Real Time Routing for Road Networks

Aakriti Gupta, J. Lakshmi, S. K. Nandy
Supercomputer Education and Research Center
Indian Institute of Science
Bangalore, India
aakritty@gmail.com, {jlakshmi, nandy}@serc.iisc.ernet.in

***Abstract*—Routing in road networks is an old problem with renewed interest. In this work, we focus on how to extend real-timeliness to the routing problem. With the increasing availability of rich time dependent data in the form of current traffic, weather conditions etc. and methods to compute its impact on the travelling time in the given road segment, location based services are increasingly becoming more demanding. While many algorithms exist to speed up the traditional Dijkstra's algorithm for computing shortest paths in static road networks, only few have shown applicability in the dynamic case adapting to traffic updates on the road network. We note that most algorithms either use history based modelling of road traffic to compute the travelling time, neglecting the real time updates or distributed algorithms are used which let the client decide which way to go based on its local view of the graph and updates relevant to it. We give example scenarios illustrating the shortcomings of both these approaches and identify the need for an approach which is able to handle real time updates and where route computation requires server involvement and global viewpoint. We model the routing problem into a real time job scheduling problem where jobs are the shortest path queries submitted by the users and deadlines are computed on the fly depending on the user characteristics (current location, speed group etc.). The framework is a proactive one, in the sense that if the system has knowledge of updates relevant to its users, the results are communicated back to the user without them specifically asking for it. This is done by sending revised route to the user before user takes the next turn on the pre-specified path. Potentially any routing algorithm that is capable of dealing with real time network updates can be used to compute shortest path jobs in our framework, we demonstrate results with Dijkstra's algorithm. We also show that using the graph characteristics and by doing some precomputation, it can be predicted that which algorithm from our library could result in a faster response for a particular query. We show results for Dijkstra's algorithm and it's bidirectional variant for this observation.**

I. Introduction

Geospatial road networks are being used extensively for many navigation based applications, most important being finding optimal route between a given source and destination. The increase in the popularity of such location based services and the need of doing real-time processing for it has resulted in a renewed interest in the optimal routing problem in the recent past. Many interesting algorithms have been developed over the years focusing mainly at speeding up the route computation on continental sized networks. Most work in the area have shown tremendous success in speeding up the traditional Dijkstra's shortest path algorithm for static graphs [1] [2] [3]. The routing problem can be modeled into a graph where the edges represent road segments and the vertices represent intersections of the road networks. The edge weight can be the distance covered on that road segment, time taken to travel on that road segment or function of both. The graph thus modelled can either be static (where edge weights remain constant throughout) or dynamic (where edge weights are variable). In time dependent scenario, the cost of travelling through an edge is generally dependent on the current traffic on that road segment. Problem is thus modeled into a graph whose edge weights can either be a function of current time, or can follow some historical pattern, or are determined by an asynchronous stream of updates.

Road networks experience changing traffic and weather conditions which makes dynamic graphs a more realistic representation for them. But many algorithms which successfully speed up shortest path computation in static graphs, fail to do so for dynamic graphs. This is because most of these algorithms rely on two phases for computation [4]. Phase I involves precomputation, which is done 'offline' and is the time consuming step. Phase II involves the actual query computation using a given source and destination. This step happens 'online' and the fastest algorithm takes a fraction of a second to do so, even on continental sized networks. For static graphs, where edge weights and the graph topology remains constant, precomputation step is done only once and usually takes a few hours. For dynamic graphs, phase I becomes prohibitive since it is not possible to do the precomputation every time the graph is updated. To overcome this and include dynamism in the graph, history based modelling of road traffic is done and edge profiles are developed. Edge profiles determine the edge cost for a given time of the day. So for example, if edge costs represents the time it takes to travel on a particular edge, edge profile could tell what will be the travel time if you begin your journey at 8am, 9am and so on. Precomputation step is then adapted for these time dependent graphs modelled after traffic history [6]. This step however, is found to be quite expensive both in terms of memory consumption and the time it takes which in some cases can take more than a day [5]. This makes revision of edge profiles and the precomputation on it increasingly difficult.

The other major disadvantage is the fact that it fails to take into account the available real time updates on the road networks. However, incorporating real time updates is indeed important to get more accurate and useful results. We show cases where results can go wrong, or information is not fully utilized if real time updates are not considered. In this work, we try to incorporate real time updates along the road network and discuss one possible approach of propagating these updates to the user in time. We do this by modelling the problem as a real time job scheduling problem. Jobs are the shortest path queries submitted by the system users when they first connect with the system or by the system itself, to recompute the shortest path in response to updates along the network.

Deadlines for these recompute jobs are computed online by the system using the user's current location, among other things. These jobs have a ***scheduling window***, which marks the time period within which it should be scheduled for computation. It is desired that recomputation takes place after the system is allowed to evolve and doesn't take place right away. So the start of scheduling window identifies the time after which it is okay to invest in the recomputation. It is also desired that the recomputed route should reach the end user before they move along the route and the update becomes irrelevant to them. Thus, the end of scheduling window marks the time before which the job should be scheduled for it's result to be useful for it's user. It is intended that any job is scheduled as late as possible within its scheduling window so that latest updates are incorporated in the route computation to the maximum extent. The actual route computation can be done using any algorithm which is capable of dealing with dynamically changing graph and doesn't rely on heavy precomputation step. In our work, we show results using Dijkstra's shortest path algorithm.

