A Continuous Query System for Dynamic Route Planning

Nirmesh Malviya #1, Samuel Madden #2, Arnab Bhattacharya *3

#MIT CSAIL *CSE, IIT Kanpur

1nirmesh@csail.mit.edu 2madden@csail.mit.edu 3arnabb@cse.iitk.ac.in

***Abstract*—In this paper, we address the problem of answering continuous route planning queries over a road network, in the presence of updates to the delay (cost) estimates of links. A simple approach to this problem would be to recompute the best path for all queries on arrival of every delay update. However, such a naive approach scales poorly when there are many users who have requested routes in the system.**

Instead, we propose two new classes of approximate techniques – K*-paths* and *proximity measures* to substantially speed up processing of the set of designated routes specified by continuous route planning queries in the face of incoming traffic delay updates. Our techniques work through a combination of pre-computation of likely good paths and by avoiding complete recalculations on every delay update, instead only sending the user new routes when delays change significantly. Based on an experimental evaluation with 7,000 drives from real taxi cabs, we found that the routes delivered by our techniques are within 5% of the best shortest path and have run times an order of magnitude or less compared to a naive approach.

I. Introduction

This paper considers the problem of building a *traffic aware route planning service*. The idea is that users register *continuous routing queries*, specifying a set of <startpoint, endpoint> tuples. In response, the system monitors delays and sends query results as updates to users (e.g., via email or SMS) if the fastest route between any of these designated start-end pairs changes, based on real-time updates to traffic delays (in our evaluation, we use real-time delays from a traffic monitoring deployment we have done on a network of 30 taxis in the Boston area). Like existing route planning services (e.g., Google Maps), our system uses a graph of road segments, and applies shortest-path planning algorithms to that graph to recommend routes to users. Unlike existing systems, however, our system maintains a large number of routes for *pre-designated* <source, target> pairs and updates those routes as traffic delays on road segments change. Our system additionally allows users to specify day and time slots along with the start and end points if they do not desire continuous monitoring (registration tuples in this case are of the form <startpoint, endpoint, day, time>). In this paper, we focus on the harder problem where all user queries require continuous monitoring.

Because small variations in delay won't significantly affect a user's travel time, we are not concerned with always finding the exact optimal route for any <start, end>pair but in detecting if a previously reported route has become substantially non-optimal in the face of updates and in providing a new route that is near-optimal and much better (but not necessarily the exact optimal). By suggesting alternate time-saving routes *before* the user begins to drive, we believe our service could prove extremely useful to commuters who tend to get stuck in peak hour traffic congestions. Our system is practically motivated: we have an iPhone application, iCarTel (available on the app store) that we are extending with the methods in this paper.

While adding support for ad-hoc traffic aware routing queries is intuitively simple, it is not immediately clear how a service could practically maintain the large number of designated routes it would need to continuously keep updated. Road network graphs contain millions of vertices and edges; even a sub-graph corresponding to a city and its surrounding suburbs can contain tens of thousands of segments – for instance, the subgraph corresponding to Boston's road network has nearly 40,000 links [1]. A naive recalculation of the optimal route on arrival of every update (or a set of updates) using a single source shortest-path algorithm such as Dijkstra's algorithm (or A* search) for all registered continuous routing queries could turn out to be a major computational overhead given that real time traffic updates usually affect only a small part of the network. Though such an approach might work if the number of registered continuous routing queries is relatively small, it is unlikely to scale as the number of queries go up. Algorithms that are able to update shortest paths in the presence of link changes do exist (e.g., [2]), but they typically have a higher space or computation overhead than is acceptable for our setting (see Section II for a discussion).

In this paper, we discuss several new techniques for maintenance of a large number of routes (specified as part of continuous planning queries) in the face of real time traffic delays. These involve a mix of precomputation and on-the-fly route calculation. Our algorithms are approximate, as they only report changes when delays change significantly (governed by threshold fraction ϵ and factor γ, both tunable). The methods we describe fall into two broad classes. One pre-computes a set of candidate routes and dynamically selects the best candidate as delays change. The other is *proximity based*, recomputing optimal routes when delays change in a "region of influence" around the source and destination routes. To the best of our knowledge, no prior work has addressed this problem at a scale larger than for graphs with several hundred nodes and few thousand edges. To summarize, our contributions in this paper are as follows:

1) We briefly survey several existing dynamic shortest path algorithms and discuss their limited applicability to our

problem.

2) We describe various techniques for efficient online maintenance of registered routes and test how well they perform in practice using several thousand hours of delay data from a network of taxis in the Boston area.
3) Through intensive experiments on real data, we see that the proximity based approach has a run time cost an order of magnitude smaller than for a complete recalculation approach while the *candidate routes* technique is two orders of magnitude faster. The *suggested routes* for both the approaches are within about 5% of the optimal in terms of cost, while a naive scheme which is *purely* precomputation-based produces routes that are more than 20% worse than optimal.

As a caveat, we note that in this paper we focus on the case where the origin and destination nodes are in the same urban area (about 50 miles between each source-destination pair). For traffic-aware commuting, we believe this is a reasonable assumption as the vast majority of commutes are within the same urban area. This *closeness* constraint keeps the running time of our pre-computation methods to a minimum; we believe it should be possible to extend our techniques to work with the many algorithms for scalable long-distance planning (see Section II) but chose not to focus on this problem initially.

The rest of this paper is organized as follows. We discuss related work in Section II, our system architecture in Section III, and our techniques in Section IV and V. We experimentally evaluate our system in Section VI and conclude in Section VII.

II. Related Work

Dijkstra's algorithm has traditionally been used [3] for route planning in road networks. Though extremely efficient implementations of Dijkstra's algorithm exist [4], even the tens of milliseconds of computation time they take per route translates into an untenable solution for our problem given that our ultimate aim is to be able to scale our system to potentially hundreds of thousands of users (leading to hundreds of thousands of standing routing queries). For the same reason, hill climbing algorithms such as A* search [5] which employ heuristics for search pruning do not serve our purpose if they are merely used for complete recomputations on every update. Dynamic variants of A* search such as *dynamic anytime A** [6] and *life long planning A** [7] also do not help us due to scalability issues in using them at run time for a large set of registered routes.

The performance of 1:n *dynamic shortest path algorithms* that update routes from a given source node in response to changes in the weights of one or more edges varies in practice depending upon the frequency of updates as well as the size of the underlying graph [2]. In general, they process edge cost updates much faster in comparison to from-scratch recomputations of their static counterparts. However, they have other limitations that make them unusable in our setting. For example, Ramalingam's incremental algorithm [8] for the dynamic single-source shortest (DSSS) path requires in-memory storage of a pre-calculated shortest-path-tree from every source node. In the setting we consider, this translates to a prohibitive space overhead – thousands of distinct nodes are routinely registered as source nodes. In contrast, our approaches require monitoring just one or a few paths for each registered query, amounting to approximately 2500 bytes of storage per routing request when multiple paths are monitored. Though the similar DynamicSWSF-FP algorithm [8] does not require pre-calculation of any input, its running time is governed by the number of affected nodes which can be large as we need to maintain shortest paths from many source nodes. Frigioni's DSSS algorithm [9] also requires a prohibitive shortest path tree computation.

Dynamic all-pairs-shortest-path algorithms have a space and time overheads(per delay update) of at least $O(n^2)$ and $O(n^2 \log n)$[2], making their use impractical for large road networks and given the high number and frequency of updates we need to process. For instance, King's dynamic all-pairs shortest path (APSP) algorithm [10] supports approximate shortest path queries in an efficient $O(n^2 \log^2 n / \log \log n)$ amortized time but is applicable only when all edge weights are integers bounded above by a small constant. We would need to set this bounding constant to a large value to take the variability in traffic conditions into account, making the solution unattractive. Demetrescu and Italiano's dynamic APSP approach [11] uses combinatorial properties of graphs to guarantee $O(n^2 \log^3 n)$ amortized time per update but the $O(mn)$ space requirement makes their solution unattractive (here m is the number of edges in the graph). In contrast, our K-candidate-routes algorithm runs in $O(Kr + u)$ time, where r is the total number of continuous routing queries registered with the system, and our proximity approach runs in worst case $O(r(u + (m + n \log n))$ for a batch of delay updates of size u.

