Assembly Queries: Planning and Discovering Assemblies of Moving Objects Using Partial Information

Reaz Uddin, Michael N. Rice, Chinya V. Ravishankar, Vassilis J. Tsotras

University of California, Riverside, CA, USA

{uddinm,mrice,ravi,tsotras}@cs.ucr.edu

ABSTRACT

Consider objects moving in a road network (e.g., groups of people or delivery vehicles), who may be free to choose routes, yet be required to arrive at certain locations at certain times. Such objects may need to assemble in groups within the network (friends meet while visiting a city, vehicles need to exchange items or information) without violating arrival constraints. Planning for such assemblies is hard when the network or the number of objects is large. Conversely, discovering actual or potential assemblies of such objects is important in many surveillance, security, and law-enforcement applications. This can be hard when object arrival observations are sparse due to inadequate sensor coverage or object countermeasures. We propose the novel class of *assembly queries* to model these scenarios, and present a unified scheme that addresses both of these complementary challenges. Given a set of objects and arrival constraints, we show how to first obtain the set of all possible locations visited by each moving object (the travel corridor), and then determine all possible assemblies, including the participants, locations, and durations. We present a formal model for various tracking strategies and several algorithms for using these strategies. We achieve excellent performance on these queries by preprocessing the network, using Contraction Hierarchies. Experimental results on real-world road networks show that we can efficiently and rapidly infer assembly information for very large networks and object groups.

Keywords

Location-Based Services, Spatio-temporal Trajectories, Shortest Paths, Contraction Hierarchies

1. INTRODUCTION

In many scenarios, we wish to make inferences about the spatiotemporal trajectories of moving objects, and in particular, about their interactions. In a surveillance task, for example, we may wish to determine whether two or more moving objects (e.g., people) could have had a meeting within a region of interest (assembly discovery). If delivery vehicles are traveling on a road network picking up and delivering items, we may wish to arrange for meetings where items picked up from one vehicle may be transferred to one or more other vehicles without violating their delivery schedules (assembly planning). Similarly, individuals who wish to meet as a group may be constrained by itineraries that require their presence at certain locations in the network at certain times.

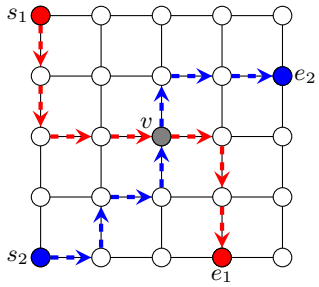


Figure 1: Sparse moving objects observations and their possible trajectories and assembly locations.

Efficiency is paramount, since arrival constraints or arrival observations may be uncertain, dynamic, or unpredictable, requiring recomputation. With delivery vehicles, for example, the schedules for pickup and delivery may depend on the items picked up. If video cameras, RFID, or GPS are used to track objects in surveillance applications, tracking may be spotty due to occlusions, inadequate sensor coverage, loss of signal, privacy concerns, or adversarial countermeasures. Even combining spotty location data for moving objects from many sources, such as location sharing in social networks, check-in applications, surveillance cameras, cell-phone usage, and sighting reports may only yield partial information about object movement.

1.1 Assembly Queries

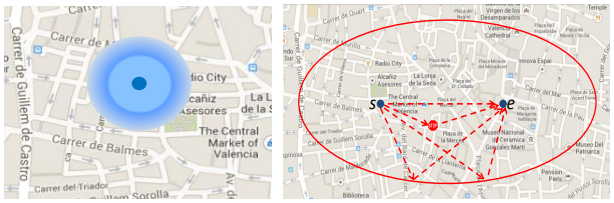
We formalize the problem of *assembly queries*, a novel and important class of queries, and show how to solve it even given incomplete trajectory information, using knowledge of the topology of the underlying transportation network.

Since the assembly planning and assembly discovery problems are equivalent, we discuss the discovery version using a surveillance scenario. We assume that we are given sparse observations, and are required to infer potential assemblies.

Figure 1 shows a simple example. We have observations of

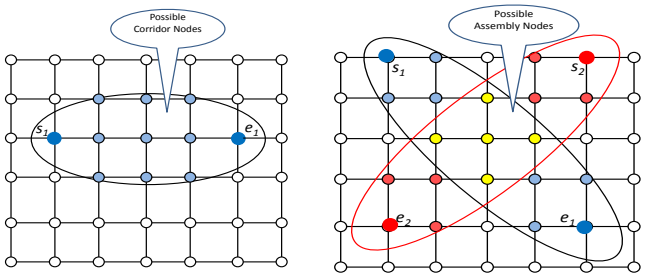
ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235



(a) (b)

Figure 2: (a) Type-I and (b) Type-II uncertainty.



(a) (b)

Figure 3: (a) Corridor (superset) of one moving object (b) Assembly (superset) of two moving objects.

two objects O_1 and O_2 of interest, but only at their respective entry points, s_1 and s_2, and exit points, e_1 and e_2. We have no information either about their trajectories (dashed lines) within the unobserved ("blind") region, or about where O_1 and O_2 could have met within this region (e.g., at node v). Regardless, we want to make inferences about their behavior inside this blind region, specifically about possible meetings and/or communications between them. We assume that we know the network topology within the region (e.g., the grid lines in Figure 1 representing the transportation network), but not the actual trajectories.

Objects may meet secretly somewhere within the blind area if they want to disguise their intentions or mask their associations. If we can identify potential *assembly sites*, where some or all of them could have been present simultaneously, we would have leads useful in detecting the associations they wish to hide. We may commit surveillance resources to these sites based on the number of objects that could have met or the possible assembly durations.

1.2 Possible Approaches

A moving object's location is specified as the coordinate pair $l_k = (x_k, y_k)$. A location *report* has the form $r_k = (l_k, t_k)$ where $l_k = (x_k, y_k)$ is the object's location at time t_k. If $r_i = (l_i, t_i)$ and $r_j = (l_j, t_j)$ are consecutive reports with $t_j > t_i$, the object is typically assumed to follow a straight line between l_i and l_j. However, this assumption is not always valid. The path taken between l_i and l_j may be curved, for instance. Even in free space, the object may follow an arbitrary route between l_i, l_j, its path being constrained only by t_i, t_j, the route length, and a posited maximum speed for the object.

Two sources of uncertainty in object locations have been considered in the literature [1]. The first arises from the imprecision of location sensing (Type-I uncertainty). The second (Type-II uncertainty), arises because location updates are not continuous. Type-I uncertainty is handled by assuming that the object may lie anywhere within a disk of a certain radius around each reported location, as in Figure 2(a). Type-II uncertainty occurs when two consecutive updates are so far apart (in space and/or time) that the intermediate position is not precisely known, as in Figure 2(b). Both types of uncertainties arise in the context of free space movement as well as in network-constrained movement.

To handle Type-II uncertainty, we can first estimate maximum object speed from constraints such as speed limits or the object's maximum achievable speed. Computing the region reachable by the object, as constrained by this maximum speed and the time interval between updates, yields an ellipse, with the two consecutive locations l_i, l_j at its foci. In free space, the object may travel through any point within this ellipse. (A better model may give a probability distribution for the object's location in the elliptical region.) In fact, the object can start at l_i, pause for some time at any point in the interior of the ellipse (but not at the boundary), and still arrive at l_j on time. If the object must travel over a road network, however, Type-II uncertainty exists only if there are several feasible routes inside the ellipse conforming to the reports (l_i, t_i) and (l_j, t_j). Even if there is only one feasible route, the object may have stopped somewhere in between l_i and l_j. Such locations will remain unknown.