We also discuss the possibility of using a query dependent algorithm for route computation. Particularly, we try to optimize the usage of current routing algorithms by taking into consideration the underlying structure of the network and using that information to choose which algorithm to use for a given query. Specifically, we treat dense regions in the network differently from the sparse regions by defining the notion of graph density. Graph density is basically an estimate of the number of neighboring vertices of a given vertex in some distance range. By doing a real time prediction of the nature of subgraph that needs to be dealt with for a particular query, we are able to choose the most appropriate algorithm for the query computation. Given a source-destination pair and the results of some static precomputations it can be determined which of the route computation algorithms from the library is most suitable for computing the optimal route. We show results for this by comparing Dijkstra's algorithm and its bidirectional variant [7]. The key contributions of this work are as follows:

- We demonstrate a proactive method for using real-time updates in road networks that can offer better utility for travellers.
- We propose a graph density based method to choose time optimal algorithm for query dependent route computation on networks.

The rest of the report is organized as follows: In section II related work is discussed which is followed by section III, where we discuss the motivation for developing a proactive routing framework which is able to handle real time updates along the road network. We then proceed with our system model in section IV, where we model the problem as a real time job scheduling problem. In section V, we present a case study and results from our simulations. We compare our framework with a FIFO based setup and find that our framework is able to handle more number of queries in a given time and avoid wastefull computation. We discuss conclusions in section VI.

II. Related Work

Even with the availability of rich real time information, current navigation systems rely on history based models for route guidance. Most popularly, they formulate models which are capable of predicting the optimal route between a given source and destination according to the time of departure [2] [3] [4]. This approach lacks in adapting to the real time changes in the road network such as an accident or development of a traffic jam. Other interesting approaches include T-drive [12] where authors have collected time dependent taxi trajectories over a period of time. For each query, they mine this data to suggest a route to the new user. They have reported good results and attribute that to the intuitiveness of taxi drivers. [13] also proposes mining of traffic data to find the optimal route.

In another work [14], a completely decentralized approach is taken where cars act as intelligent agents and communicate among each other to reach an optimal route independently. Game theoretic methods are employed to reach the conclusion. The drawback of such decentralized approaches is that they require the presence of a high-end device at the client's side, which needs to store and maintain network data. We would ideally like the system to work well for thin clients too, who are only capable of broadcasting their location (using GPS etc.) and have some means of communication to get routes from the server. In [15], authors propose sending 'k' best possible routes to the client and then let the client choose which one to take. Server also broadcasts any real time updates across the road network and its up to the client to select relevant updates for its local view of the graph. Using this, the client then computes the best possible route to follow. This approach is proposed to handle the scalability issues by shedding some load off the server to the client itself. But the reported results suggest that the local computation does not always result in globally optimum results.

One of our assumptions is the availability of some mechanism which analyzes the updates along the graph and makes the proper edge weights assignments. In [16] authors integrate historical and real time information from multiple sources and develop energy/emission parameters for vehicles experiencing different traffic and road conditions.

III. Motivation

Many algorithms use history based modelling of road traffic to compute the travelling time. They neglect the real time updates completely or accumulate the updates over time and adapt the model of road traffic accordingly. But with the real time updates being available along the network, routing systems can now use it to provide better utility to its users. Accumulated updates can only be used for route computation at a later stage, and will not benefit the users who are currently travelling on the affected areas. One common limitation of all these approaches seen thus far is that after route computation, user is abandoned and only after the user specifically asks for route updation, a refreshed route query is registered and computed using the current location. In this work, we instead propose a proactive system, which continuously tries to track the user and any updates on the path that the user is on. Consider the scenario in Figure 1. Here, initially when the user asks for a route from point A to point B, the system computes it as A-P-Q-B (Fig. 1(a)). Upon getting this result, user takes the road segment AP and starts moving. However, due to some event (an accident for example) as shown in Fig. 1(b), traffic on edge QB has started slowing down, making A-P-R-B a more favorable path (Fig. 1(c)). If however, the system fails

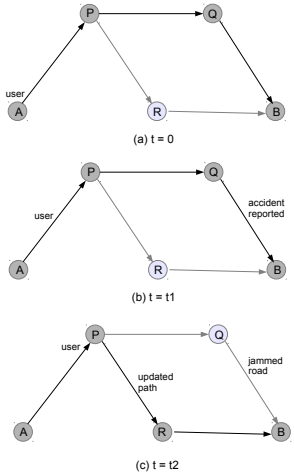

Fig. 1. Motivating example for proactive route updation

to communicate this to the user before she starts travelling on the road segment PQ, then this user ends up travelling on a less optimal path in spite of the system being able to prevent it. A proactive system, which updates the route for its users based on latest information along the network can avoid such scenarios and make sure all the information that it possesses is used to benefit maximum users.