Work on adaptive fastest path computation by Gonzalez [12] uses historical data to mine traffic patterns at different points of time in an attempt to take non-concrete parameters about different road segments into account. Only historical information is used in their algorithm with no real time traffic monitoring. Kanoulas et al [13] use historical data to get a different average speed for each hour of the day and uses that to compute the best route for a given day and time. Again, real-time traffic conditions are not taken into account.

A precomputation based solution to APSP problem has a significant space overhead but this can be reduced by using approaches based on hierarchical decomposition [14] of the road network, such as precomputation of paths from all source nodes only up to a certain distance or breaking the underlying graph into a set of several fragmented graphs and a boundary graph. Materialization strategies [15] for hierarchical routing algorithms also offer a balance between increased space usage and reduced query time and are considered to be important in querying large spatial network databases [16]. But as discussed earlier, any scheme which merely returns precomputed results (without taking into account dynamic edge costs) defeats the purpose of our system.

The current application of shortest path algorithms to

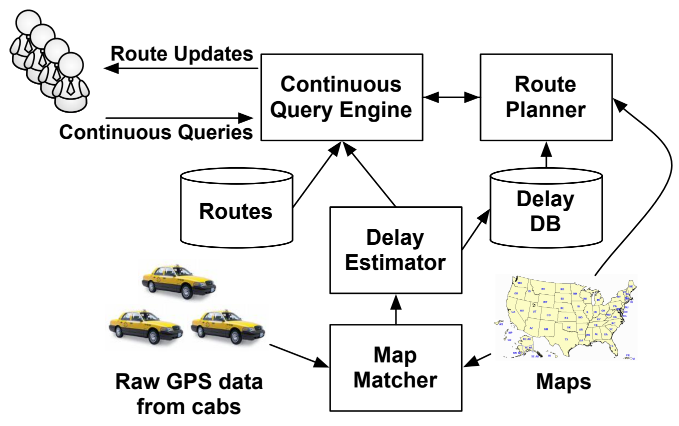

Fig. 1. The architecture of our system.

transportation planning systems is largely based on graph preprocessing under the assumption of static conditions to speed up query response times [17]. *Reach based routing* [18] improves query performance by adding shortcut edges to reduce nodes's *reaches* during preprocessing and though useful for routing in large road networks, it overlooks real time delays and is thus not applicable to our problem. *Landmark Indexing* [19] and *Transit Routing* [20] speed up run time query performance by using precomputed distances between certain set of landmarks (or transit nodes) chosen according to sophisticated algorithms. As with hierarchical schemes, distance between all landmark pairs is precomputed and is subsequently used unchanged, and so real time traffic delays cannot easily be taken into account.

There have been a number of recent studies in the area of pre-computation based shortest path calculation in the presence of edge weight changes. An adaptation of landmark-based routing to dynamic scenarios [21] yields good query times but requires that a link's cost not drop below its initial value. A dynamic variant [22] of *highway-node routing* gives fast response times but can handle only a small number of edge weight changes. A *bidirectional ALT* algorithm for dynamic time-dependent graphs [23] is fast but can again handle edge weight increases only (they focus on handling traffic jams in road networks). Our algorithms do not have these limitations.

Other related but orthogonal work on spatial databases and route planning in the database and algorithms communities include probabilistic path queries [24], dynamic k-NN [25], path indexing [26], path oracles and efficient storage [27] and trip planning [28]. Though some of these ideas, like path indexing, may be useful in our system when maintaining a very large number of paths between a single source and destination pair, none of them address the problem of building a scalable routing service which handles continuous routing queries from potentially hundreds of thousands of users and updates them on the fly as traffic updates stream in.

III. Architecture

The basic architecture of our system is shown in Figure 1.

Raw data in the form of GPS readings arrives from a fleet of about 30 taxi cabs we have deployed in the Boston area as a part of the the CarTel project [29]. This raw data is processed by a map-matching algorithm to identify the road segments that are traversed; a delay estimation algorithm computes average statistics about the expected time to traverse each segment by combining historical and real time delays. Delay estimates are stored in a delay database. We currently have about 20 million delay estimates (<time, segmentid> pairs) for about 90,000 road segments collected over the past 6 months. Our system uses underlying map (road connectivity) data from NAVTEQ [1] and OpenStreetMap.org. As our cars travel mainly in Boston, we largely focus on this part of the graph which has about 40,000 segments and 30,000 nodes. We have also tested our algorithms on the entire US network, consisting of about 83 million segments and 77 million nodes.

The route planner is responsible for finding the minimum cost route between two points on the road network based on current traffic delays. We use historical delay data for precomputation of candidate routes for continuous queries registered with system (this method is detailed in Section V-A). It is also used when the user fires an ad-hoc query (as opposed to registering a continuous routing query with the system) between a start and destination for <day, time> pair other than the current time. For example, an ad-hoc routing query for a Monday afternoon time would be answered using aggregated delay statistics for that Monday afternoon time from the system's accumulated history.

Currently our backend is used by the iCartel application for route planning on the continental US road network. Users can submit route planning requests by running the iPhone application and asking for a path from their current location to some destination. Our system reads new traffic delay data and updates the road delays every t units of time – the frequency of updates we use in the paper is 15 minutes, but as more and more users adopt the system, we expect the updates to be many more in number and their processing much more frequent.

Our goal in this paper is to describe the algorithms and design of the *continuous query engine*, which takes standing queries from users specifying the start point and the destination of their travel, and sends new routes to users whenever the delay on the best route for their query changes substantially, which our system captures using factors ϵ and γ.

IV. Preliminaries

In this section, we overview the A* search algorithm [5] used for finding the shortest path between two vertices in a graph, and Yen's algorithm [30] which finds the K shortest loopless paths between two given nodes in a network. We use both these algorithms as a basis for the techniques we discuss in the next section.

A. A* search

The idea underlying A* search is the same as Dijkstra's algorithm but it uses a *distance-plus-cost* heuristic function $f(x) = g(x)+h(x)$ to determine the order in which the search visits the graph nodes. Here $g(x)$ is the path-cost function which is the cost from the starting node to the current node and $h(x)$ is an admissible "heuristic estimate" of the distance to the goal,*i.e.* it must not overestimate the distance to the goal.

For route planning in road networks, we use the "as-the-crow-flies" distance at a maximum speed limit on a road segment to compute $h(x)$. We have found that this heuristic is very effective in practice; on our Boston subgraph, most A* route planning queries complete in about 100 ms. Our choice of A* as a search tool is because of its effectiveness and simplicity. Our ultimate aim is to show the scalability and effectiveness of our techniques regardless of the specific search algorithm used at the lowest layer.

B. Yen's Algorithm

If $\mathcal{P}$ is the set of all possible paths between the start node s and the target node t and the cost of a path p is given by $c(p)$, the K-shortest path problem is to determine a set of paths $\mathcal{P}_{\mathcal{K}} = \{p_1, \ldots, p_K\} \subseteq \mathcal{P}$ such that:

- $\forall\ k \in \{1, \ldots, K-1\}$, $c(p_k) \leq c(p_{k+1})$,
- For all simple paths $p \in \mathcal{P} - \mathcal{P}_{\mathcal{K}}$, $c(p_K) \leq c(p)$,
- For any $k \in \{1, \ldots, K-1\}$, p_k is determined before p_{k+1}

The most efficient solution for this problem is Eppstein's algorithm [31], but it can return paths with cycles which do not make sense in a road network. The significantly harder K-shortest *loopless* paths problem has the additional restriction that $\forall\ k \in \{1, \ldots, K\}$, p_k must be *simple* (should not contains any cycles). Yen's algorithm [30] solves this problem in $O(Kn(m+n\log n))$ time. Although more efficient techniques [32] exist, they do not work for all directed graphs and as Yen's bound is the best known for the general case, we use it in our work. Before describing the algorithm, we introduce some terminology as discussed in [33].