In the context of road networks, we will refer to road intersections and other locations of interest[1] as *nodes*. Consider a pair of objects O_1 and O_2 moving in a road network, subject to Type-II uncertainty. Let the sets U_1 and U_2 be the nodes within the elliptical uncertainty regions for O_1 and O_2, respectively (see Figure 3(a)). Let the sets R_1 and R_2 of reachable *corridor* nodes be the collection of nodes on all feasible routes for O_1 and O_2, respectively. Note that $R_1 \subseteq U_1$ and $R_2 \subseteq U_2$, as the corridor nodes are further constrained by travel on the underlying road network. Clearly, the set of assembly sites is $C \subseteq R_1 \cap R_2 \subseteq U_1 \cap U_2$ (Figure 3(b)). Some nodes in this intersection may not be true assembly nodes, however, since we must also account for temporal constraints. Specifically, O_1 and O_2 must be present simultaneously at a node to represent a true assembly.

1.3 Our Contributions

In this paper we first consider the problem of identifying the set of corridor and/or assembly nodes of an arbitrary number of moving objects. We then consider the more general $[\gamma, \tau]$-assembly query, which returns the set of assembly nodes at which at least γ objects could have met for at least τ time units. Finally, we consider the [top-$k(\gamma), \tau$] query, that finds the k assembly nodes with the *largest-possible meeting sizes* and the $[\gamma,$ top-$k(\tau)]$ query, that reports the k assembly nodes with the *longest-possible meeting durations*. We present efficient algorithms to address these queries; moreover, using preprocessing based on the well-known Contraction Hierarchies (CH) approach [2], our solutions are extremely fast, answering the above queries in only a few seconds for large numbers of observed objects. As we shall show, our proposed CH-based method is able to efficiently process the corridors and assemblies of all objects simultaneously within a single aggregate search, as opposed to the naive method which requires O(n) distinct Dijkstra

[1]See Appendix A for further discussion.

searches for n objects. Thus we are able to achieve orders of magnitudes of speed up over the naive method.

The rest of the paper is organized as follows: Section 2 provides related work, while Section 3 formally introduces the tracking model used. The various queries we address are discussed in Section 4 and our algorithms are presented in Section 5. The experimental evaluation using real road networks appears in Section 6. Section 7 concludes the paper.

2. RELATED WORK

Recently, there has been much research on access methods for moving objects; such indexes answer spatio-temporal range queries about past and/or future positions of the moving objects. They typically assume that a moving object moves on a straight line between subsequent trajectory locations and involve some variation on hierarchical trees [3, 4], or some form of a grid-based structure [5] or indexing in parametric space [6]. [7] discuss indexing objects moving in networks. A typical spatiotemporal range query specifies a rectangular spatial range and a temporal interval, and requires all objects that went through that rectangle during the given time interval. Type-I uncertainty for such are considered in [8, 9, 10]. Ni et al. [8] examine Type-I uncertainty about the location of static spatial objects. They consider spatial join of uncertain polygons (representing the boundary of static spatial objects) and propose an R-Tree based method for the join. [9] provides a detailed model for including uncertainty in data indexing, as well as algorithms for querying both past and future locations of the moving objects. [11] addresses Type-I uncertainty (in free space) by using a stochastic model. According to this model, the location probabilities of an object at a particular time depend on its previous location.

Range queries with Type-II uncertainty are considered in [12, 13]. In particular, [12] examines range queries on moving objects in free space while [13] considers snapshot and continuous range queries in network space. [14] considers the continuous nearest neighbors query in free space with Type-II uncertainty. These works are implicitly using the notion of a corridor between two consecutive location updates. Among them, [13] is the closest to ours as it considers Type-II uncertainty in a road network. However, we examine the assembly problem (which requires to compute spatio-temporal intersections among the corridor nodes in the road network of a number of moving objects), which was not attempted in any previous work.

Works on the map matching problem [15] are also related. The aim is to match GPS data to a digital map or road network. Map matching can be affected by both Type-I and Type-II uncertainty. [16, 17, 18] propose map matching algorithms to find the potential routes taken by a moving object under Type-II uncertainty in a road network. [16] uses historic trajectories to find potential routes while [17, 18] use road network topology and spatio-temporal attributes of the trajectory (for which map matching is being done). Nevertheless, none of these algorithms can be used to identify the corridor between two contiguous GPS samples.

Other related problems include destination prediction and alternative route computation. *Predestination*[19] predicts the destination of a moving object by computing the likelihood of each possible region as the final destination. The likelihood depends on the object's past travel history and route taken so far in the current trip. As the current trip progresses the prediction becomes more likely to be correct. [20] starts with destination prediction and then predicts possible routes to the destination as well. [21] finds alternative routes that are significantly different than the shortest path but also 'reasonable with no unnecessary detours'. It maintains that an alternative path should be locally optimal(shortest) to be reasonable. [22] also presents methods to find alternative routes with the goal to be able to measure the quality of alternative routes based on the graph structure. While these methods consider possible routes, none of them can be used to identify all corridor and assembly nodes.

G	Weighted, directed graph of the road network
V	Vertices (a.k.a., nodes) representing road intersections or locations of interest in G
E	Edges representing road segments in G
w	Weights for road segments in E representing minimum time to traverse a road segment
r	Number of objects
O	Set of moving objects $\{o_1, o_2, \ldots, o_r\}$
s_i	Starting node of object $o_i \in O$
e_i	Ending node of object $o_i \in O$
d_i	Minimum time to travel from s_i to e_i
δ_i	Actual transit time from s_i to e_i for object o_i
$d_{\min}$	Minimum d_i amongst objects in O
$\alpha_i(u)$	Availability interval of object o_i at node u
$A_{O'}$	Assembly nodes for objects in $O' \subseteq O$
$\Delta_{O'}(u)$	Assembly interval of objects $O' \subseteq O$ at node u
γ	Minimum assembly size
τ	Minimum assembly duration
k	Number of top nodes in top-k query
ϵ	Detour length scale factor

Table 1: Symbols used in this paper.

3. FORMAL TRACKING MODEL

The underlying transportation network is represented as a weighted, directed graph $G = (V, E, w)$, where V is the set of nodes representing road intersections, $E \subseteq V \times V$ is the set of edges representing roads crossing between nodes in V, and $w : E \to \mathbb{R}_{>0}$ is a positive weight function mapping the edges to their respective travel times. Let $d(u, v)$ denote the *minimum time* of travel, at the maximum speed(s), along the quickest path between nodes u and v.

Let $O = \{o_1, o_2, \ldots o_r\}$ be the set of moving objects being tracked. We use $o_i@\langle u, t\rangle$ to denote a *location record*, i.e., that object o_i was at node $u \in V$ at time t. We consider object locations only at nodes in V (road intersections). If all the nodes that an object visited are known, then there is no uncertainty about this object's route on the road network. Since we assume that not all location records are known, we focus on the route uncertainty between two known (time-) consecutive location records. Between the two nodes reported in these consecutive location records there can be multiple possible routes that the object may have taken, especially if the time interval between these two location records is large.

Let $o_i@\langle s_i, t_{s_i}\rangle$ and $o_i@\langle e_i, t_{e_i}\rangle$ be the start and end records for object $o_i \in O$ as it enters and exits an unmonitored region, with $s_i, e_i \in V$ and $t_{s_i}, t_{e_i} \in \mathbb{R}_{\geq 0}$. Our challenge is to

make inferences about o_i's behavior within this unmonitored region. Object o_i's *transit time* from s_i to e_i is $\delta_i = t_{e_i} - t_{s_i}$. Clearly, $\delta_i \geq d(s_i, e_i) = d_i$ (see Table 1).