The other trend followed in related work is to use distributed algorithms which let the client decide which way to go based on their local view of the graph and updates relevant to it [14]. Such approaches cannot guarantee optimality. Note that even in the absence of an accident on the current path, there may be scenarios when a proactive system will fair better for its clients. For example, when there is disturbance on nearby roads, because of which traffic gets routed to the current path user is travelling on, thereby decreasing the desirability of this path. Also note that not all mobile devices today have the capability to handle computations and communication that is required here. This makes such applications limited to fat clients and prohibitive for thin clients.

Motivation for discussing the possibility of using a query dependent algorithm for route computation, comes from the various work done in literature. Many algorithms that are able to speed up Dijkstra's algorithm do so by using the underlying graph structure of road networks, which makes it different from other general graphs. In [8], authors use the fact that certain roads (or graph edges) like highways, are likely to be preferred in a long distance travel. Using this property, they have optimized their route computation for such queries. But highways play no special role when it comes to short distance queries. Using the geographical locations of the source and destination points, and the underlying graph characteristics it can thus be estimated what kind of algorithm should be used for the computation. A study of the algorithms is required in this case, to categorize them based on their use cases. For example, algorithm A is more suitable for short range queries, algorithm B is more suitable for dense regions of the graph, etc. In the next section, we propose our proactive framework, which deals with real time updates and tries to communicate

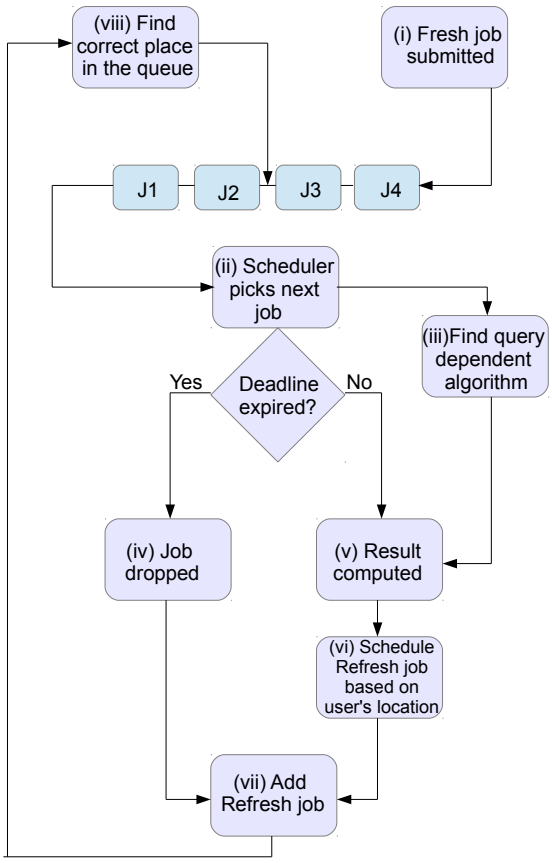

Fig. 2. Overall System Model

the results back to the user in time, followed by our approach for query dependent route computation.

IV. System Model

Routing in road networks is modelled as shortest path problem on a graph with positive edge weights. Users pose queries to find optimal paths between a source and a destination, and navigation systems typically respond to them in a best effort fashion. As discussed in the previous section, route re-computation is of utility to it's users if network condition changes and should reach them before they take the wrong turn. To enable this, we model the problem as a real time job scheduling problem, where such recompute jobs come with a deadline.

Overall system model is shown in Fig. 2. *Fresh jobs* enter in the job queue from one end, and are removed from the Job queue for computation from the opposite end. It is scheduler's job to find the next job to be computed. If the deadline to compute this next job has not expired, its result is computed otherwise it is dropped. In either case, a *refresh job* is added in the system so that user can be updated about the changing network conditions and the path that has become more suitable as a consequence. These *refresh jobs* come with a deadline, which decides their priority and appropriate place in the job queue. The system model is detailed below in the subsections.

We make the following assumptions: Users can be polled for their current location and some mechanism is in place which analyzes the graph updates in the form of current traffic, weather conditions etc. and appropriately adjusts the edge weights in the graph. For our purpose, we assume the edge weights represent the time it would take to travel on the road segment represented by the edge.

A. Defining a Job

Jobs are the shortest path queries submitted into the system either by the users, or by the system itself in response to updates along the network. We broadly define three categories of jobs: *fresh*, *refresh* and *redo* as follows. While *fresh jobs* are added by the users, *refresh jobs* are system generated as a proactive response to the changing network. *Redo jobs* can either be generated by user or system itself.

- ***fresh job***: Represents a new user connecting to the system, asking for a shortest path. Each fresh job is given a unique ID which represents this user. Subsequent refresh and redo jobs carry forward this same ID.
- ***refresh job***: Represents a case when user is following the path and system pro-actively decides whether the user should continue or switch to a different path.
- ***redo job***: Represents scenario where system was late in giving response to the user, who is now on a different path. Or the user chooses to take a different path and re-computation is required based on current location. Presently, in our implementation we are treating them as fresh jobs.

