Roads, Codes, and Spatiotemporal Queries

Sandeep Gupta, Swastik Kopparty, Chinya Ravishankar
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521
{sandeep,swastik,ravi}@cs.ucr.edu

ABSTRACT

We present a novel coding-based technique for answering spatial and spatiotemporal queries on objects moving along a system of curves on the plane such as many road networks. We handle join, range, intercept, and other spatial and spatiotemporal queries under these assumptions, with distances being measured along the trajectories. Most work to date has studied the significantly simpler case of objects moving in straight lines on the plane. Our work is an advance toward solving the problem in its more general form.

Central to our approach is an efficient coding technique, based on hypercube embedding, for assigning labels to nodes in the network. The Hamming distance between codes corresponds to the physical distance between nodes, so that we can determine shortest distances in the network extremely quickly. The coding method also efficiently captures many properties of the network relevant to spatial and spatiotemporal queries. Our approach also yields a very effective spatial hashing method for this domain. Our analytical results demonstrate that our methods are space- and time-efficient.

We have studied the performance of our method for large planar graphs designed to represent road networks. Experiments show that our methods are efficient and practical.

1. INTRODUCTION

In recent years, technologies such as the Global Positioning System have greatly improved our abilities to track moving objects. Correspondingly, there has been increased interest in spatiotemporal queries, or queries over time-dependent position information of moving objects. In its most general form, this problem deals with range and join queries on spatial and temporal attributes of objects moving along arbitrary trajectories in $\mathbb{R}^n$. As we discuss in greater detail in Section 3, however, this general problem is extremely complex, and the literature has focused on much simpler versions. The most common formulation addresses selection queries on objects moving in straight lines in $\mathbb{R}^2$.

In practice, however, linear motion is more the exception than the rule. For example, applications as diverse as traffic congestion control and tactical battlefield situations require answers to spatiotemporal queries on objects moving along highly non-linear networks of roads. An additional complication in such applications is that distances between objects are frequently required to be specified *along the roads*, and not as the Euclidean separations in $\mathbb{R}^2$, as in most current work. Range queries typically have the form "Find all objects that are in region R during a time interval $[t_{q1}, t_{q2}]$", "Find all objects that are within distance d of object o_1", or "Find all objects that are encountered by object o_1 in the time interval $[t_{q1}, t_{q2}]$". Typical join queries have the form "Find all pairs of objects which pass a common waypoint", or "Find all pairs of objects that are within distance d of one another at a given time instant t_q", and typical intercept queries have the form "Find the time that it will take for object o_1 to catch o_2".

1.1 Our work

In this paper, we pick a significantly more general version of this problem, and address spatiotemporal queries on objects moving along the edges of certain class of planar graphs. We have found these graphs to be reasonable models of many kinds of road networks, and quite useful in many real situations. Our approach is based on a novel method for assigning codes to nodes in such graphs, which allows us to compute the distance between two nodes as the Hamming distance between the corresponding codewords. The coding is accomplished by embedding the planar graph into a higher-dimensional hypercube, using techniques based on the work in [3].

Road networks are static, so assigning these codes during a preprocessing phase is a reasonable strategy. While we focus in this paper on a class of spatiotemporal queries, we believe the methods we present are of more general applicability.

The rest of the paper is organized as follows. In Section 2 we give an overview of the problem. In Section 3 we describe work on problems that are similar in flavor to our problem. Then, in Section 4 we explore the intricacies of the problem and motivate the scheme that we ultimately develop. In Section 5 we describe our own scheme for handling large amount of distance data. In Section 6 we describe how our scheme can be used to answer a vast range of spatial and spatiotemporal queries on road networks efficiently. Then, in Section 7 we investigate the space and time requirements of our scheme, both theoretically and experimentally. Finally, in Section 8 we conclude with open problems and possible future work.

2. PROBLEM FORMULATION

Consider a weighted graph $G = (V, E, W)$ on n nodes. V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $W : E \to \mathbb{Z}^+$ is a set of integer edge weights.

