springer, 2011.

[4] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms*, 2012.

[5] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Alternative routes in road networks. *Journal of Experimental Algorithmics (JEA)*, 18:1–3, 2013.

[6] V. Akgün, E. Erkut, and R. Batta. On finding dissimilar paths. *European Journal of Operational Research*, 121(2):232–246, 2000.

[7] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *ALENEX*, pages 147–154, 2014.

[8] R. Bader, J. Dees, R. Geisberger, and P. Sanders. Alternative route graphs in road networks. In *International Conference on Theory and Practice of Algorithms in (Computer) Systems*, 2011.

[9] F. Barth and S. Funke. Alternative routes for next generation traffic shaping. In *Proceedings of the 12th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, 2019.

[10] S. Cabello, E. W. Chambers, and J. Erickson. Multiple-source shortest paths in embedded graphs. *SIAM Journal on Computing*, 42(4):1542–1571, 2013.

[11] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Continuous monitoring of distance-based range queries. *IEEE TKDE*, 23(8):1182–1199, 2010.

[12] M. A. Cheema, W. Zhang, X. Lin, Y. Zhang, and X. Li. Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. *The VLDB Journal—The International Journal on Very Large Data Bases*, 21(1):69–95, 2012.

[13] Y. Chen, M. G. Bell, and K. Bogenberger. Reliable pretrip multipath planning and dynamic adaptation for a centralized road navigation system. *IEEE Trans. on Intelligent Transportation Systems*, 2007.

[14] T. Chondrogiannis, P. Bouros, J. Gamper, U. Leser, and D. B. Blumenthal. Finding k-dissimilar paths with minimum collective length. In *ACM SIGSPATIAL*, pages 404–407, 2018.

[15] T. Chondrogiannis, P. Bouros, J. Gamper, U. Leser, and D. B. Blumenthal. Finding k-shortest paths with limited overlap. *The VLDB Journal*, pages 1–25, 2020.

[16] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1959.

[17] H. Döbler and B. Scheuermann. On computation and application of k most locally-optimal paths in road networks. 2016.

[18] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[19] A. H. Jones. Method of and apparatus for generating routes, Aug. 21 2012. US Patent 8,249,810.

[20] M. Kobitzsch. An alternative approach to alternative routes: Hidar. In *European Symposium on Algorithms*, pages 613–624. Springer, 2013.

[21] M. Kobitzsch, M. Radermacher, and D. Schieferdecker. Evolution and evaluation of the penalty method for alternative graphs. In *ATMOS*, 2013.

[22] L. Li, M. A. Cheema, H. Lu, M. E. Ali, and A. N. Toosi. Comparing alternative route planning techniques: A web-based demonstration and user study. *arXiv preprint arXiv:2006.08475*, 2020.

[23] Y. Li, M. L. Yiu, N. M. Kou, et al. An experimental study on hub labeling based shortest path algorithms. *PVLDB*, 11(4):445–457, 2017.

[24] H. Liu, C. Jin, B. Yang, and A. Zhou. Finding top-k shortest paths with diversity. *IEEE TKDE*, 2017.

[25] D. Luxen and D. Schieferdecker. Candidate sets for alternative routes in road networks. *Journal of Experimental Algorithmics (JEA)*, 19:2–7, 2015.

[26] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *Proceedings of the 32nd international conference on Very large data bases*, pages 43–54. VLDB Endowment, 2006.

[27] D. Ouyang, L. Qin, L. Chang, X. Lin, Y. Zhang, and Q. Zhu. When hierarchy meets 2-hop-labeling: efficient shortest distance queries on road networks. In *SIGMOD*, pages 709–724, 2018.

[28] D. Ouyang, L. Yuan, L. Qin, L. Chang, Y. Zhang, and X. Lin. Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees. *PVLDB*, 13(5):602–615, 2020.

[29] A. Paraskevopoulos and C. D. Zaroliagis. Improved alternative route planning. In *ATMOS*, 2013.

[30] C. Salgado, M. A. Cheema, and M. E. Ali. Continuous monitoring of range spatial keyword query over moving objects. *World Wide Web*, 21(3):687–712, 2018.

[31] Z. Shao, M. A. Cheema, D. Taniar, and H. Lu. VIP-tree: An effective index for indoor spatial queries. *PVLDB*, 10(4):325–336, 2016.

[32] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, 2007.

[33] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, pages 75–75. IEEE, 2006.

[34] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *PVLDB*, 5(5):406–417, 2012.

[35] B. Yao, Z. Chen, X. Gao, S. Shang, S. Ma, and M. Guo. Flexible aggregate nearest neighbor queries in road networks. In *ICDE*, pages 761–772, 2018.

[36] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE TKDE*, 2015.

[37] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *SIGMOD*, 2013.