All jobs are aperiodic and independent of each other in the sense that they don't follow any precedence relations among them. Jobs are also classified as non-preemtive, because if a job is paused to make resources free for another job, then it can't be resumed from where it was left, since network might have seen updates during this time. Hence, computation will have to start from beginning. Job scheduling is dynamic and priority-driven. Scheduling is dynamic, since job can be dispatched from the job queue to any of the available processors in the system, there is no binding relation between a job and a particular processor. It is priority-driven since scheduling decisions are based on the priorities of the jobs and take place when events such as job completions occur.
Fresh jobs are submitted without any deadlines, and are added at the back of the job queue (Fig. 2, block [i]). So their computation depends upon the length of job queue at the time of their addition. After any shortest path computation is completed (in the form of any job), a *refresh job* with source as the next hop and the same destination is added (Fig. 2, block [vii]). For example, in Fig. 1 after job(A,B) is computed, job(P,B) is added as a *refresh job*. *Refresh jobs* come with a deadline since the result of these jobs must be computed before the user reaches the next turn (vertex P in the above example), otherwise the user will go on the previously specified path. This ensures that for a given user ID, atmost one job exists in the system at a given time.

B. Computing Job Deadlines

Refresh jobs come with a **scheduling window**, marked by two timestamp values that are unique to each *refresh job*. We note that *refresh jobs* should not be scheduled too soon. For example, it doesn't make much sense to recompute a user's route when they just began. Thus, the scheduling of a *refresh job* must be delayed as much as possible so we can get the most recent updates on the graph. We call Timestamp1 as the *maturity time* for a job, after which it is eligible for scheduling. A *refresh job* must not be scheduled before its Timestamp1 and must not be scheduled after its Timestamp2 value, which marks its *deadline*. These jobs have a deadline since the system must send the results back to the users before they take any turns forcing them to travel on a suboptimal path. These timestamp values are computed as follows and are illustrated in Fig. 3.

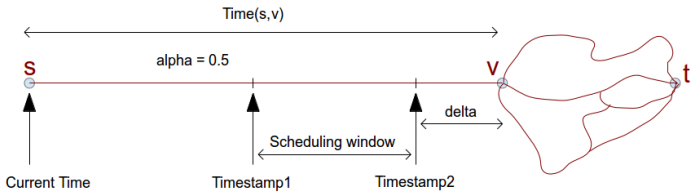

Fig. 3. Timestamps for refresh jobs

$$Timestamp1 = Current\_time + \alpha * time(s, v) \quad (1)$$

$$Timestamp2 = Current\_time + time(s, v) - \Delta \quad (2)$$

$$Timestamp1 \leq Scheduling\_time \leq Timestamp2 \quad (3)$$

Here α is any constant ϵ (0,1) which is used to delay the execution of the refresh job so that more updates can be included. Function time(s,v) returns the time it would take for user to travel from point s to point v in the current network scenario. Δ is the upper limit on the estimated time taken to compute the shortest path + communicate the result back to user. Higher α value indicates that the route will be computed as late as possible and routing window will be narrow. α is used in the computation of the maturity time (Timestamp1) of a job and Δ is used in the computation of the deadline (Timestamp2) of a job. While scheduling of a job after Timestamp1 ensures that the latest updates in the network are included in the route computation, scheduling before Timestamp2 ensures that the results are sent to the user in time, before she reaches the next junction and hence can take the appropriate turn. We drop a job if the current time exceeds its Timestamp2 value since it would result in wasted computational effort (Fig. 2, block [iv]).

C. Job Scheduling

The overall scheduling algorithm is summarized in the algorithm below.

Algorithm 1 Job Scheduling

```
1: while (1) do
2:   while !(Queue Empty) do
3:     ptr ← queue head
4:     if ptr.jobtype = fresh then
5:       goto compute
6:     else
7:       if ptr.jobtype = refresh & matured(ptr) =
         true then
8:         goto compute
9:       else
10:        if ptr.jobtype = refresh & expired(ptr) =
           true then
11:          dropped jobs ++
12:          goto add
13:        else
14:          ptr ← next(ptr)
15:        end if
16:      end if
17:    end if
18:  end while
19:  compute:
20:    compute shortest path
21:  add:
22:    find next hop
23:    add refresh job
24: end while
```