G represents a road network or some similar system of trajectories. The edges in E are curves in $\mathbb{R}^2$, and the weight of an edge measures the cost of traversing it, and may correspond, for example, to its length. We are given a set $O = \{o_1, o_2, \dots, o_r\}$ of objects, with object o_i traveling from its source node s_i to its destination node d_i, along a prespecified path in G, at speed u_i. In general, we allow the speed to be a function $u_i(t)$ of time.

We address a broad class of spatiotemporal queries in this environment by reducing the problem to that of finding the shortest distance between two nodes on a planar graph. We develop a method based on the work in [3] to assign codes $c(v_i)$ to nodes $v_i \in V$, so that the distance between nodes v_j and v_k is simply the Hamming distance [4] between $c(v_j)$ and $c(v_k)$.

We allow the edges connecting vertices to be general curves, rather than merely straight lines, as in most previous work. We choose parametric representations to represent object motion in the Cartesian plane, using the distance along the curve as the primary parameter. This approach solves two problems. First, the parametric representation of a curve is often cleaner than a Cartesian representation of the form $f(x, y) = 0$. Second, distance is readily computed as the velocity $u_i(t)$ integrated over time. Conversely, time is computable from distance. Such interconvertability makes spatio-temporal queries cleaner to deal with.

2.1 About Our Model

In order to make the problem tractable, we make some assumptions about the planar graph. We say that a planar graph has isometric cycles if the shortest path between two vertices on the same given interior face is along the edges bounding the face. We generalize this condition to a slightly more technical condition, and in Section 5.2, we describe an encoding technique that is applicable to planar graphs that satisfy this generalized isometry property.

Since we are dealing with trajectories in $\mathbb{R}^2$, the edge weights of the graph that model this set of trajectories will not be completely arbitrary. It is reasonable to expect the distance along a curve between two vertices in the underlying system of curves to correlate generally with the Euclidean distance between those vertices. We express this constraint formally as the generalized isometry property.

Finally our model assumes integral edge weights, but in general, they will be non-integral. However, it is possible to approximate these weights with integers by appropriately scaling them, or by introducing user-defined approximation bounds. Finally, we observe that it is very reasonable to assume that objects move from source to destination along the shortest path. If an object o_i is required to pass through a way-point w that is not on this shortest path, we can decompose its trajectory into two shortest-path segments, one from s_i to w and the second from w to d_i. We discuss this issue further in Section 6.

3. RELATED WORK

Research in spatiotemporal queries on moving objects has proceeded along three directions: data models and query languages, index structures, and efficient query processing techniques. As we have observed, motion models tend to be simple, with many researchers assuming linear motion. Techniques for handling systems of curves remain underdeveloped.

Güting et al. [7] have developed a comprehensive model for implementing spatiotemporal DBMS extensions. Data models and query languages for road networks have been discussed in [16].

The research community has paid special attention to the issue of indexing moving objects [12, 1]. Index structures for moving data are more complex than those for traditional data because they must deal with continuously changing positions with an additional *time* dimension. Kollios et al. [8] solved the problem for objects moving in 1-dimension by using the line-point duality transformation. The authors have alluded to the problem of objects moving in restricted trajectories under the title of 1.5-dimensional trajectories. They propose to represent each predefined route as sequence of connected straight line segments and to index motion on a route using the duality transform for each segment. However, their technique is inapplicable to queries involving distances *along* the road.

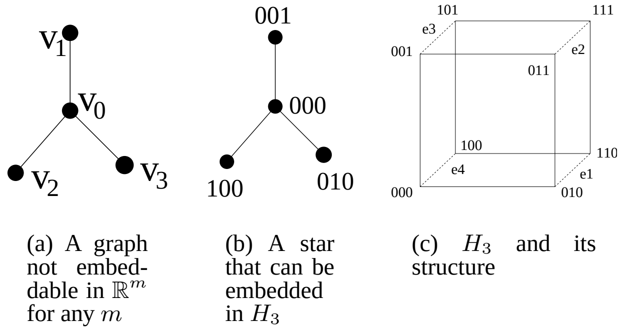
Recent work [8, 12, 14] on efficiently handling queries on objects moving on the unrestricted plane are only applicable as a first cut filter. For the final cut, ultimately, a slow shortest distance algorithm has to be run on the underlying planar graph, making these solutions less satisfactory.