For a node $u \in V$, let $ea_i(u) = t_{s_i} + d(s_i, u)$ be the *earliest-arrival time* at which o_i can arrive at node u, and let $ld_i(u) = t_{e_i} - d(u, e_i)$ be the *latest-departure time* at which o_i can depart u to reach e_i at t_{e_i}. Clearly, object o_i can pass through a node u if and only if $ea_i(u) \leq ld_i(u)$. Let us define $\alpha_i(u) = [ea_i(u), ld_i(u)]$ to be the *availability interval* of o_i at node u. Abusing notation slightly, we will treat an availability interval as being synonymous with the set of time instants it includes, and speak of unions and intersections of intervals. We will call an interval $[a, b]$ *empty* whenever $a > b$, and write $[a, b] = \emptyset$. We say u is *reachable* by o_i iff $\alpha_i(u) \neq \emptyset$.

We define the set of *corridor nodes* for o_i to be the set $R_i = \{u \in V \mid \alpha_i(u) \neq \emptyset\}$. These are the nodes reachable by o_i along some route from s_i to e_i that respects the timestamps at s_i and e_i. A meeting is feasible at node u between objects o_i and o_j if their availability intervals overlap at u. That is, we must have $\alpha_i(u) \cap \alpha_j(u) \neq \emptyset$. The longest possible assembly duration at node u for a given set of objects $O' \subseteq O$ is clearly

$$\Delta_{O'}(u) = \bigcap_{o_i \in O'} \alpha_i(u).$$

Let the set of *assembly nodes* for a subset of objects $O' \subseteq O$ be defined as $A_{O'} = \{u \in V \mid \Delta_{O'}(u) \neq \emptyset\}$. Since availability intervals are closed intervals, if an object arrives at a node at precisely the instant that another object departs it, we consider the two objects to have met at that node.

4. QUERY TYPES

The basic queries we would want to support under the tracking model presented above would report (1) the set of corridor nodes for a given object o_i, to discover where o_i could have traveled, and (2) the set of assembly nodes for a given subset of objects $O' \subseteq O$, to discover where these objects could have met. Our model is sufficiently extensible to support other queries of interest by incorporating additional constraints and query objectives.

The first of these extended queries is the $[\gamma, \tau]$-query, which reports the set of nodes at which assemblies of size at least γ and duration at least τ are possible. That is, it returns $\{u \in V \mid \exists O' \subseteq O, |O'| \geq \gamma, |\Delta_{O'}(u)| \geq \tau\}$. This allows for finer control over the sizes of groups or assembly durations which we are interested in tracking.

Other extended queries that we also consider in this model include the [top-$k(\gamma), \tau$] query, which reports the k assembly nodes with the *largest possible assembly sizes* γ of at least some fixed duration τ, and the $[\gamma,$ top-$k(\tau)]$ query, which reports the k assembly nodes with the *longest possible assembly durations* τ of at least some fixed assembly size γ.

5. METHODS

Let $O = \{o_1, o_2, \ldots, o_r\}$, and let object $o_i \in O$ have transit time δ_i across the unmonitored region, as in Sec. 3. Let A_O be the set of assembly nodes for the object set O, given the set of transit times $\{\delta_k\}$. We first describe an algorithm for a $[\gamma, \tau]$-query with $\gamma = |O| = r$ and $\tau = 1$. We shall call this the $[r, 1]$-query. We then extend this approach to show how to evaluate $[\gamma, \tau]$-queries for arbitrary γ and τ, as well as top-k queries.

For simplicity, we assume in the rest of the paper that we have exactly two consecutive records $o_i@(s_i, t_{s_i})$ and $o_i@(e_i, t_{e_i})$ for each object o_i, recording its movement between nodes s_i and e_i. These may represent its entry and exit from an unobserved area, such as in Figure 1. The generalization to many records per object is straightforward.

5.1 The Naïve Solution

The naïve solution we present for $[r, 1]$-queries has two phases: a *search phase* and an *evaluation phase*. The search phase begins with a bidirectional Dijkstra search [23] for each object $o_i \in O$, running forward from the source s_i and backward from the destination e_i, with the region transit time δ_i as the search cutoff bound. Initially, we set $\Delta_O(u) = [\Delta_a^u, \Delta_b^u] = [-\infty, \infty]$ $\forall u \in V$. We also set $ea_i(u) = \infty$, $ld_i(u) = -\infty$ $\forall o_i \in O$. Each forward and backward search then sets $ea_i(u) = t_{s_i} + d(s_i, u)$ and $ld_i(u) = t_{e_i} - d(u, e_i)$, only when $d(s_i, u) \leq \delta_i$ and $d(u, e_i) \leq \delta_i$, respectively. After the bidirectional search for a given object o_i is done, we set $\Delta_O(u) = [\max\{\Delta_a^u, ea_i(u)\}, \min\{\Delta_b^u, ld_i(u)\}], \forall u \in V$. After each bidirectional search, we may also easily establish the set of corridor nodes R_i for the corresponding moving object o_i, according to the definition given in Sec. 3.

A node may lie in the corridor of several objects. During the search phase we also keep count of the total number of objects that can reach a node u. We call this the *reachability number* $\rho(u)$ of node u. We set $\rho(u) = 0$ for all $u \in V$ before beginning the search phase. During the bidirectional search for any object o_i, we increment $\rho(u)$ when node u becomes reachable by o_i.

In the evaluation phase, we process nodes to determine if they are possible assembly nodes for the objects of interest. Since an $[r, 1]$-query requires nodes at which *all* r objects could have met, we process a node iff $\rho(u) = r$. This condition shows that u lies in the corridors for all r objects, but does not guarantee they could have all been simultaneously present at u. Therefore, we check the assembly duration $\Delta_O(u)$ for each node u with $\rho(u) = r$. If $\Delta_O(u) \neq \emptyset$, then u is an assembly node, that is, all objects could meet at u.

For r objects, this method requires exactly $2r$ Dijkstra searches. Hence, in the worst case, a node may be visited by up to $2r$ distinct searches. In Sec. 5.4, we propose an alternate method for speeding up shortest-path computations based on a pre-processing technique. This method requires us to explore every node no more than twice, making for a much faster and more scalable approach overall.

5.2 Arbitrary $[\gamma, \tau]$-Query

We can ensure assembly durations of at least τ by changing the start times t_{s_i} to $t^\tau_{s_i} = t_{s_i} + \tau$. This makes $\delta^\tau = t_{e_i} - t^\tau_{s_i} = t_{e_i} - (t_{s_i} + \tau) = \delta_i - \tau$ before the search phase. Henceforth, we take the transit times to be $\{\delta^\tau_1, \ldots, \delta^\tau_r\}$, the set of adjusted transit times, unless specified otherwise. Now, after the search phase, $ea_i(u) \leq ld_i(u)$ iff $|\alpha_i(u)| \geq \tau$ (based on the original, unmodified start times). Similarly, for any subset of objects O', if $\Delta_{O'}(u) = [a, b]$ and $a \leq b$, then the objects in O' can assemble at u for at least time τ.

Let $\hat{O}^A_u$ be a maximal set of objects that can form an assembly at node u. For u to be an assembly node, we also require $|\hat{O}^A_u| \geq \gamma$. Unlike with the $[r, 1]$-query, we cannot update the assembly duration for a node because we do not know in the search phase which subset(s) of O will satisfy the $[\gamma, \tau]$ condition. Hence, we cannot identify assembly nodes

merely by checking their reachability numbers and assembly durations. Even if $\rho(u) \geq \gamma$, the availability intervals at u for all objects may not overlap, leading to $|\hat{O}_u^A| < \gamma$. To evaluate a general $[\gamma, \tau]$-query, potential nodes are examined during the evaluation phase, after the search phase.