Yen's algorithm works as follows. Supposed we have already determined the k-th shortest loopless path $p_k = < v_1^k(=s), v_2^k, \ldots, v_{l_k}^k(=t) >$. To compute p_{k+1}, candidate shortest loopless paths which deviate from p_k in v_i^k are computed for every node v_i^k to be analyzed. The loopless property of paths is ensured by temporarily removing nodes on the subpath $\text{sub}_{p_k}(s, v_{i-1}^k)$ before the shortest loopless path from v_i^k to t is determined. The outgoing edge from v_i^k to v_{i+1}^k is also temporarily removed prior to this computation to avoid generation of a duplicate candidate path. Thus, in the worst case, the algorithm performs $O(n)$ A* shortest path computations for each of the K output paths.

V. Efficiently Answering Continuous Route Planning Queries

In this section, we describe our techniques for online maintenance of near-optimal routes for <src,dst> pairs requested by a set of continuous queries. We explore two different classes of algorithms:

- Approaches which involve a mix of precomputation of *K candidate paths* and on-the-fly cost recalculation
- A *proximity based approach* which computes an ellipse surrounding the previously reported optimal route

We describe these two approaches in more detail in subsections to follow; a comparison of the practical performance of the two algorithms along with conditions in which either should be selected over the other appears in Section VI-C. First we assume that our system has a total of r standing route planning queries (specifying <*origin, destination*> tuples) which it must continuously answer.

A. K Candidate paths

The motivating idea behind this approach is to find K different routes between a <src,dst> pair such that at least one of them is always *good* in the sense that change in traffic delays do not adversely affect all K routes simultaneously. Note that we do *not* store K candidate paths for each pair of nodes in the road network but only for the src-dst pairs registered as a part of some continuous routing query. Thus, the *storage overhead* per query of the K-candidate-paths approach is $O(K)$, with the hidden constant accounting for the space required to store the segments/nodes on each path.

As the queries registered by most users require continuous monitoring, we use overall historical delay statistics to compute and store K paths for each such continuous routing query. When registering a continuous routing query, the user is asked to specify only the $<$ origin, dst $>$ pair for the query (if the user also specifies a day and time range of travel along with the query, we store an array with a set of K paths per requested time slot), the different alternate paths are computed and stored internally by the system without any user intervention.

We aim to avoid run time shortest route recalculation by first precomputing these K candidate routes for all r pairs and then doing a simple re-rank among them on arrival of traffic delay updates. Re-ranking involves simply summing the delays along each segment of all precomputed paths and finding the route with least overall delay.

We do not re-rank candidate routes for a <src, dst> pair on arrival of every delay update. The decision to do so is based on both the number and degree of cost changes; we rerank as soon as one of the following events occurs:

1) More than a fraction ϵ of segments on any of these K paths has updated delays. This is because even small delay updates per edge could change the overall delay of the route substantially if there are a large number of edges with updated weights.
2) At least one of the edges has a real time delay more than γ times the previous delay (or if the previously reported delay is γ times greater than the delay on any of the K-candidate paths). If an edge has not had a real-time update recently, we compare the new cost to its historical delay. Real-world scenarios such as traffic jams and sudden blocking of streets (and their subsequent clearing up) can be modeled by high values of γ, irrespective of whether it is an increase or a decrease.

After a re-rank, we check to see if the current best path is different from the previously reported best path for each query and pro-actively inform the user if that is the case.

A simple route recomputation strategy is to use historical delay statistics to find the K least cost loopless paths by employing Yen's algorithm. However our experiments indicate that for road networks, the K paths returned by Yen's algorithm have overwhelmingly large number of segments in common, with just 1-2% of the segments differing from each other. Thus a potential traffic congestion is likely to affect all K paths whereas we desire that the set of candidate routes be more robust to real time changes in traffic delays.

Known effective methods for finding spatially dissimilar paths between two nodes in a transportation network include the Minimax method [34] and p-dispersion [35] but these heuristics suffer from a high computation cost which is undesirable for our setting where precomputation overhead for generating the K candidate routes must be kept small so that the system is able to register new continuous routing queries quickly.

Below we describe three practically fast techniques we have devised for computing a good set of K paths between two nodes. We remark that the real-time travel costs on these paths are not guaranteed to be the best-possible (optimal or near optimal) because the link costs change as traffic delays change in real time, thus the various K-candidate-routes algorithms we propose are approximate. We however experimentally show the near-optimality of these approaches and the proximity based technique in Section VI-B3.

1) K-AS-VARIANCE : Perturbing the edge costs: The first idea for computing K different paths is to iteratively run the A* search algorithm on the underlying graph, each time starting from the source until we find the destination node – but with graph edges having slightly different weights in different iterations. Our idea is to maintain a Gaussian distribution (mean μ and variance σ^2) for the historical delay observed for each edge during each time slot, and to sample the delay from this distribution. We call this technique K-AS-VARIANCE.

We perform edge cost computation *on-the-fly i.e.* if a segment is encountered while computing the shortest route, we re-sample the cost. This idea avoids wasting time in setting all edge costs to a new value at the beginning of an iteration. Note that it is possible that despite different edge costs, two iterations still return the same shortest path between the start and end nodes under consideration. To address this issue, we run the algorithm until K distinct paths are found.

We observe that this technique fares better when segments have high historical delay variances, as otherwise successive iterations will return nearly identical paths. Although a large number of iterations may eventually return K different paths, they are likely to differ only in a few segments and unlikely to be a large improvement over Yen's algorithm. For this reason, we terminate the algorithm after a fixed number of iterations even if it has not found K distinct paths yet. However, experimentally we observe that for $K = 5$, such a forced termination happens only for a small fraction of src-dst pairs.

This approach requires $\Omega(K)$ A* searches to find the K candidate routes; we bound the number of searches above by $O(K^2)$ (more details on the precomputation time of this approach in Section VI-B5).

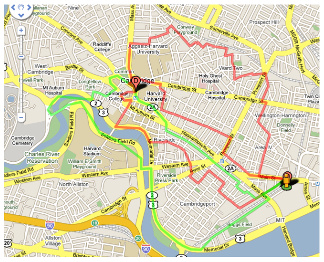
Fig. 2. $K = 5$ different paths between MIT and Harvard computed by our aggressive strategy

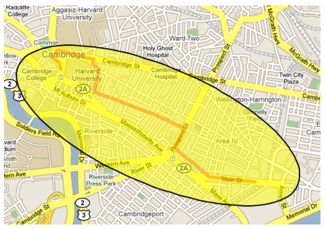
Fig. 3. The computed ellipse for the shown route between MIT and Harvard

2) K-AS-AGGRESSIVE : Nearly Disjoint paths: The similarity of paths returned by Yen's algorithm as well as by our K-AS-VARIANCE technique when variances in link costs are low motivated us to develop a technique which is guaranteed to return a set of K paths which substantially differ from each other. In particular, our goal is to find paths that have minimal (or zero) overlapping edges so that the paths are **nearly disjoint**. We observe that such a K path set represents K alternate ways of reaching one's destination starting from the source and this makes it likely that despite traffic congestions in parts of a city, some of these K paths will be uncongested and easily navigable.

We achieve near-disjointness by *aggressively* deleting all edges on a shortest path found in any previous iteration from the graph (such deletions are temporary and the edges are restored after the algorithm's run). This ensures that the $(l + 1)$th shortest path p_{l+1} from source to the destination has different edges from those on paths $\{p_1, \ldots, p_l\}$. Finding K completely disjoint paths however may be impossible, it could be that the degree of either node is smaller than K or that fewer than K bridges connect s and t in the graph. For this reason, we relax our disjointness condition by allowing up to n_d edges near the start and end nodes to be repeated between all the K paths. Observe that the algorithm does at most K A* searches. The time spent in deleting the edges from the previous iteration's output path can be subsumed within the time required for the A* search, so that the overall precomputation time is still of the order of K A* searches.

Note that the sole purpose of having the parameter $n_d (> 0)$ is to allow for edges very close to the start and end nodes to be shared between different candidate paths so that K different nearly disjoint paths can actually be found. We empirically set $n_d = min(K, 5)$. Because n_d is small compared to the typical number of segments on a route, its selection is *not* critical to finding the best possible disjoint paths. We've found that small variations in its value do not alter the returned routes.