Several algorithms have been proposed for the shortest distance problem, and in particular, for planar graphs. However, they all turn out not to have the properties that we require. Dijkstra's celebrated algorithm [4] takes $O(n)$ space and $O(n \log n)$ time, when implemented with heaps. An $O(n\sqrt{\log n})$ time algorithm [5] that works for planar graphs is the most practical and efficient of all the algorithms available. However, since we would like to be able to handle very large networks, we would prefer a response time even smaller than provided by current methods. One simple approach might be to compute shortest paths in advance and do a simple table lookup to give a constant time response. However, this method is also impractical since even a network with just 100,000 nodes would require a table with five billion entries.

Shahabi et al. [13] solved the k-nearest neighbor problem with distance measured as the shortest path on the road network. They propose to use modified Lipschitz embedding [9] which approximately preserves shortest distances on the road network. Spatiotemporal queries on road networks were also addressed by Papadias et al. [11]. They use an R-tree over the road network to prune the search space. However, both methods are not very useful to answer exact queries, as we still need to compute shortest paths in the underlying graph.

Gavoille et al. [6] addressed the problem of labeling the nodes of a graph to allow computation of distances directly from their labels. They show that given any distance labeling scheme on planar graphs, there exist graphs for which the labels need $\Omega(n^{1/3})$ bits per node. They give an encoding scheme based on separators that assigns labels of length $O(\sqrt{n} \log n)$ to each vertex, with the distance computable from the labels in time $O(\log n)$. Their labeling scheme takes O($n\sqrt{n} \log n$) time to compute the labels. We use Hypercube embeddings of graphs to obtain distance labels. The labels obtained by Hypercube embeddings allows the use of a hashing scheme that significantly improves the performance of spatiotemporal queries.

Hypercube embeddings of general graphs have applications in the



(a) A graph not embeddable in $\mathbb{R}^m$ for any m

(b) A star that can be embedded in H_3

(c) H_3 and its structure

Figure 1: Hypercube Embedding

domains of Mathematical Chemistry and Distributed Computing. Chepoi et al. [3] used hypercube embeddings to estimate the Weiner index of certain aromatic chemical compounds. They developed novel techniques for identifying a broad class of planar graphs embeddable in hypercubes.

In our work, we build on top of their techniques. We show methods to assign codes for a given planar graph based on hypercube embeddings. The encoding allows for a hashing scheme that significantly optimizes spatiotemporal queries on graph. Although not all road networks can be modeled as planar graphs, these techniques are significant step forward toward solving spatiotemporal queries. It is likely that a suite of similar techniques can be built using the work in [6]

4. SOME ISSUES AND CHALLENGES

Spatiotemporal queries typically use a heuristic in a filter stage, and perform exact queries on the results in a refinement step. We argue that in the case of road networks, *the refinement step is a bottleneck*, since it typically requires the computation of the shortest distances in the underlying planar graph. Our approach yields an efficient shortest distance problem in the first place. We will see that our method does very well even without a filter step.

As we observe in Section 3, the space and time requirements of current algorithms for the shortest-path problem on weighted graphs are far higher than we would like in our application domain. Specifically, we want to treat the graph vertices as points in a suitable metric space, with an easily computed metric. Unfortunately, because of their generality and discrete structure, graphs do not readily permit definition of such metrics.

The standard realization of this approach in the literature is to map the vertices $v_1, v_2, \ldots v_n$ to points $p_1, p_2, \ldots p_n$ in $\mathbb{R}^m$ for sufficiently large m, such that the Euclidean distance between p_i and p_j in $\mathbb{R}^m$ is equal to, or arbitrarily close to, the shortest distance between the vertices v_i and v_j on the planar graph. One could then index these points using some suitable multidimensional spatial index structure.