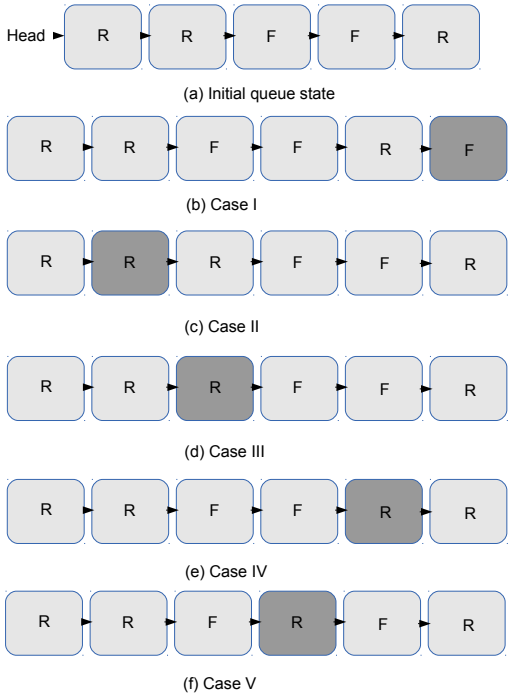

Fig. 4. Job Queue Insertion

Relative priority between two *refresh jobs* is calculated using their deadlines or Timestamp2 values. Earliest deadline first algorithm with Timestamp2 as deadline is used to find the next job to be scheduled for all the jobs that have matured at a given time. Relative priorities between *refresh jobs* and the *fresh jobs* are dynamically assigned using aging for *fresh jobs*. A job queue is maintained, such that at any given time the first job in the queue is having the highest priority for scheduling. If, however this job is not matured at the time, then next job is checked and so on. Insertion in the job queue (Fig. 2, block [viii]) follows the rules as stated below. Assume queue state as shown in Fig. 4(a). Here Queue is maintained as a linked list, where cell with 'R' denotes a *refresh job* and 'F' denotes a *fresh job*. Queue head is on the left, from where next job to be computed is searched.

- Case I: If a *fresh job* is to be inserted in the job queue, it is added in the back as shown in Fig. 4(b).
- Case II: If a *refresh job* is to be inserted, such that its Timestamp2 value falls between two consecutive refresh jobs, it is simply added in the middle as shown in Fig. 4(c).
- In case there are *fresh jobs* in between, then depending upon the priority, the position of the new job changes. If *refresh job* is given higher priority over *fresh jobs* then case III is followed as shown in Fig. 4(d). If *fresh job* is given higher priority then case IV is followed as shown in Fig. 4(e). If the relative priorities of *fresh jobs* and *refresh jobs* are dynamically computed, then any of the case III, IV or V [Fig. 4(f)] can be followed.

In our framework, we followed dynamic priority between *fresh jobs* and *refresh jobs* by giving an aging factor to the *fresh jobs*. Initially Case III is followed, giving higher weightage to a *refresh job* over *fresh job*. But, each time a *fresh job* is pushed back in the queue to make room for a *refresh job*, its aging factor increases. If the age increases beyond a threshold, all other *refresh jobs* are added behind it following case IV and case V. Addition of *refresh job* is explained in algorithm2 below. When a *fresh job* enters queue, it's age is 0. After every insertion of a *refresh job* before it, its age is incremented. If its age value exceeds some predefined threshold, then it gets higher priority then all the other *refresh jobs* that will enter the system from then on.

Consider two refresh jobs denoted as R and fresh job as F. The following equations define the relative priorities.

If, $Timestamp2(R1) < Timestamp2(R2)$, then:

$$Priority(R1) > Priority(R2) \tag{4}$$

If, $AgingFactor(F) < Threshhold$, then:

$$Priority(R) > Priority(F) \tag{5}$$

If, $AgingFactor(F) \geq Threshhold$, then:

$$Priority(F) > Priority(R) \tag{6}$$

D. Query Dependent Route Computation

While computing route for a given query, further optimizations can be made by selecting the best algorithm for that particular query (Fig. 2, block [iii]). Most algorithms use bidirectional search over simple Dijkstras algorithm because of its

Algorithm 2 Insertion of refresh job

```
1: if (Queue Empty) then
2:    queue head ← next_refresh_job
3:    return done
4: end if
5: ptr ← queue head
6: while ptr != NULL do
7:    if ptr.jobtype = refresh && ptr.timestamp2 > next_refresh_job.timestamp2 then
8:       insert next_refresh_job before ptr
9:       return done
10:   end if
11:   if ptr.jobtype = fresh && ptr.age < threshold then
12:      insert next_refresh_job before ptr
13:      temp ← ptr
14:      while temp.jobtype = freshjob do
15:         temp.age++
16:         temp ← next(temp)
17:      end while
18:      return done
19:   end if
20:   ptr ← next(ptr)
21: end while
22: add next_refresh_job at queue end
23: return done
```

lesser average query computation time. However, we observed that for a significant percentage of queries, bidirectional search does not perform better than Dijkstras algorithm. We tried to predict which algorithm out of two will be more suitable for a given query. Generally, if an algorithm is able to reduce it's search space, it takes lesser time to find the solution. In the context of shortest path algorithms, reducing the search space means reducing the number of vertices that the algorithm visits while coming up with the optimal path.
Hence, if we are able to successfully estimate the search spaces of two algorithms for a given query, we can choose to use the one with lesser search space, thereby reducing the time taken for that particular query. In this work we estimate the search space using graph density, which is a term we use to donate the number of vertices around a given vertex within some distance range. By using this notion, we are able to predict which of the two algorithms (Dijkstra's or it's bidirectional variant) should be used for a given query so that we are able to achieve lesser average query time than if we were using either of the two algorithms for all of the queries.