In Figure 2, we show $K = 5$ different paths returned by K-AS-Aggressive when the origin node is MIT, Cambridge and the destination is Harvard University. Note that three paths are shown in red overlays and the other two in green. Though the red colored path to the right seems convoluted, our delay statistics show that its cost is not significantly higher compared to the other paths.

3) Variants of Yen's Algorithm: Although the *aggressive* strategy usually identifies a *good* set of K paths, there are two motivating reasons to explore other possible techniques: (1) complete disjointness of paths might exclude many routes which are otherwise attractive, e.g., a route with a quick detour from somewhere close to the middle of a returned "aggressive path"; (2) real users often prefer routes which appear less convoluted when seen on a map. In particular, we believe drivers may shy away from such convoluted routes even if the estimated travel times are low (our system does include *turn penalties* to prefer routes with fewer turns, but such routes can still arise on very congested paths.). To address this concern, we explored several variants of Yen's algorithm that return a set of good set of K-paths without resorting to the extreme disjointedness of the K-AS-AGGRESSIVE algorithm. Both the techniques described below have a worst case computational complexity of $O(Kn(m+n\log n))$, being derivatives of Yen's algorithm. However, we found their performance to be much faster in practice (see Section VI-B5 for average precomputation times).

We call our first variant Y-MODERATE. In Yen's algorithm, when candidate paths for p_{l+1} are being generated from the l-th shortest route p_l, Y-MODERATE does not permit candidates with more than a fraction $1/f$ of common prefix nodes with p_l. Thus, if $len(p_k)+1$ is the number of nodes on the path p_l, no candidate generated in the lth iteration is allowed to share more than $\frac{1}{f}(len(p_l)+1)$ prefix nodes with p_l. We set $f=2$ for our implementation. It may appear that setting f to large values simulates K-AS-AGGRESSIVE since for large f, $1/f \approx 0$. However this is not necessarily true because Yen's algorithm does not guarantee disjoint paths beyond the deviation nodes, so we may end up with K paths which have many nodes common close to the destination node. Our experiments show that the *moderate* variant of Yen's algorithm is better than a direct application of Yen's algorithm, but that the K routes it returns still tend to be similar.

Thus, Y-MODERATE has a few problems similar to what motivated us to define the K-AS-AGGRESSIVE algorithm in the first place. To achieve our goal of identifying K-path sets which are robust to real time traffic delay changes, we define the Y-STATISTICAL strategy which is a combination of the ideas of the K-AS-AGGRESSIVE technique ideas with the K loopless paths computation of Yen's algorithm. The idea is to remove some, but *not all* edges of the path found in the previous iteration. Suppose l-th shortest path p_l has already been found. Then, before generating new paths for p_{l+1}, we pick each segment on p_l and temporarily delete it from the graph with a certain probability. We experimented with several ways to choose this probability of deletion, including deleting edges that occur in a large fraction of routes, or deleting edges that are traversed more frequently with a higher probability. We found, however, that randomly deleting an edge with a fixed probability **Pr** (**Pr** = 1/2 in our experiments) worked as well as the above heuristics for $K=5$, so we use that strategy in our reported results.

Similar to the K-AS-Aggressive algorithm, Y-Statistical computes K paths that are significantly different from each other but with a smaller fraction of 'weird' routes.

B. Proximity Measures

Unlike the algorithms in the previous section which seek a good set of precomputed paths, our proximity-based approach attempts to devise a *proximity measure* that triggers a recomputation of the shortest route at run time only when there are "sufficiently large" number of segments with new delays that are geographically near to the previously reported fastest route (which is initially the precomputed historically fastest route). By using a *region-of-influence* based heuristic technique, we maintain *only one* fastest route between every source-destination pair at all times.

Intuitively a region of influence around an object is a closed curve like a circle or an ellipse. Different shapes could work better for certain situations. However, computing arbitrary shapes is much more complex, testing whether a segment falls within an arbitrary (not necessarily convex) shape is also computationally expensive. Furthermore, arbitrary curves need more parameters to define them, resulting in a variable as well as higher storage space per route. If we assume the region of influence of any path p between s and t to be an ellipse that encloses the entire path, we require five parameters (details below). A simpler shape such as a rotated rectangle would also require the same number of parameters. The advantage of using an ellipse is that it restricts the number of segments that need to be considered while processing updates that we might otherwise consider unnecessarily.

The more the number of segments with updated delays within the path's enclosing ellipse, the higher the chance that the previously reported shortest route is no longer optimal – thereby requiring a recomputation at run time. Our technique differs from similar ideas in [36] in several ways: (1) we use the proximity region to evaluate whether or not the best route needs to be recomputed as opposed to using it to constraint the shortest path search space; (2) our region of influence is an ellipse instead of a MBR of an ellipse; and (3) the algorithm for computing the enclosing ellipse is completely different (detailed in the following section).

We remark that a recomputation of the optimal route is triggered only when the number of segments inside the ellipse with updated delays exceeds a fraction ϵ of the ellipse's area times the average number of segments found in a unit area of the road network.

1) Computing the enclosing ellipse: Finding an ellipse which encloses a set of points in two dimensional space (every path can be seen as an ordered list of latitude-longitude pairs) is a computational geometry problem. Known solutions [37] are designed to compute approximate hyper-ellipsoids for arbitrarily high dimensional Euclidean spaces and thus despite being linear in the number of input points, they have a high computational cost in practice.

Instead we have developed a simple but fast technique for approximate computation of the enclosing ellipse, which requires just two linear scans of the input set. The pseudocode appears in Algorithm 1.

Algorithm 1 Pseudocode for finding the enclosing ellipse of a path p.

```
1:  procedure FAST-ENCLOSING-ELLIPSE($x_s$, $y_s$, $x_e$, $y_e$, $p$, $f_r$)
2:      $(x_o, y_o) \leftarrow ((x_s + x_e)/2, (y_s + y_e)/2)$;
3:      $a \leftarrow f_r$*crowDistance($x_s, y_s, x_e, y_e$)/2;
4:                                  ▷ Compute equation of $L_1$ and $L_2$
5:          ▷ Compute maximum perpendicular distance $d_{max}$ to $L_1$
6:      $b \leftarrow d_{max}$;
7:      for all vertices $v$ on $p$ do
8:          if ($v$ is not inside ellipse) then
9:   ▷ Assume $v$ to be on the ellipse and use the ellipse's equation to compute the new value of $b$
10:         end if
11:     end for
12:     if ($b < a$) then
13:         $a = \sqrt{b^2 + a^2}$;
14:         $L_{major} = L_1$
15:     else
16:         $L_{major} = L_2$
17:     end if
        return $\{x_o, y_o, a, b, slope(L_{major})\}$;
18: end procedure
```

An ellipse is characterized by two parameters: the semi-major axis length a and the semi-minor axis length b. The two foci of the ellipse lie on the major axis, with their position determined by a, b, the orientation (or slope) of the major axis L_{major} and the position of the ellipse's center on the major axis. The line L_{minor} containing the minor axis is always perpendicular to L_{major}.

Our algorithm proceeds as follows: we first assume the center (x_o, y_o) of the ellipse to be the midpoint of the two node coordinates (x_s, y_s) and (x_e, y_e) and compute the equation of line L_1 joining them and also of the line L_2 perpendicular to L_1 and passing through (x_o, y_o). We assume L_1 to be L_{major} and set the parameter a to $f_r \frac{d_{s,t}}{2}$. Here $d_{s,t}$ is the "as-the-crow-flies-distance" between (x_s, y_s) and (x_e, y_e) and $f_r (> 1)$ is a small relaxation factor to ensure that nodes do not lie beyond either end of the axis along L_1. Note that this may not turn out to be the length of the semi-major axis as we adjust a in later steps.