The evaluation phase examines all nodes with reachability number $\rho(u) \geq \gamma$ as potential assembly sites. Let O_u^R be the set of objects for which u is reachable. A brute force method would compute $\Delta_{O'}(u)$ for all subsets $O' \subseteq O_u^R$ with $|O'| \geq \gamma$. However, in the worst case, this is exponential in $|O_u^R|$ and thus impractical. Instead, we use the following *line sweep* algorithm.

5.2.1 The Line-Sweep Algorithm

Figure 4 illustrates the line-sweep algorithm with 10 objects $a, b, \ldots, j$. The values t_o, t'_o denote the *ea*, *ld* timestamps for each object $o \in \{a, b, \ldots, j\}$ at some node u. We start by selecting nodes $\{u \mid \rho(u) \geq \gamma\}$. To determine whether at least γ objects from O_u^R can meet at node u, we sort the *ea* and *ld* timestamps of all objects in O_u^R in increasing order. Then we "sweep the line" from the earliest to the latest timestamp, keeping a counter c of intervals that have started but not ended yet. These are the *live entries*.

The value c counts the number of overlapping intervals at the current position on the line. The maximum value $c_{\max}$ of c gives the maximum number of objects that can meet at node u (six in Figure 4). There may, of course, be many subsets of objects satisfying the assembly size constraint γ. If we want all such subsets, we must continue the line sweep up to the end. If, however, we want to know only whether there is at least one subset of size γ or larger, we can stop the line sweep as soon as $c \geq \gamma$. For example, if $\gamma = 3$ then we can stop at timestamp t_c in Figure 4.

Another strategy for early stopping is to consider the number of objects f that have not yet been encountered in the line sweep. Clearly, if $c + f < \gamma$, then an assembly of γ objects at node u is impossible, and we can stop sweeping. If $\gamma = 8$ in Figure 4, for example, then at timestamp t'_c we have $c = 3$ and $f = 4$, and we can stop sweeping early. This test is useful whether we are looking for all subsets that satisfy the query condition or just a Boolean answer.

5.3 Top-k Queries

Going beyond the standard $[\gamma, \tau]$ query, we consider two types of top-k queries. The first, the [top-$k(\gamma)$, τ] query, returns the assemblies that have the k highest γ values, subject to a given minimum τ value. The second, the [γ, top-$k(\tau)$] query, returns the assemblies having the k highest τ values, subject to a given minimum γ value.

We will maintain priority queues Q^γ and Q^τ, for the [top-$k(\gamma)$, τ] and [γ, top-$k(\tau)$] queries, respectively, in the evaluation phase of the standard $[\gamma, \tau]$ query. Nodes with a certain reachability number (depending on the query type, as detailed below) are evaluated, and the priority queue updated if the current top-k candidate list can be improved.

5.3.1 [*top-$k(\gamma)$*, τ] *Queries*

Let $\chi_{\min}^{Q^\gamma}$ be the minimum assembly size in the priority queue Q^γ at a given time. For the [top-$k(\gamma)$, τ] query, a node u is always evaluated if $|Q^\gamma| < k$. If Q^γ has at least k elements, however, the node is evaluated iff $\rho(u) > \chi_{min}^{Q^\gamma}$.

We use the same line sweep algorithm described above to evaluate a node, but we cannot use the same early stopping conditions described previously. This is because whenever the maximum number of objects that can meet at node u exceeds the minimum assembly size in Q^γ, we must update Q^γ by deleting the node with minimum assembly size and inserting the node u and its maximum assembly size. We cannot compute the maximum assembly size at node u without completing the line sweep algorithm for u.

Regardless, we can stop sweeping as soon as we determine that the node cannot improve the current top-k list. One early stopping criterion we use is $c + f \leq c_{\max}$, which means continuing sweeping will not improve the maximum assembly size we have already found for this node. Even if we stop early with this condition we could still have $c_{\max} > \chi_{min}^{Q^\gamma}$, and we will have to update Q^γ.

To see this, consider a [top-$k(\gamma)$, τ] query and $\chi_{min}^{Q^\gamma} = 5$. At time t_g in Figure 4, we have $c + f = 6$, which will not improve the current best value for this node. We hence stop, but we still update Q^γ, and $\chi_{min}^{Q^\gamma}$ becomes 6. Another early termination condition is $c + f \leq \chi_{min}^{Q^\gamma}$, which implies that an assembly larger than $\chi_{min}^{Q^\gamma}$ cannot be achieved from the current position of the sweeping line. In this case, we need not update Q^γ. As an example, consider the above query with $\chi_{min}^{Q^\gamma} = 9$. At time t'_f we have $c + f = 9$. So we stop at this point and do not update Q^γ.

5.3.2 [γ, *top-$k(\tau)$*] *Queries*

To evaluate a [γ, top-$k(\tau)$] query with a minimum assembly size γ, we can not adjust the travel times to account for a minimum assembly duration (as suggested in Sec. 5.2), since the desired duration is not known beforehand. We evaluate a node u if $\rho(u) \geq \gamma$, using a modified line sweep algorithm.

Since we aim to maximize the assembly duration subject to a minimum assembly size γ, we need not consider assemblies larger than γ. Thus, as we sweep the line we record the start of an assembly as soon as the number of live objects reaches γ. We continue sweeping as long as the assembly size is at least γ, and record the end of the assembly when the number of live objects drops below γ at the end of an interval.

The assembly duration is computed from the start time of the γ^{th} interval. To do this, we must keep track of the γ^{th} interval during the line sweep, as intervals start and end. We maintain a list of live intervals sorted by their start time, as well as a pointer to the γ^{th} interval in this list. When an interval with an earlier start time than that of current γ^{th} interval ends, we advance the pointer to the γ^{th} interval by one (when there are more than γ intervals in the list of live intervals). If an interval that started after the γ^{th} interval ends we simply delete it from the live list without modifying the pointer.

Consider a [γ, top-$k(\tau)$] query with $\gamma = 3$ in the example of Figure 4. At time t_c we have 3 live intervals $\{a, b, c\}$. So c becomes the γ^{th} interval and t_c is the start time of an assembly of at least γ objects. Then intervals d, e, and f start and interval f ends but the γ^{th} entry remains the same (and so does the start time of the assembly). Next, interval a ends at t'_a and the pointer to the γ^{th} interval is moved forward to d since a is within the first γ intervals. When the end of an interval requires updating the γ^{th} interval pointer, we compute the current assembly duration and, if necessary, update the maximum assembly duration found so far, before updating the γ^{th} interval pointer.

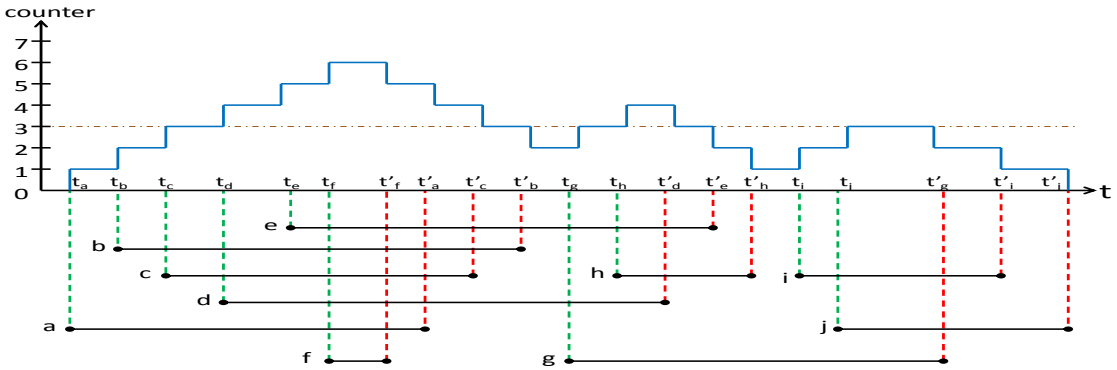


Figure 4: The Line-Sweep Algorithm. There are ten objects $\{a, b, c, d, e, f, g, h, i, j\}$ with the availability intervals $[t_a, t'_a]$, $[t_b, t'_b]$, etc. at the same node u.