1) Bidirectional Search: Bidirectional version of Dijkstra's search is a method in which two parallel Dijkstra's searches are run, one from source and the other from the destination, until they meet somewhere in the middle. Now the actual shortest path can be found among the vertices seen thus far by the two searches. The success of bidirectional search over Dijkstra's algorithm is in the reduction of the search space as can be seen in Fig. 5. If the reduction in search space is not much (beyond a certain limit) then the speedup of carrying out bidirectional search falls short of the overhead incurred. Intuitively, if for a given query, graph is dense in the regions pruned by bidirectional search, it is beneficial. But if the pruned regions are sparse, we don't extract much benefit out of using bidirectional search for this query as search space

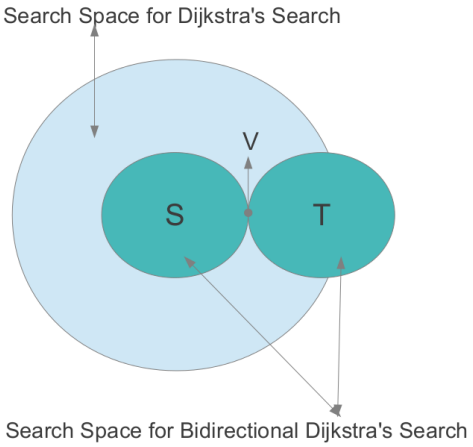

Fig. 5. Search space of Dijkstra's search and bidirectional Search

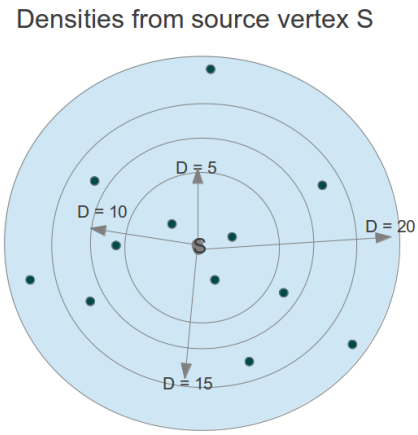

Fig. 6. Graph Density

reduction is not much. In our method, we try to estimate the reduction in search space for a given source and destination, before running the query itself.

2) Graph Density: In our setup, graph density is a function of two values: source vertex 'S' and distance range 'D'. Density the number of vertices whose shortest path distances from source are inside the given distance range, shown in Fig. 6. For example, density around source S for distance range = 10 is 5, whereas for distance range = 15, it is 9.

3) Estimating search space: To calculate exact density we need to compute all pair shortest path distances and form our density table for lookup as part of pre-computation. This can be improved using the approximation approach as discussed in [11]. They reduce the $O(V^2)$ space requirement to $O(\frac{V}{\varepsilon^2})$ where V is the number of vertices in the graph and ε is the error in the accuracy of the shortest path distance. Using this precomputed information, first the search space of Dijkstra's search is found which is equal to the density of source vertex, with distance range equal to the approximate shortest path distance to the destination. Then search space of bidirectional search is approximated, by adding the densities of source vertex and destination vertex with distance range equal to the distance upto a middle vertex. Then ratio of search space of

Dijkstra's search over bidirectional search is computed. If this ratio is higher than some threshhold, bidirectional search is used, else Dijkstra's search is used.

V. Case Study

In this section we present findings from our simulations. Road networks used for the experiments are various cities of USA as provided by the Dimacs Implementation Challenge [9]. We worked on the Great Lakes, USA (LKS) graph instance for testing our real time framework, and many other input graphs for comparing Dijkstra and bidirectional Search, details of which are provided in Table I.

TABLE I. Graph instances used

Graph Name	Number of Vertices	Number of Edges
TG	18263	23874
BAY	174956	223001
SF	321270	800172
COL	435666	1057066
LKS	2758119	6885658

A. Experimental Setup

All experiments were performed on a system with 7.9 GiB memory, Intel Core2 Quad CPU Q9550 @ 2.83GHz x 4 processor and 300 GB disk. The implementation is C based, using pthreads library. Some pre-computations for computing graph density were done in R using the igraph package [10]. The graphs are static in nature, and the edge costs represents the time it takes to travel on the road segment represented by the edge. Due to the lack of availability for dynamic datasets, we used random updates along the graph to demonstrate our model. Random shortest path queries are generated using internal BASH function $RANDOM.

B. Implementation

The framework was implemented using C pthreads library. Main thread or scheduler, initializes a fixed number of worker threads whose job is to compute shortest path and add a refresh job if the destination hasn't been reached yet. Scheduler manages the job queue, and selects the next job to be computed by traversing the queue from the head. The first fresh job or matured refresh job is picked and given to a free worker thread. A separate thread focuses on updating the edge weights along the network. New jobs are regularly fed to the scheduler in a random order.