We initially set b (half the length of the other axis) to the maximum among the shortest distances of all nodes on the precomputed path from L_1. We iteratively refine the value of b by processing all vertices on the shortest path in increasing order of their distances from (x_s, y_s) and (x_e, y_e) *along* L_1. If a vertex v does not lie inside the ellipse determined by the current value of a and b, we assume that v lies *on* the ellipse and compute the new (larger) value of b using the equation of the ellipse (we have already set a to a temporary value). This particular order of processing these nodes reduces the number of recomputations of b by ensuring that outlier nodes far from the center (along L_1) are encountered early in the algorithm's run so that after the first few iterations, the ellipse already has most of the points to be processed in the remaining iterations. Finally, if it turns out that $b > a$ at the end of the computation, we let L_2 be the major axis instead and b be the semi-major axis length. Otherwise, L_1 still remains the major axis and the value of a is increased to reflect that the initial value of a captured half the distance between the two foci, so that now: $a_{final} = \sqrt{b^2 + a_{init}^2}$. We can now identify our enclosing ellipse uniquely using four parameters which we have already computed: center of the ellipse, a, b, slope of the major axis (L_{major}).

An example ellipse for the shortest route between MIT and Harvard appears in Figure 3. As our algorithm is a fast heuristic, it has some shortcomings; in particular, our assumption that the line joining the two nodes is one of the axes may sometimes result in the ellipse becoming larger than it needs to be. However, we experimentally observe that with appropriate choice of parameters, the approach is still quite efficient.

2) Precomputation Complexity: The per query storage complexity of the proximity approach is $O(1)$ because we simply need to store the best path per query, and the enclosing ellipse can be stored by simply storing five parameters described in the pseudocode: $\{x_o, y_o, a, b, slope(L_{major})\}$ (see Algorithm 1).

The proximity approach requires only one A* search to precompute the optimal route between a src-dst pair and time taken to compute the ellipse is of the order of the number of edges on the best route, which is far below the time an A* search takes, so that the overall time complexity in terms of the number of A* searches is $O(1)$.

3) Processing real time delay updates: On arrival of a batch of delay updates, we check if the number of segments with delay updates that lie inside the pre-computed ellipse for a routing query exceeds ϵ times the average number of segments lying inside an ellipse of this area. If so, we re-run A* search and return a real-time optimal path to the end user. At the same time, we also recompute and store the bounding ellipse corresponding to the new path.

For r registered queries and a batch update size of u, the worst case processing time of our algorithm is $O(r(u + (m + n \log n)))$. In practice, the A* search is re-run only for a small fraction of queries and the algorithm runs much faster.

VI. Evaluation

In this section, we experimentally evaluate the techniques we described in Section V. In addition to the methods above, we also consider a *brute force* method that re-runs a complete A* search any time a batch of delay updates arrive (in our system delays arrive every 15 minutes). Specifically our goals were to (1) gauge how well our techniques perform in comparison to the naive brute-force strategy for recalculation of the optimal route (using A* search) in terms of both the time taken and the number of updates sent to the user, (2) quantify (using expected travel times) how much worse the routes our system suggests turn out to be in comparison to the brute force method, and (3) see how the choice of ϵ and γ affect performance. We found that all of our algorithms outperform brute-force on metric 1 and are about 5–7% worse on metric 2. We also observed that our algorithms scale well with the number of registered routes, suggesting that our algorithm is practical.

A. *Dataset and Experimental Setup*

We evaluate our system using a database of actual commutes, gathered from our network of 30+ taxi cabs in Boston. We chose the set of designated routes to be monitored by randomly picking 7000 <start, end> pairs from our CarTel log containing approximately 150,000 real drives. For our experiments, we used the actual real time traffic delays on road segments – computed by the match matching and delay estimation components (see Figure 1).

All of our experiments were run on a 2.66 GHz 8-core Intel Xeon workstation with a memory of 16 GB and running Fedora Linux 10. The underlying NAVTEQ map data and the set of designated source-destination pairs were stored in a MySQL database on a separate Intel Core 2 Duo 2.66 GHz machine with 4 GB of RAM. We used [38] as a base implementation of the actual Yen's algorithm and modified it substantially to suit the requirements of our work (and to interface with other parts of our code). The entire road network as well as queries and cached routes fit into memory. We report on the total state required to store precomputed routes later in this section.

We ran our system by initializing it with a number of continuous routing queries (up to 7000). We then simulated updates to the real-time delays on the road network by replaying data from two months of our traffic delay database, with delay updates arriving in 15 minute batches.

B. *Results*

Below we present results for the two classes of techniques we have proposed: the proximity approach and the K-candidate-paths approach for $K = 5$. We found that higher values of K simply increased pre-computation overhead with little added benefit in route quality. In all experiments below, we set epsilon to $\epsilon = 0.25$ and $\gamma = 1.75$; we evaluated the benefit of variable values of ϵ and γ in separate experiments.

1) Experiment 1: Runtime Performance: As real time traffic delays stream in, each K-path technique checks if more than a fraction ϵ of the segments on any of the K-paths for a designated pair have new delay values or if there is a segment of the current best path with more than a factor of γ increase/decrease in its cost and re-computes the best path ("re-ranks") for each such pair. Similarly, the proximity measure recomputes the best path if the number of segments with new delay updates lying inside a pair's ellipse (of area A) exceeds ϵ times the average number of segments in an ellipse of area A or if there is a segment on the current optimal route with a greater than γ increase/decrease in link cost.

Figure 4 shows the runtime performance of our methods vs. the number of registered routes. Note that the y-axis is log-scale. Here, our K-path techniques beat the brute force strategy by up to two orders of magnitude and the proximity approach is at least an order of magnitude faster. The higher runtime cost of the proximity approach is due to its use of A* path computations when an update is required. In contrast, the K-best-paths approaches have low runtime cost because they simply compare the costs of the K paths.

A key observation from this plot is that the K-paths scale very well with the number of registered routes. This suggests that our algorithms are well suited to our continuous route planning scenario.

2) Experiment 2: Fraction of routes updated: In this section, we measure the fraction of updated routes for our different techniques, as ϵ and γ vary.

Figure 5 shows the average fraction of routes updated by each of our methods for $\epsilon = 0.25$ and $\gamma = 1.75$ when a new batch of delay estimates arrives (each batch consists of 3000 real time updated segments on average). The fraction of updated routes grows with increasing value of r (the total number of continuous routing queries registered with the system), which is expected because an increased size and more diverse set of designated <src, dst> pairs means that the probability of delay updates affecting at least one such pair is now higher. We see that compared to a brute force method, all of our strategies except K-AS-Aggressive update only about 15-30% routes, a substantial reduction. The performance of K-AS-Aggressive technique is also good, with 20-40% updates required. The aggressive technique requires more updates because its precomputed routes are more diverse, so updates have a higher probability of affecting at least one of a set of K nearly disjoint paths. We note that for the K-best-paths approaches, the time to perform a re-ranking is much less than the time to perform a full A* search (as described above), and that overall the runtime of K-AS-Aggressive is not substantially worse than the other K-best-paths approaches.

The number of updates issued is clearly a function of the value of ϵ. This is because as we vary ϵ from a little over zero to higher values (close to 1), fewer queries have a high fraction ($\geq \epsilon$) of segments on one of their K-paths updated (or a high number of segments with updated delays inside their best path's enclosing ellipse for the proximity approach). Figure 6 confirms this intuition, showing that both strategies see a drop in the fraction of updated routes with increasing ϵ. The *proximity measure* results in a higher number of updates; this is because the enclosing ellipse for a route encloses many more road segments than the total segments on the K paths and is thus more sensitive to traffic delay updates. We disable route updates resulting due to new link delays greater than the factor γ for Figure 6 so as to avoid route updates due to both ϵ and γ from masking the effect of change in ϵ on the number of routes updated.