At time t'_c the set of live intervals is $\{b, d, e\}$. Next, the end of the interval b reduces the set of live intervals to size $2 < \gamma$ and the γ^{th} interval pointer becomes invalid. At this point the assembly duration subject to at least γ objects is $t'_b - t_e$. As sweeping proceeds, assemblies of size at least γ start at times t_g and t_j and end at t'_e and t'_g, with duration $t'_e - t_h$ and $t'_g - t_j$. We take the longest of these assemblies and update Q^τ if necessary. We note that we not only find the longest assembly duration as we sweep, but we also know the object IDs of those who were present in each assembly. This line-sweep algorithm can also therefore be used to find all possible assemblies of size at least γ.

5.4 A Solution Using Contraction Hierarchies

We now show how the search phase can be significantly improved by combining our proposed method with the *path contraction* idea, introduced in Contraction Hierarchies [2]. A contraction replaces a path between two nodes of a graph with a so-called *shortcut* edge, preserving the distance between these nodes. CH-based methods are currently some of the fastest and most flexible approaches for answering shortest path queries and their variations on road networks. [24, 25].

Once again, we begin by describing the search phase for the $[r, 1]$-query. The evaluation phase is the same for both the Dijkstra-based and the CH-based solutions and the extension to other queries is done by using the line-sweep algorithm in the same way. However, we will show that for the CH-based methods, the evaluation phase can be carried out concurrently with the search phase. That is, a node is evaluated when it is accessed during the search phase. We start with a brief description of CH before presenting our methods in detail.

To use the CH-based method we first need to pre-process the graph representing our road network (note that this pre-processing only needs to happen once, after which time all subsequent queries may exploit it). The CH pre-processing has two phases: (1) ordering (or ranking) the nodes of the graph, and (2) contracting nodes one by one, in that order. Any arbitrary ordering would preserve the correctness of the algorithm but some orderings improve the performance more than others (see [2] for details).

5.4.1 CH-Based Shortest Path Computation

The nodes in the graph G are first arranged in some order $\phi : V \to \{1, \ldots, |V|\}$, and contracted in this order. A node v is contracted by adding shortcuts to bypass the node if there is some unique shortest path through it. For example, if there is a pair of incoming and outgoing edges (u, v) and (v, x) respectively, s.t. $\min\{\phi(u), \phi(x)\} > \phi(v)$ and (u, v, x) is a unique shortest path, then a shortcut edge (u, x) with weight $w(u, v) + w(v, x)$ is introduced. The contracted node remains in the graph, but the new shortcut edges are introduced to bypass this node during shortest path computation. After all nodes are contracted, the graph contains a superset of its original edges, since we will have $G = (V, E \cup E^S, w)$ where E^S is the set of newly-introduced shortcut edges.

To speedup the shortest path computation from s to e, where $s, e \in V$ in the contracted graph, the usual bidirectional Dijkstra algorithm must be modified. The forward search from s is run in the subgraph $G_\uparrow = (V, E_\uparrow)$, where $E_\uparrow = \{(u, v) \in E \cup E^S \mid \phi(u) < \phi(v)\}$ and the backward search from e is run at the same time in $G_\downarrow = (V, E_\downarrow)$, where $E_\downarrow = \{(u, v) \in E \cup E^S \mid \phi(u) > \phi(v)\}$. A candidate shortest path cost is updated when the two searches meet at some node in the graph. The search in a particular direction may be stopped when the minimum cost in the priority queue for that direction is higher than the minimum candidate path cost seen so far.

This approach provides significant speedups over standard Dijkstra search by exploring a much sparser overall subgraph (bypassing lower-ranking nodes by using the added shortcut edges). Moreover, it remains correct because the CH pre-processing ensures that there now exists a shortest path between any two nodes that first strictly increases in node rank and then strictly decreases in node rank (a so-called *weakly-bitonic shortest path* [24]), which these two focused searches are guaranteed to find when intersected.

5.4.2 Naïve CH-Based Corridor Computation

Our goal, however, is to compute corridor nodes for each (s_i, e_i) pair, not just the shortest paths. We can then compute the set of assembly nodes, which is a subset of the intersection of the corridor sets.

A two-phase CH-based algorithm for finding corridor nodes for a given (s_i, e_i) pair and duration δ_i^τ is presented in [24]. The first phase (upward search) runs a forward search from s_i in $G_\uparrow$ and a backward search from e_i in $G_\downarrow$. In the second

phase (downward search), nodes touched in the first phase are accessed in decreasing ϕ order and downward edges are expanded to find the corridor nodes $v \in R_i$ (see Fig. 5 for conceptual illustration). Both the upward and downward searches are bounded by a threshold (the transit time δ_i^τ, in our case). The intuition is that by the time a node v is encountered during the downward search phase, we are guaranteed to know the shortest-path cost from s_i to v as well as from v to e_i by this same weakly-bitonic shortest path property, and can thus easily determine if $v \in R_i$.

To compute corridor nodes for multiple (s_i, e_i) pairs, a straightforward way would therefore simply be to use the above algorithm once for each (s_i, e_i) pair.

5.4.3 Optimized CH-Based Corridor Computation

We can exploit the structural properties of CH even further, however, to compute the corridor nodes for all (s_i, e_i) pairs simultaneously. We run a modified version of the above two phase algorithm just once, bounded by thresholds δ_i^τ.

The extension (described in detail below) is to process *all* objects simultaneously within the same type of two-phased search: one upward-search phase that establishes the strictly-increasing-rank portions of a shortest path forward from *all* s_i toward some v and backward from *all* e_i toward some v (e.g., the red subpaths in Fig. 5), followed by one downward-search that completes the shortest paths to each node v that falls within at least one object's corridor by establishing the strictly-decreasing-rank portions of those same shortest paths (e.g., the blue subpaths in Fig. 5).

The intuition here (formalized in Theorem 1) is the same as it was for the single-object corridor search in [34]; only now, we are able to guarantee that, by the time a node is removed from the downward search queue, we have established all relevant shortest-path information for *all* objects *simultaneously*. Thus, at that point during the search, we can immediately evaluate whether that node belongs in the solution or not. The algorithm is as follows.

For vertex v, let $d_{s_i}(v)$ and $d_{e_i}(v)$ be the times to traverse the quickest path found searching forward from s_i and backward from e_i, respectively. Consider the following extension of the above corridor search algorithm.

Start with an increasing rank order queue, *incQ*, holding $s_i, e_i, \forall_{o_i \in O}$. Before the search, set $d_{s_i}(s_i) = d_{e_i}(e_i) = 0$, and $d_{s_i}(v) = d_{e_i}(v) = \infty$ where $v \neq s_i, e_i$. When a node v is accessed from *incQ* it is inserted into a decreasing rank order queue *decQ* if $d_{s_i}(v) \leq \delta_i^\tau$ or $d_{e_i}(v) \leq \delta_i^\tau$ for some $o_i \in O$. Now, v's upward edges are expanded as follows.