C. Results for Real Time Routing Framework

We compared our framework with FIFO queue setup. In FIFO setup, priority is given to the job which was submitted earlier and this priority doesn't change dynamically. Since, job deadlines are not maintained, FIFO responds in best effort fashion. Our framework on the other hand prioritize jobs based on their deadlines and dynamically change it using aging. We also, avoid doing computations whose results can't reach users in time. Keeping everything else same, jobs are added in the FIFO queue using their Timestamp1 values and are computed in first come first serve basis without any regard to their deadlines. If a job is computed after it's Timestamp2 value expires, then it is considered as wasted computation and termed as *dead job*. The number of dead jobs are then

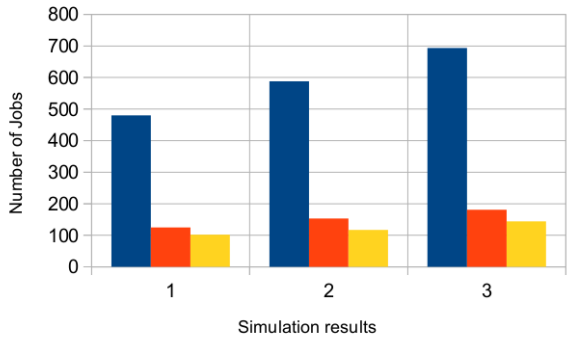

Fig. 7. Simulation with α = 0.5 and two worker threads

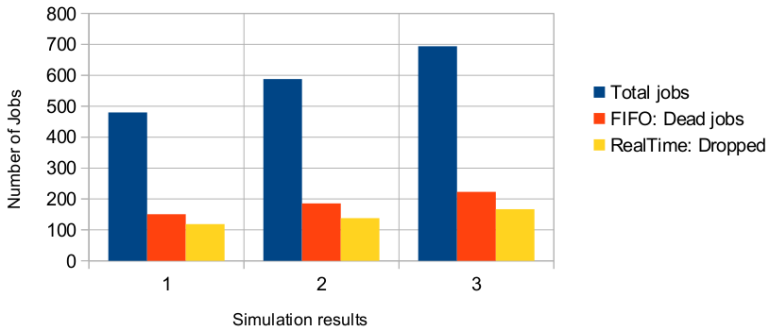

Fig. 8. Simulation with α = 0.6 and two worker threads

compared with the number of *dropped jobs* from our setup. Fig. 7 shows the simulation results with α = 0.5. Total queries (fresh and refresh) submitted are shown in blue, red bar shows the number of dead computations being made in case of FIFO. Yellow bar shows the number of dropped jobs when our framework was used. Not only is the number of dropped jobs lower than the dead jobs, it is to be noted that our framework took significantly lesser computation time since a lot of wastefull query computation was avoided. We can see that for an increasing number of total queries, number of dead jobs (or wasted computation) by FIFO also increases, and stays higher than the number of dropped jobs in our framework. Note that dropped jobs are not wasted computation since they are not executed at all. Fig. 8 shows simulation results for α = 0.6 for the same number of total queries. With increasing α, the scheduling window decreases and we see that more number of queries result into dead computation or are dropped as compared to before. Higher α means that there is a requirement of including most recent updates in the results, which in turn implies increased number of dead and dropped jobs. In this case it is better to use our framework as FIFO will spend a significant percentage of time in wasteful computations. Lower value of α means that it is not required to include recent updates along the network in the path computation, in which case our framework works similar to FIFO. The aging factor for fresh jobs was fixed at 3 for these experiements (at most three refresh jobs can push a fresh job back in the queue).