Figure 7 shows the fraction of route updates for increasing values of γ with no re-ranks due to fraction ϵ (to independently observe the dependence of route updates on γ). As expected, we see a rapid fall in route updates as γ varies from slightly over 1 till 5. Invariably, there are a number segments with delay updates within 5-10% of the previous real time cost (or the historical link cost in absence of updates in the recent

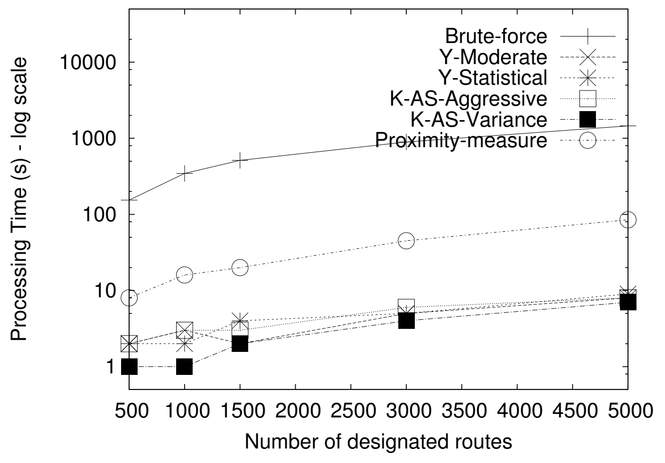

Fig. 4. Runtime processing cost (log scale) vs the number of paths

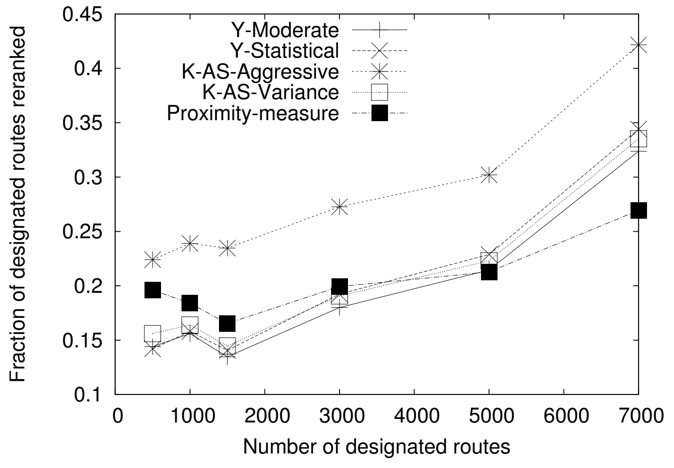

Fig. 5. Fraction of routes updated vs the total number of routes

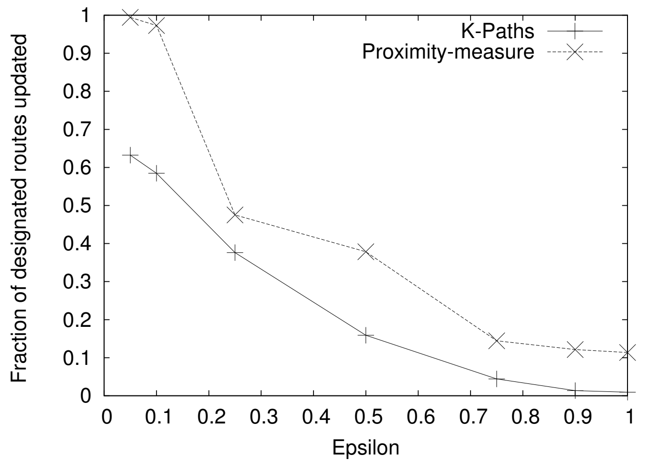

Fig. 6. Number of routes updated vs. ϵ

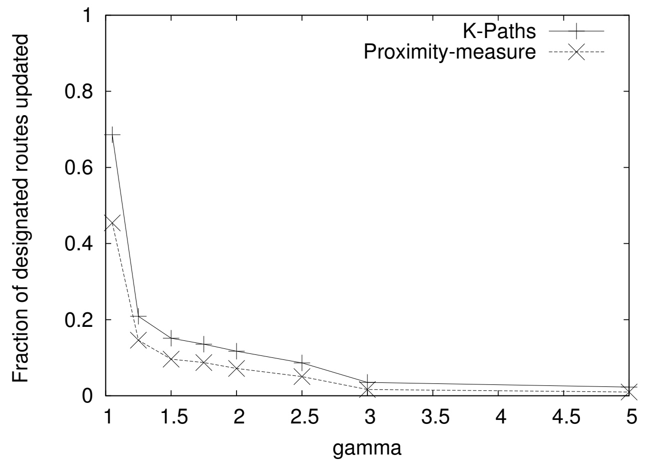

Fig. 7. Number of routes updated vs. γ

past), leading to a huge number of reranks for $\gamma = 1.05$ Few segments tend to have real time cost changes by a factor greater than γ values and even fewer lie on one of the pre-calculated routes for each query, explaining the rapid fall in number of route updates with increasing γ. This time the proximity approach does fewer updates, because only a drastic update on only one path (the previously reported route) can lead to updation of the best route for a query whereas for the K-Paths techniques, there are K such paths. Note that even for road segments with routine congestion during peak traffic hours, cost update factors greater than γ are few for one of two reasons: (1) a segment's historical delay value for a given day and time already captures that the route is routinely congested, (2) we measure change with respect to the most recent real-time delay (if available) and the update reflecting the congested value was processed in the previous batch of updates.

3) Experiment 3: Optimality: In addition to speed and scalability, the practical utility of our system depends on routes being "reasonably accurate", *i.e.* their cost should be close to the optimal value such that end users are not misled. We measure optimality as follows: suppose c_{opt} is the true cost (as returned by a *brute force* A* search) and the cost of the route returned by our system is c. We define the **optimality ratio** $OR = \frac{c - c_{opt}}{c_{opt}}$ to be the quality of our routes. Note that $c \geq c_{opt}$, meaning the cost estimate can never be an underestimate as c_{opt} is (by definition) the least cost path. Figure 8 shows how OR changes vs. ϵ for our different techniques (with no pruning based on γ). In this experiment, we measure optimality only when our system recomputes a new best route.

The proximity approach recomputes the best shortest path every time it issues an update and therefore is always optimal in its suggested cost. We see that for all ϵ (ranging from 0.05 to 1), the costs *reported* by all our K-path techniques are always within 10% of the optimal cost (e.g., on a 30 minute route, the travel time of the reported route is within 30 to 33 minutes). As ϵ grows larger, the suggested routes have costs closer to the true cost. This is because for small ϵ, when a recomputation does happen, our algorithms – especially those that find very different paths are likely to return the same route since changes in the delays of a few segments are unlikely to affect the overall best path. For larger values of ϵ, more segments will have changed and our algorithms are more likely to return a different path, which would be closer to the true optimal. This also explains why Y-Moderate and K-AS-Variance fare better as both of them have a set of K paths which is similar and so are more responsive to small changes in delays of a few segments.

There are many routes for which the K-path and proximity approaches do not report an update because neither at least ϵ fraction of segments are updated, nor there is any segment on the best path with a substantially different link cost (as measured by γ). In such a case, the user must assume that the previously reported (or precomputed) shortest route is still the best. Let c_{act} be the cost of this previous route with new delays for some <src,dst> pair. In this case, we define the optimality ratio as $OR = \frac{c_{act} - c_{opt}}{c_{opt}}$. We show the variation of OR with ϵ for such non-updated pairs in Figure 9. The optimality gap falls for larger values of ϵ because routes are now re-ranked for a smaller fraction of queries, and re-ranked routes naturally have overall delay values closer to the optimal (with OR $\rightarrow 0$ as $\epsilon \rightarrow 1$). Here the optimality gap for the proximity measure is the highest around 10% for all ϵ, which suggests that using the average number of segments for an ellipse of a given area does not accurately capture the actual probability of the optimal route changing. The curves for K-path techniques are more promising. They show that for low ϵ values, the routes we report are relatively close to the optimal, with errors less than 5% for ϵ values up to 0.2.

Figures 8 and 9 show opposite trends for the K-candidate-paths methods; for that reason, ϵ values in the range 0.2-0.5 appear to be the best choice, offering errors of 3–7% from the optimal for both updated as well as non-updated routes. We see that a smaller value of $\epsilon = 0.25$ is better for K-AS-Variance and Y-Moderate and $\epsilon = 0.5$ is a more prudent choice if K-AS-Aggressive or Y-Statistical strategies are being used.

Figure 10 shows variation of OR of updated pairs with change in γ (route updates due to ϵ are not considered in this plot). The routes suggested by the proximity approach are always optimal, so OR= 0 for all updated pairs. For the K-candidates approach, the OR grows worse on average as γ increases which is expected because sudden drastic changes in link cost may not always be captured by the precomputed K-paths. For non-updated pairs, the OR values remain nearly constant around 10% as γ varies (plot omitted due to lack of space). Thus, for all techniques, errors from the optimal are within 10% for γ values in the range $1 - 2$. Keeping Figure 7 in mind, we see that a γ value in the range 1.5–2 is however a more prudent choice.