For all outgoing $(v, x) \in E \cup E^S$, such that $\phi(v) < \phi(x)$, set $d_{s_i}(x) = \min\{d_{s_i}(x), d_{s_i}(v) + w(v, x)\}$ for all $o_i \in O$. If $x \notin$ *incQ* and $d_{s_i}(x) \leq \delta_i^\tau$ for some $o_i \in O$, add x to *incQ*. For all incoming $(u, v) \in E \cup E^S$, such that $\phi(v) < \phi(u)$, set $d_{e_i}(u) = \min\{d_{e_i}(u), d_{e_i}(v) + w(u, v)\}$ for all $o_i \in O$, and if $u \notin$ *incQ* and $d_{e_i}(u) \leq \delta_i^\tau$ for some $o_i \in O$, add u to *incQ*. The upward search ends when *incQ* becomes empty.

After the upward search, we access nodes in *decQ* in decreasing rank order. When a node v is accessed from *decQ* its edges are expanded downward as follows. For all outgoing $(v, x) \in E \cup E^S$, such that $\phi(v) > \phi(x)$, set $d_{s_i}(x) = \min\{d_{s_i}(x), d_{s_i}(v) + w(v, x)\}$ for all $o_i \in O$. If $x \notin$ *decQ* and $d_{s_i}(x) + d_{e_i}(x) \leq \delta_i^\tau$ for some $o_i \in O$, add x to *decQ*. For all incoming $(u, v) \in E \cup E^S$, such that $\phi(v) > \phi(u)$, set $d_{e_i}(u) = \min\{d_{e_i}(u), d_{e_i}(v) + w(u, v)\}$ for all $o_i \in O$. If $u \notin$ *decQ* and $d_{s_i}(u) + d_{e_i}(u) \leq \delta_i^\tau$ for some $o_i \in O$, add u to *decQ*. The downward search ends when *decQ* is empty.

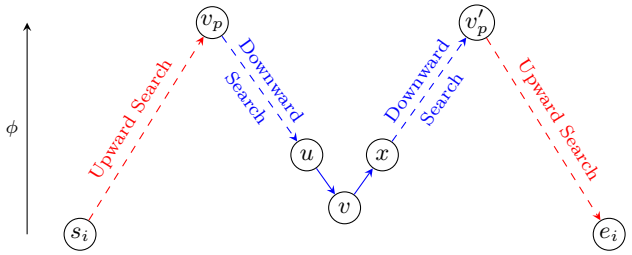


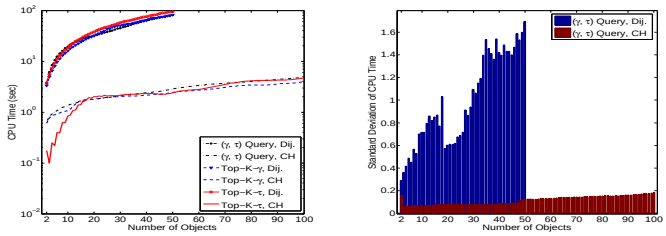
Figure 5: Weakly-bitonic shortest paths computed forward from some s_i to v and backward from some e_i to v for some $v \in R_i$ during the two-phased search.

THEOREM 1. *When a node is accessed from decQ in the downward search phase, $d_{s_i}(v) = d(s_i, v)$ and $d_{e_i}(v) = d(v, e_i)$ for all $o_i \in O$.*

PROOF. Between any pair of connected nodes v_1 and v_q after preprocessing, there must exist a *weakly-bitonic* [24] shortest path $P = \langle v_1, \ldots, v_p, \ldots, v_q \rangle$, such that $\phi(v_1) < \ldots < \phi(v_{p-1}) < \phi(v_p)$ and $\phi(v_p) > \phi(v_{p+1}) > \ldots > \phi(v_q)$. (The path is *weakly* bitonic since $v_p = v_1$ or $v_p = v_q$ may be true.) Let v be any node removed from *decQ* in the second search phase. Now, for every $o_i \in O$, there must exist weakly-bitonic shortest paths from s_i to v and from v to e_i (as in Fig. 5). Let the first of these be $P = \langle s_i = v_1, \ldots, v_p, \ldots, v_q = v \rangle$ from s_i to v, such that v_p is the highest-ranking node in P. In the first, upward search phase, paths are processed forward from s_i in increasing node rank order, and thus, in shortest-path order up to v_p, correctly establishing the costs along the subpath $P_1 = \langle v_1, \ldots, v_p \rangle$. In the second, downward search phase, the search processes paths in decreasing node rank order, and also thus, in shortest path order for the remainder of the path, correctly establishing the costs along the remaining subpath $P_2 = \langle v_p, \ldots, v_q \rangle$. By the time v is removed from *decQ*, $d_{s_i}(v_q) = d_{s_i}(v) = d(s_i, v)$ is therefore correctly established. A symmetric argument holds for the weakly-bitonic shortest path from v to e_i, ensuring $d_{e_i}(v) = d(v, e_i)$. □

After removing a node v from *decQ* in the second (downward) search phase, we may immediately proceed to evaluate v as a candidate for inclusion in the query solution, exactly as before. This is because Theorem 1, guarantees that we already have all necessary information required to establish v's availability intervals for all relevant objects. Furthermore, any node v that belongs to the corridor of at least one object $o_i \in O$ will be guaranteed to be removed from *decQ* for evaluation. This is because all nodes along the weakly-bitonic shortest paths forward from s_i and backward from e_i to v will, by definition, be reachable from said terminals within the requisite threshold δ_i^τ, which is sufficient for inclusion in both the *incQ* and *decQ* priority queues during the search. Therefore, we ensure that any valid candidate nodes will be properly evaluated.

Our experiments show that our method is faster than the naive Dijkstra-based method. The two methods have equivalent evaluation phases, but the search phase of our method has a better worst-case asymptotic time complexity overall. The Dijkstra-based method requires $O(r(m + n \log n))$



(a) (b)

Figure 6: (a) Query Evaluation Time vs Number of Objects. (b) Standard Deviation of Query Evaluation Time vs Number of Objects.

Parameter	Minimum	Maximum	Default
ϵ	0.1	1	0.5
r	2	100	20
k	10	100	100
γ	$r/10$	r	$r/2$
τ	$\epsilon d_{\min}/10$	$\epsilon d_{\min}$	$\epsilon d_{\min}/10$

Table 2: Default parameter Values.

time (two distinct Dijkstra searches per object). Our CH-based search reduces the total worst-case time complexity to $O(rm + n \log n)$, as each explored node is only sorted by rank at most twice and we relax costs for each object at most $O(m)$ times. Our advantage becomes sharper as r grows.

As with most CH-based search algorithms, our method benefits most significantly in practice from the sparse search spaces afforded by this exploitation of shortcut edges and weakly-bitonic shortest paths.

6. EXPERIMENTAL EVALUATION

We evaluated our algorithms on a real-world dataset, which was the combined road network data for California and Nevada. This combined network has 1,890,815 nodes and 4,630,444 edges. The values of the various parameters used in our experiments appear in Table 2.

An edge was weighted by the travel time over that road segment. We generated queries by randomly selecting a source-destination (s, e) pair for each moving object. Since the travel time δ_i between a pair (s_i, e_i) had to be at least $d_i = d(s_i, e_i)$, we set $\delta_i = (1+\epsilon)d_i$. All experiments were run on an Intel Xeon 3.0GHz quad core processor running Linux 2.6.18 with 8GB of main memory. Below we report the CPU time for all query evaluations (the road networks used in the experiments as well the structures needed to maintain the observed objects can all fit in the available main memory).

We first examine the effect of the number of moving objects r being tracked on the query evaluation time for all three types of queries. The results are shown in Figure 6. We varied the number of objects from 2 to 100 for the CH-based methods. For Dijkstra-based methods we stopped the experiments after 50 objects since these methods do not scale, and the running times became unmanageably large. For each value of r, we averaged the query time over 100 random queries. Thus, for 10 objects, we had 100 different sets of moving objects, each of size 10.