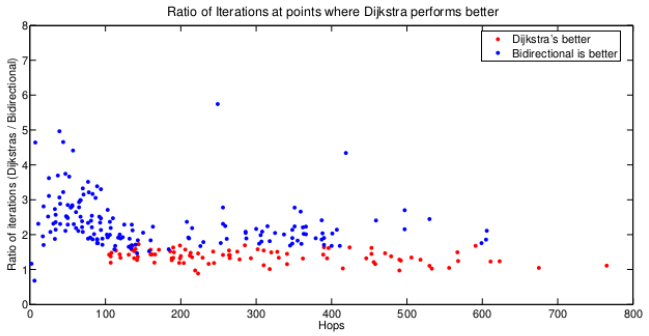

Fig. 9. Ratio of iterations at points where Dijkstra's algorithm performs better

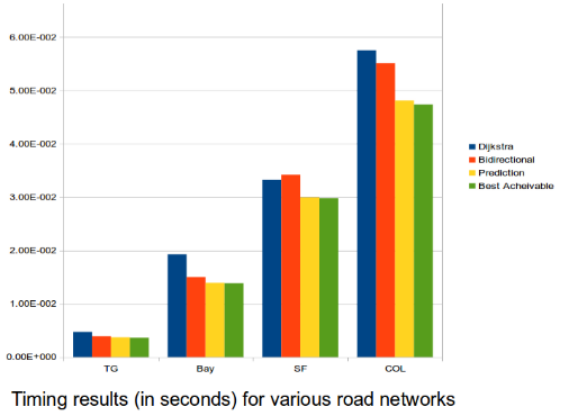

Fig. 10. Dijkstra vs. Bidirectional vs. Predicted vs. Max achievable

D. Results for Query Dependent Route Computation

Using the notion of density as defined above, we ran experiments where we take ratio of the search space as will be spanned by Dijkstra's search over that of bidirectional search. Experimentally we found if the ratio is not greater than 1.4, Dijkstra's search performs faster. For cases where ratio is greater running bidirectional search is more likely to give better results. Fig. 9 shows that out of 250 random queries, Dijkstra's algorithm outperformed bidirectional search for 83 queries for San Francisco Road Network (SF). These are shown in red, rest in blue. The horizontal axis here represents the number of hops in the shortest path. Vertical axis represents ratio of iterations taken by Dijkstra over bidirectional search. It can be observed that there's a nice separation between the queries for which bidirectional search is better at around y = 1.4. Also, we observe that it has nothing to do with long or short distance queries (hops). At the query processing time, given source and destination vertex, we do a lookup to determine in which distance range does the destination vertex lie and thus get the density value of source for that distance range. Fig. 10 shows speedup achieved in average query time by using this lookup based prediction.

VI. Conclusion and Future Work

In this work we developed a deadline based, real time routing framework which is proactive in it's route updation and includes the real time updates for route computation that are available along the network. The categorization of jobs as fresh and recompute jobs and calculating their scheduling window is the takeaway from this work. Experimental results suggest that our framework fairs better over a general FIFO like method as it reduces the wasted computational effort and thus is able to do more work. Possible extensions to this work are including driver dependent information like speed group categorization, and changing the parameters accordingly. We also discussed the possibility of using query dependent algorithm for route computation. Results suggest that between Dijkstra's search and it's bidirectional variant, the nature of subgraph does influence the speed up that can be attained. Future work in this direction would be to categorize more algorithms using underlying network structure and develop their use cases.

References

[1] E. W. Dijkstra, A note on two problems in connexion with graphs, *Numerische mathematik*, vol. 1, no. 1, pp. 269271, 1959.

[2] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, Contraction hierarchies: Faster and simpler hierarchical routing in road networks, in *Experimental Algorithms*. Springer, 2008, pp. 319333.

[3] H. Bast, S. Funke, P. Sanders, and D. Schultes, Fast routing in road networks with transit nodes, *Science*, vol. 316, no. 5824, pp. 566566, 2007

[4] P. Sanders and D. Schultes, Engineering highway hierarchies, in *AlgorithmsESA* 2006. Springer, 2006, pp. 804816.

[5] D. Delling and D. Wagner, Time-dependent route planning, in *Robust and Online Large-Scale Optimization*. Springer, 2009, pp. 207230.

[6] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, Time-dependent contraction hierarchies. in *ALENEX*, vol. 9, 2009.

[7] A. V. Goldberg, H. Kaplan, and R. F. Werneck, Reach for a*: Efficient point-to-point shortest path algorithms. in *ALENEX*, vol. 6, no. 2, 2006, pp. 129143

[8] Abraham, Ittai; Fiat, Amos; Goldberg, Andrew V.; Werneck, Renato F. "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms". *ACM-SIAM Symposium on Discrete Algorithms*, pages 782-793, 2010.

[9] 9th DIMACS Implementation Challenge: Shortest Paths (2006), http://www.dis.uniroma1.it/challenge9.

[10] G. Csardi and T. Nepusz, The igraph software package for complex network research, *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.sf.net

[11] J. Sankaranarayanan and H. Samet, Query processing using distance oracles for spatial networks, *Knowledge and Data Engineering, IEEE Transactions on*, vol. 22, no. 8, pp. 11581175, 2010.

[12] Yuan J., Zheng Y., Xie X, and Sun G. T-drive: Enhancing driving directions with taxi drivers intelligence. In *Transactions on Knowledge and Data Engineering* (TKDE 12).

[13] Hector Gonzalez, Jiawei Han, Xiaolei Li, Margaret Myslinska, and John Paul Sondag. 2007. Adaptive fastest path computation on a road network: a traffic mining approach. In Proceedings of the *33rd international conference on Very large data bases* (VLDB '07).

[14] Vasilis Verroios, Konstantinos Kollias, Panos K. Chrysanthis, and Alex Delis. 2008. Adaptive navigation of vehicles in congested road networks. In Proceedings of the *5th international conference on Pervasive services* (ICPS '08). ACM, New York, NY, USA, 47-56.

[15] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, and Jeffrey Xu Yu. 2009. Monitoring path nearest neighbor in road networks. In Proceedings of the *2009 ACM SIGMOD International Conference on Management of data* (SIGMOD '09)

[16] Boriboonsomsin, K.; Barth, M.J.; Weihua Zhu; Vu, A., "Eco-Routing Navigation System Based on Multisource Historical and Real-Time Traffic Information," *Intelligent Transportation Systems, IEEE Transactions* on , vol.13, no.4, pp.1694,1704, Dec. 2012