Route updates when done using both ϵ and γ for appropriate choice of these parameters as discussed above ensures that we are within 5% of the optimal cost for the K-candidates approaches, and within 10% of the optimal cost for the proximity approach.

One concern with the above results might be that *any* route would provide relatively low OR values. To invalidate this hypothesis, we computed the best route recommended by each of our approaches at the beginning of our simulation (when the delays in the road network were computed only from historical averages), and measured the average OR of that route after the simulation had completed. We found that the average OR value was over 20% in all cases, and often as high as 80%, suggesting that the OR values of 3–7% from our system are quite good.

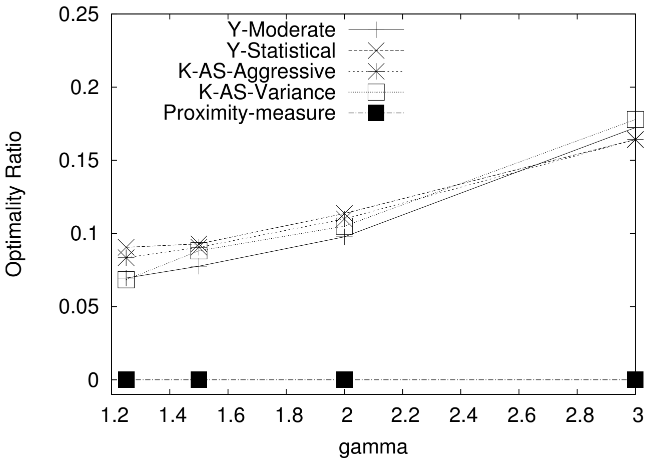

Fig. 10. Variation of optimality ratio OR with γ for suggested routes

4) Experiment 4: Day and Slot Pruning: Continuous monitoring of all routes has the advantage that the user is always aware of the best route. However, if the user is interested in only updates for specific times on chosen days ("slots"), our system allows explicit specification of the day and time of travel while registering the continuous routing query. We take advantage of this by checking if the slot the user has asked the system to monitor will be affected by newly arriving delay updates. The system can simply provide continuous monitoring for a few hours prior to the specified slot (we used a value of 1 hour in our experiments).

We measured the effect of this and found that this pruning is very effective in the sense that if all registered continuous queries contain <day, time> details, the system's load reduces by at least an order of magnitude (a factor of 20 when each routing request is for a randomly selected hour of the day on all days of the week).

5) Experiment 5: Pre-Computation Cost: Both the K-best-paths and proximity techniques employ precomputation (as described in section V). Though the overhead incurred in doing so is offline, it is nevertheless important to make sure that this cost is reasonably small so that the precomputation for all designated routes is practical. Aggregated historical delays are likely to change over time (even if not very significantly), necessitating a periodic re-computation of pre-computed routes to keep them updated with the current traffic delays.

Table 11 shows the average cost incurred per designated <start, end> pair for our different techniques (in seconds). For any designated pair, the *proximity measure* requires one A* search to compute the shortest route and a single run of the linear time enclosing ellipse algorithm. We thus expect it to be extremely fast, as shown. Similarly, one would expect that as the *K-AS-aggressive* strategy requires just K(=5) A* computations, its precomputation cost should be about K times slower than the proximity approach. Surprisingly, the experimentally observed value is much higher. This is due to the fact that for some src-dst pairs, deletion of the edges on the previous best paths leads to a rather expensive A* search to find the next best route. The extremely high variance for *K-AS-*

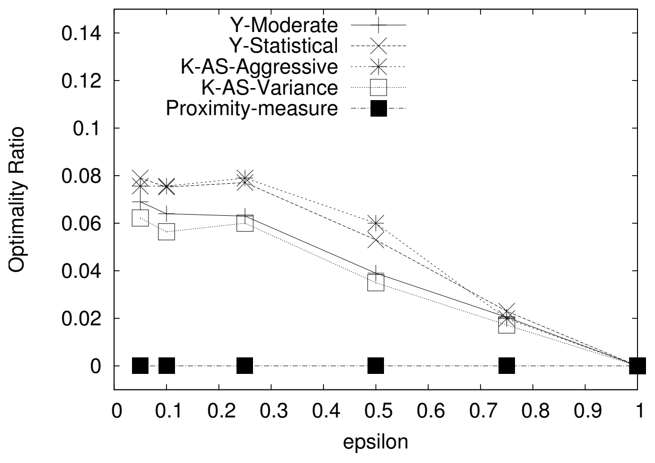

Fig. 8. Variation of optimality ratio OR with ϵ for suggested routes

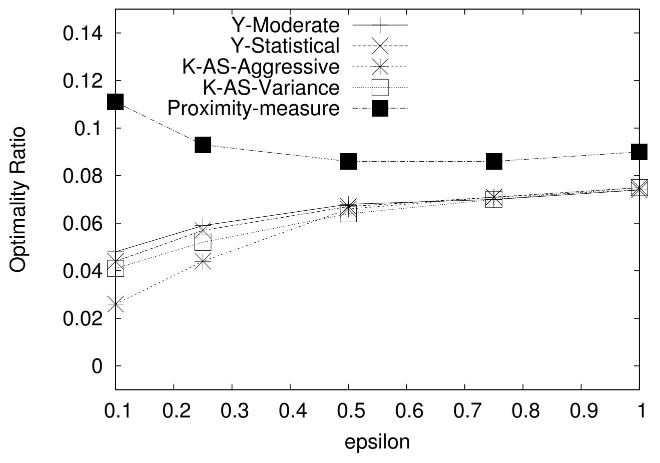

Fig. 9. Optimality ratio vs ϵ for non-updated pairs

aggressive supports our argument that expensive A* searches for select <src,dst> pairs pull the overall average up. For the *K-AS-Variance* technique, we expect that the time required to find all the K-paths could be much higher than than for K iterations as several iterations in the technique's run return the same best path (which must be checked against all existing paths to be certain there's no exact match). We experimentally observed that on an average K^2 iterations are required for this.

For the techniques based on Yen's algorithm, we require a total Kn iterations in the worst case (as discussed in section IV), which can be very expensive owing to large $n(= |V|)$ for a road network. However, A* search's pruning ensures that the number of such iterations is small in practice and is confirmed by the observed average running times. *Y-statistical* is slow for the same reason as the K-AS-Aggressive technique. Overall, however, the Yen's algorithm-based techniques are practical despite their theoretically unimpressive bound of $O(Kn(m + n \log n))$.

6) Space Overhead due to Precomputation: A summary of the space costs of the precomputed routes for the different approaches per continuous query appear in Table 12 (values reported are for $K = 5$).

Because the proximity approach stores only the best path and the parameters of the ellipse associated with it, its storage overhead is the smallest. The K-candidates routes approaches store K routes for each registered query - so as expected, they have approximately K times larger storage cost. The K-AS-Aggressive and Y-Statistical approaches have a relatively high storage overhead compared to K-AS-Variance and Y-Moderate because they store a more varied set of routes for each query, so that some of the K candidate routes computed by them contain a larger number of edges in comparison to their other approaches' counterpart routes which are more similar to the most optimal historical route and thus are likely to contain fewer edges. We note, however, that in all cases the overhead is small enough to store a million or more registered routes in RAM.

C. Discussion

Our experiments show that both K-candidate-paths and proximity approach are efficient in their run time processing of traffic delays and are able to maintain routes with near-optimal costs. Figure 4 shows that the K-candidate-paths techniques are more scalable than the proximity approach. Additionally, their run times are an order of magnitude faster than the proximity approach. Furthermore, the optimality gaps of the K-candidate-paths strategies are about a factor of two better than that of the proximity technique. In particular, *K-AS-Variance* and *Y-Moderate* give better cost guarantees for small ϵ values on reported updates, while the *K-AS-Aggressive* and *Y-Statistical* strategies do a better job of reporting a route which stays close to optimal for small ϵ values when no route updates are performed. For this reason, we prefer the use of K-candidate-paths approaches in general, with the *K-AS-Variance* and *Y-Moderate* approaches slightly preferred due to their lower precomputation times.