These experiments used $\epsilon = 0.5$, $\gamma = r/2$, $\tau = \epsilon d_{min}/10$.

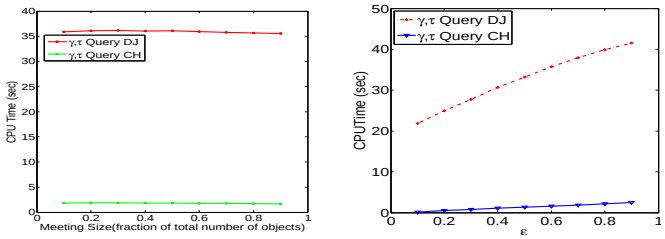
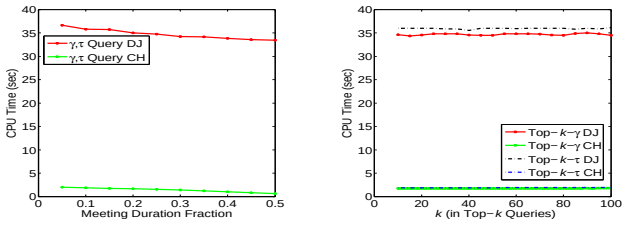


Figure 7: (a) Min γ vs Query Evaluation Time for (γ, τ)-queries. (b) The effect of the ϵ parameter on the Query Evaluation Time.



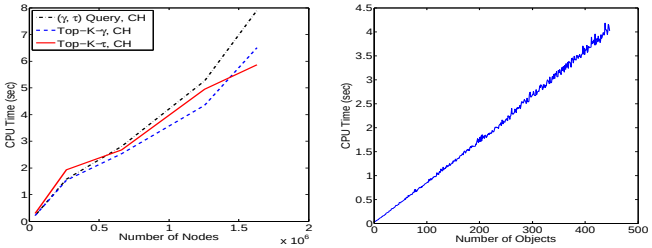
(a) (b)

Figure 8: (a) Min τ vs Query Evaluation Time for (γ, τ)-queries. (b) Query Evaluation Time for Top-k queries.

That is, at least half the moving objects were required to spend at least one tenth of their minimum additional travel time ($\epsilon d_{\min}$) in an assembly. The query times are presented in Figure 6(a), which uses a logarithmic scale. Our CH-based methods run 1-2 orders of magnitude faster and rise much more slowly than do the Dijkstra-based methods. The query times for top-k queries are very similar to those for $[\gamma, \tau]$-queries. This implies that the additional work of maintaining the top-k queues is quite minimal. Note that, the standard deviation of query times for the $[\gamma, \tau]$-query using the Dijkstra-based approach is much higher than that for the CH-based method, as depicted in Figure 6(b). That is, the query times for the Dijkstra-based approach are less predictable.

Figure 7(a) examines the effect of the minimum assembly size γ on query evaluation time for $[\gamma, \tau]$-queries. We varied the minimum assembly size from 10% to 90% of the total number of objects, setting $\epsilon = 0.5$, $\tau = \epsilon d_{min}/10$ and the number of objects to 20. Clearly the CH-based approach outperforms the Dijkstra-based method. Because of the drastic difference in query evaluation time we depict the performance of each method separately, in Figures 10(a) and (b) (see Appendix B). Since a node is not evaluated if its reachability number is less than γ, a higher minimum assembly size lowers the number of nodes evaluated. Hence, we expect the query evaluation time to decrease with γ. This trend is clear at larger γ values. We note that the query time first increases before it starts to plummet. This is because for small γ, there are many nodes with reachability number higher than γ. Above a certain γ value, however, the query time falls quickly.

Next, we study the effects on $[\gamma, \tau]$-queries of the parameter ϵ, which determines the additional travel time we allow beyond the fastest travel time. Since the size of the search



(a) (b)

Figure 9: (a) The effect of network size (number of nodes) on the query evaluation time.

space for our algorithms depends on the value of ϵ, we varied ϵ from 0.1 to 1.0 in steps of 0.1. For these experiments we set the number of objects to 20, $\gamma = r/2$, $\tau = \epsilon d_{\min}/10$. The average query evaluation time is computed by running 100 random queries for each value of ϵ. As Figure 7(b) shows, the query time rises very slowly with ϵ for the CH-based method, but very sharply for the Dijkstra-based methods. This is because although the search space grows quadratically with ϵ, the CH-based approach does not touch all nodes in the search space and is less affected. Since top-k queries touch same number of nodes as $[\gamma, \tau]$-queries, we expect the effects of ϵ on them to be similar.

Next we examine the effect of the minimum assembly duration τ on the query evaluation time for $[\gamma, \tau]$-queries. For these experiments we varied the minimum assembly duration from 10% to 50% of the additional travel time, setting $\epsilon = 0.5$, $\gamma = r/2$, and number of objects to 20 for this experiment. The results appear in Figure 8(a); as the value of minimum τ increases, the query evaluation time decreases. Similarly with the previous min γ experiment, a larger minimum assembly duration results in the evaluation of fewer nodes, resulting in a lower query time.

We proceed with the experimental evaluation of top-k queries. The results appear in Figure 8(b) for values of k ranging from 10 to 100. For these experiments we set $\epsilon = 0.5$, $\gamma = r/2$, $\tau = \epsilon d_{min}/10$ and the number of objects to 20. The query time for both [top-$k(\gamma), \tau$] and [γ, top-$k(\tau)$] queries remains almost constant as k increases. This is expected, as the value of k affects the evaluation phase only, but *not* the search phase, and the search phase accounts for the bulk of the query processing time.

In Figure 9(a) we examine the effect of the road network size on the query evaluation time of the CH-based methods. The X-axis depicts the number of nodes in the network being considered. This graph, in conjunction with earlier results, such in Figure 6, shows that our CH-based method scales very well. In these experiments, we chose the road network contained within a rectangular region of the California-Nevada road network, and increase the number of nodes by enlarging the rectangle size. The experiment was run for 100 objects and using the default values of all other parameters. The number of nodes varied from a few thousands to more than 1.5 million. All the CH-based algorithms behave similarly showing a close to linear increase in the query evaluation time as the number of nodes increases.

Finally, we examined how the CH-based algorithms scale with respect to the number of observed objects. For this experiment we used a rectangular region from the California-Nevada road network that has $265K$ nodes and $637K$ edges. Using default values of all other parameters, we increased the number of observed objects up to 450 objects. Figure 9(b) depicts the results for the $[\gamma, \tau]$-query (the top-k queries behaved similarly); clearly we can see that the query evaluation time for the CH-based method scales linearly with the number of objects.

7. CONCLUSIONS

We introduced the novel and important class of *assembly* queries; such queries appear in the form of either "assembly discovery" (determine whether two or more moving objects could have had a meeting within a region of interest) or "assembly planning" (arrange for meetings for a group of friends visiting a city without violating their remaining schedules). We provide efficient solutions to such queries given incomplete trajectory information, using knowledge of the topology of the underlying transportation network. In particular, we consider a number of variations where the query can specify meeting duration or meeting size or seek top-k nodes (meeting locations) based on these attributes. We present a formal model for the general problem and prove the correctness of our algorithm. Furthermore, we show how to utilize a preprocessing method based on Contraction Hierarchies to gain orders of magnitude speed up over the naïve Dijkstra-based methods.

There are various interesting open problems that the assembly queries lead to. In the present paper we assume that the weight of an edge (travel time) is static; we are examining ways to extend our algorithms when considering dynamic traffic patterns (for example the weight of an edge depends on the time of day it is traveled; for this scenario extensions to the CH-based approach will be needed). We are currently extending our algorithms to address reachability queries (transitive communications) within this framework. We are also examining whether parallelism can be used to further improve query processing.

8. ACKNOWLEDGMENTS

This research was partially supported by the National Science Foundation grants IIS-1527984 and CNS-1330110.

9. REFERENCES

[1] D. Pfoser and C. S. Jensen, "Capturing the uncertainty of moving-object representations," in *Symp. Adv. in Spatial Databases*, 1999, pp. 111–132.

[2] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: faster and simpler hierarchical routing in road networks," in *Proc. 7th Intl. Wksp. on Experimental Algorithms (WEA)*, 2008.

[3] G. Kollios, D. Gunopulos, and V. J. Tsotras, "On indexing mobile objects," in *Proc. 18th ACM PODS*, 1999, pp. 261–272.

[4] S. Chen, B. C. Ooi, K.-L. Tan, and M. A. Nascimento, "ST2B-tree: A self-tunable spatio-temporal B+-tree index for moving objects," in *Proc. ACM SIGMOD Conference*, 2008, pp. 29–42.

[5] J. M. Patel, Y. Chen, and V. P. Chakka, "Stripes: An efficient index for predicted trajectories," in *Proc. ACM SIGMOD Conference*, 2004, pp. 635–646.

[6] J. Ni and C. V. Ravishankar, "Indexing spatiotemporal trajectories with efficient polynomial

approximation," *IEEE Trans. on Knowl. and Data Eng.*, vol. 19, no. 5, pp. 663–678, 2007.

[7] V. T. De Almeida and R. H. Güting, "Indexing the trajectories of moving objects in networks," *Geoinformatica*, vol. 9, no. 1, pp. 33–60, Mar. 2005.

[8] J. Ni, C. V. Ravishankar, and B. Bhanu, "Probabilistic spatial database operations," in *Proc. 8th SSTD*, 2003, pp. 140–158.

[9] G. Trajcevski, O. Wolfson, K. Hinrichs, and S. Chamberlain, "Managing uncertainty in moving objects databases," *ACM Trans. Database Syst.*, vol. 29, no. 3, pp. 463–507, Sep. 2004.

[10] G. Trajcevski, "Probabilistic range queries in moving objects databases with uncertainty," in *Proc. 3rd ACM MobiDE*, 2003, pp. 39–45.

[11] T. Emrich, H.-P. Kriegel, N. Mamoulis, M. Renz, and A. Zufle, "Querying uncertain spatio-temporal data," in *Proc. IEEE ICDE*, 2012, pp. 354–365.

[12] G. Trajcevski, A. N. Choudhary, O. Wolfson, L. Ye, and G. Li, "Uncertain range queries for necklaces." in *11th MDM Conf.*, 2010, pp. 199–208.

[13] K. Zheng, G. Trajcevski, X. Zhou, and P. Scheuermann, "Probabilistic range queries for uncertain trajectories on road networks," in *Proc. EDBT*, 2011, pp. 283–294.

[14] G. Trajcevski, R. Tamassia, H. Ding, P. Scheuermann, and I. F. Cruz, "Continuous probabilistic nearest-neighbor queries for uncertain trajectories," in *Proc. EDBT*, 2009, pp. 874–885.

[15] J. S. Greenfeld, "Matching gps observations to locations on a digital map," in *Transportation Research Board 81st Annual Meeting*, 2002.

[16] K. Zheng, Y. Zheng, X. Xie, and X. Zhou, "Reducing uncertainty of low-sampling-rate trajectories," in *Proc. IEEE ICDE*, no. 1144-1155, 2012.

[17] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang, "Map-matching for low-sampling-rate gps trajectories," in *ACM GIS*, 2009, pp. 352–361.

[18] J. Yuan, Y. Zheng, C. Zhang, X. Xie, and G.-Z. Sun, "An interactive-voting based map matching algorithm," in *11th MDM Conf.*, 2010, pp. 43–52.

[19] J. Krumm and E. Horvitz, "Predestination: Inferring destinations from partial trajectories," in *Ubicomp*, 2006, pp. 243–260.

[20] J. Krumm, R. Gruen, and D. Delling, "From destination prediction to route prediction," in *Journal of Location Based Services*, vol. 7, no. 2, 2013.

[21] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Alternative routes in road networks," in *ACM JEA*, vol. 18, 2013.

[22] R. Bader, J. Dees, R. Geisberger, and P. Sanders, "Alternative route graphs in road networks," in *TAPAS Conf., LNCS*, vol. 6595, 2011, pp. 21–32.

[23] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Numerische Mathematik*, vol. 1, no. 1, 1959, pp. 269–271.

[24] M. N. Rice and V. J. Tsotras, "Parameterized algorithms for generalized traveling salesman problems in road networks," in *ACM GIS*, 2013, pp. 114–123.

[25] R. Geisberger, M. N. Rice, P. Sanders, and V. J. Tsotras, "Route Planning with Flexible Edge Restrictions," *ACM JEA*, vol. 17, no. 1, 2012.

APPENDIX

A. Availability Intervals on Edges.

So far we have discussed the concepts of availability intervals, corridors, and assemblies with respect to nodes only, not along edges. However, it is likely that objects would actually meet at a location somewhere along the midspan of a road (edge), e.g., at a restaurant, mall, gas station, etc.

There are two possible approaches to address this scenario. One is that, if we know what midspan edge locations are of interest to us a priori, then we can easily subdivide their corresponding "containing" edges by making these locations proper nodes in the graph as well. Then our current model "just works" as-is. However, if dealing with a lot of possible locations of interest, this can potentially explode the size of the graph. Therefore, another perhaps-more-desirable and flexible approach is to simply use the existing earliest-arrival and latest-departure information that we have already calculated from our simpler node-based model to interpolate information for a more detailed model along the edges. For example, for any position p in the range $[0, 1]$ along an edge $(u, v) \in E$, we can calculate its availability interval as:

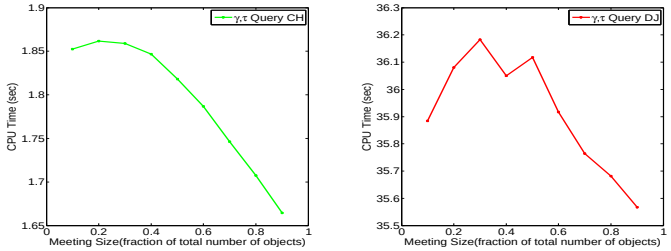
$$[ea_i(u) + p \cdot w(u, v),\ ld_i(v) - (1 - p) \cdot w(u, v)]$$

Note that this assumes, if the traveler enters the edge (u, v) at u, that they must also exit the edge at v. However, this is not always the case, especially for meeting locations with, e.g., parking lots, such as restaurants, gas stations, etc. In these cases, the traveler might enter the edge at u, travel to position p to meet, then depart p and head back to u (in the opposite of the direction they came, along the opposing edge (v, u)), effectively completing a "u-turn". In order to accommodate this more flexible and realistic possibility, we could assume that the traveler could have done either (completing the traversal of (u, v) or doing a u-turn), giving us the (potentially larger) availability interval of:

$$[ea_i(u) + p \cdot w(u, v), max\{ld_i(v) - (1 - p) \cdot w(u, v),\ ld_i(u) - p \cdot w(v, u)\}]$$

Note also that not all edges will support such u-turn scenarios (e.g., roads with medians), so we can be selective as to which edges for which we allow this type of availability interval to occur.

B. Additional Figures.



(a) (b)

Figure 10: Detailed Min γ vs Query Evaluation Time for the (γ, τ)-query: (a) CH-based approaches, (b) Dijkstra-based approaches.