The main advantage of the proximity approach is its extremely low precomputation overhead (see Table 11), which could be important in settings where new users register routes very frequently. The K-candidate-path approaches also have a K times higher space overhead, which could be an issue in which memory is constrained.

VII. Conclusions

In this paper, we described scalable techniques for continuous route planning queries on a road network. We explored two classes of algorithms: a *proximity-based* algorithm that recomputes the optimal route when more than some fraction of road delays change within a bounding ellipse, and several *K-candidate-paths* algorithms that compute a set of K possible routes and periodically re-evaluate the best route as road delays change. Our results on 7,000 drives from a network of taxi cabs show that these algorithms are one-to-two orders of magnitude faster than a naive scheme which recomputes routes using A* search whenever a new set of delays arrive, and only 5–7% worse in terms of travel time of the returned route. We also showed that our algorithms scale sub-linearly with the number of registered routes, suggesting their suitability for a large scale deployment. Finally, we have integrated these algorithms into a complete route planning system that processes data in real time and anticipate their inclusion in our iCarTel iPhone application.

Technique	Avg (s)	Var
Proximity (Ellipse)	0.10	0.09
K-AS-Aggressive	4.49	119.98
K-AS-Variance	2.58	11.34
Y-Statistical	5.69	142.25
Y-Moderate	2.15	25.18

Fig. 11. Average pre-computation times in seconds for a continuous routing query (designated <src,dst> pair)

Technique	Average space (bytes)
Proximity (Ellipse)	485
K-AS-Aggressive	3015
K-AS-Variance	2570
Y-Statistical	2830
Y-Moderate	2363

Fig. 12. Average storage overhead (due to precomputation) in bytes per continuous routing query for different approaches ($K = 5$)

References

[1] "NAVTEQ map data," http://corporate.navteq.com/database.html.

[2] C. Demetrescu, S. Emiliozzi, and G. F. Italiano, "Experimental analysis of dynamic all pairs shortest path algorithms," in *SODA*, 2004, pp. 369–378.

[3] U. Zwick, "Exact and approximate distances in graphs - a survey," in *9th Annual European Symposium on Algorithms*. London, UK: Springer-Verlag, 2001, pp. 33–48.

[4] A. V. Goldberg and C. Silverstein, "Implementations of dijkstra's algorithm based on multi-level buckets," in *Lecture Notes in Economics and Mathematical Systems, Vol. 450*, P. Pardalos, D. Hearn, and W. Hages, Eds. Springer, 1997, pp. 292–327.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, February 2007. [Online]. Available: http://dx.doi.org/10.1109/TSSC.1968.300136

[6] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime dynamic A*: An anytime, replanning algorithm," *International Conference on Automated Planning and Scheduling*, 2005.

[7] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning a*," *Artif. Intell.*, vol. 155, no. 1-2, pp. 93–146, 2004.

[8] G. Ramalingam, *Bounded Incremental Computation*, J. v. Leeuwen, J. Hartmanis, and G. Goos, Eds. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1996.

[9] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully dynamic algorithms for maintaining shortest paths trees," *J. Algorithms*, vol. 34, no. 2, pp. 251–281, 2000.

[10] V. King, "Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs," in *FOCS*, 1999.

[11] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths," *J. ACM*, vol. 51, no. 6, pp. 968–992, 2004.

[12] H. Gonzalez, J. Han, X. Li, M. Myslinska, and J. P. Sondag, "Adaptive fastest path computation on a road network: a traffic mining approach," in *Proceedings of the 33rd international conference on Very large data bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 794–805. [Online]. Available: http://portal.acm.org/citation.cfm?id=1325851.1325942

[13] E. Kanoulas, Y. Du, T. Xia, and D. Zhang, "Finding fastest paths on a road network with speed patterns," in *Proceedings of the 22nd International Conference on Data Engineering*, ser. ICDE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 10–. [Online]. Available: http://dx.doi.org/10.1109/ICDE.2006.71

[14] N. Jing, Y.-W. Huang, and E. A. Rundensteiner, "Hierarchical optimization of optimal path finding for transportation applications," in *CIKM*, 1996, pp. 261–268.

[15] S. Shekhar, A. Fetterer, and B. Goyal, "Materialization trade-offs in hierarchical shortest path algorithms," in *Symposium on Advances in Spatial Databases*, 1997.

[16] S. Shekhar and D.-R. Liu, "CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations," *TKDE*, vol. 9, no. 1, pp. 102–119, 1997.

[17] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," Tech. Rep., August 2008.

[18] A. V. Goldberg, H. Kaplan, and R. F. Werneck3, "Reach for a: Efficient point-to-point shortest path algorithms," Tech. Rep., October 2005. [Online]. Available: http://www.avglab.com/andrew/pub/msr-tr-2005-132.pdf

[19] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005, pp. 156–165.

[20] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "In transit to constant time shortest-path queries in road networks," in *ALENEX*, 2007.

[21] D. Delling and D. Wagner, "Landmark-based routing in dynamic graphs," in *WEA'07: Proceedings of the 6th international conference on Experimental algorithms*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 52–65.

[22] D. Schultes and P. Sanders, "Dynamic highway-node routing," in *WEA'07: Proceedings of the 6th international conference on Experimental algorithms*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 66–79.

[23] D. Delling and G. Nannicini, "Bidirectional core-based routing in dynamic time-dependent road networks," in *ISAAC '08: Proceedings of the 19th International Symposium on Algorithms and Computation*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 812–823.

[24] M. Hua and J. Pei, "Probabilistic path queries in road networks: traffic uncertainty aware path selection," in *Proceedings of the 13th International Conference on Extending Database Technology*, ser. EDBT '10. New York, NY, USA: ACM, 2010, pp. 347–358. [Online]. Available: http://doi.acm.org/10.1145/1739041.1739084

[25] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis, "Continuous nearest neighbor monitoring in road networks," in *VLDB*, 2006.

[26] Y. Xiao, W. Wu, J. Pei, W. Wang, and Z. He, "Efficiently indexing shortest paths by exploiting symmetry in graphs," in *EDBT*, 2009.

[27] J. Sankaranarayanan, H. Samet, and H. Alborzi, "Path oracles for spatial networks," *PVLDB*, vol. 2, no. 1, pp. 1210–1221, 2009.

[28] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng, "On trip planning queries in spatial databases," in *SSTD*, 2005.

[29] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden, "Cartel: a distributed mobile sensor computing system," in *SenSys*, 2006.

[30] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971. [Online]. Available: http://www.jstor.org/stable/2629312

[31] D. Eppstein, "Finding the k shortest paths," *SIAM J. Comput.*, vol. 28, no. 2, pp. 652–673, 1999.

[32] J. Hershberger, M. Maxel, and S. Suri, "Finding the k shortest simple paths: A new algorithm and its implementation," *ACM Trans. Algorithms*, vol. 3, no. 4, p. 45, 2007.

[33] V. Martins, M. Margarida, B. Pascoal, and E. Queir, "A new implementation of Yen's ranking loopless paths algorithm," *4OR: A Quarterly Journal of Operations Research*, vol. Volume 1, no. 2, pp. 121–133, 2003.

[34] M. Kuby and X. Zhogyi, "A minimax method for finding the k best differentiated paths," *Geographical Analysis*, vol. 29, no. 4, pp. 298–313, 1997.

[35] V. Akgn, E. Erkut, and R. Batta, "On finding dissimilar paths," *European Journal of Operational Research*, vol. 121, no. 2, pp. 232 – 246, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/B6VCT-3Y51T2P-3/2/de56ccd7dc4e358bb102b3f92c144ccb

[36] B. Huang, Q. Wu, and F. B. Zhan, "A shortest path algorithm with novel heuristics for dynamic transportation networks," *Int. J. Geogr. Inf. Sci.*, vol. 21, no. 6, pp. 625–644, 2007.

[37] P. Kumar and A. Yıldırım, "Computing minimum volume enclosing axis-aligned ellipsoids." *Journal of Optimization Theory and Applications*, vol. 136, no. 2, pp. 211–228, 2008.

[38] "Yen's K-shortest-paths," http://code.google.com/p/k-shortest-paths.