While this approach sounds reasonable, it fails badly because of the inherent structure of $\mathbb{R}^m$. Consider the graph shown in Figure 1(a). If $d(v_i, v_j)$ is the shortest distance along the graph between vertices v_i and v_j, we observe that the "triangle equality" $d(v_1, v_2) = d(v_1, v_0) + d(v_0, v_2)$ must hold. In $\mathbb{R}^m$, this requires $||p_1 - p_2|| = ||p_1 - p_0|| + ||p_0 - p_2||$, making points (p_1, p_0, p_2) collinear (in that order). For similar reasons, (p_2, p_0, p_3) and (p_3, p_0, p_1) would also need to be collinear triples of points. Clearly, such a configuration of points is impossible in $\mathbb{R}^m$. In fact, arbitrary close approximations are impossible as well. Thus any graph which contains a star shaped subgraph similar to that in Figure 1(a) cannot be embedded in $\mathbb{R}^m$. Unfortunately, this star shaped figure is found in real road networks at every T-junction.

4.1 Hypercube Embeddings

We therefore seek a metric space which, among other things, can handle multiple triangle equalities simultaneously. A good instance of such a metric space is the m-dimensional hypercube $H_m = \{0,1\}^m$ with the Hamming distance as the metric[1]. The Hamming distance between two points $a_0a_1a_2 \ldots a_{m-1}$ and $b_0b_1b_2 \ldots b_{m-1}$ ($a_i, b_i \in \{0,1\}$) is defined as the number of k such that $a_k \neq b_k$. For example, the Hamming distance between 011010 and 101100 is 4, because the two strings differ in 4 bit positions. It turns out that the structure of H_m is actually rich enough to embed a large subclass of the class of planar graphs, such that the distance between points on the graph is preserved as the distance between the corresponding points in the hypercube.

We now state some standard facts about the m-dimensional hypercube H_m, which are useful in developing our algorithm for hypercube-embedding of planar graphs. Consider the set V_{0k} of all vertices of H_m that have 0 in their k^{th} bit position, and the set V_{1k} of all vertices of H_m that have 1 in their k^{th} bit position. Clearly V_{0k} and V_{1k} are connected graphs, each isomorphic to H_{m-1}. Let E_k be the set of all edges that join a vertex in V_{0k} to a vertex in V_{1k}. Given a pair of vertices $v_0 \in V_{0k}, v_1 \in V_{1k}$, any shortest path between v_0 and v_1 must use one and only one edge in E_k. Conversely, if the shortest path between two vertices v_0 and v_1 intersects E_k in one edge, then v_0 and v_1 must be in different components of H_m (either in V_{0k} or in V_{1k}). Clearly, every edge in H_m is in exactly one E_k for some k.

For example, Figure 1(c) represents H_3. V_{00} is the set $\{000, 001, 010, 011\}$, and V_{10} is the set $\{100, 101, 110, 111\}$. E_0 is the set of edges $\{e_1, e_2, e_3, e_4\}$. Observe that V_{00} and V_{10} are isomorphic to H_2 (the square). Also, shortest paths within V_{00} lie entirely within V_{00}, and shortest paths from V_{00} to V_{10} necessarily intersect E_0.

Let G be a planar graph embeddable in H_m. Let G_{0k} and G_{1k} be the sets of vertices whose k-th bit is 0 and 1, respectively. Then the set E_k of all edges between G_{0k} and G_{1k} will be such that no shortest path contains more than one edge from E_k. Conversely, assume we can find a set E_k of edges that connects two disjoint partitions of a graph, such that no shortest path intersects E_k more than once. Then E_k would define the boundary between partitions G_{0k} and G_{1k} for some k.

Given such a set of edges E_k, we assign 0 in bit position k to all nodes in one of the partitions created, and 1 to the all nodes in the other partition. If we find that every edge of the planar graph falls into one such set, our embedding is complete, because we have now taken every bit position into account.

The isometric cycle property guarantees that for two opposite edges e_1 and e_2 on a face, any shortest path that passes through e_1 cannot pass through e_2. This idea can easily be generalized to edge sequences called *alternating cuts*, in which consecutive edges are op-

[1]H_m is the graph with elements of $\{0,1\}^m$ as vertices. Two points u and v have an edge between them iff u and v differ in exactly one bit position

posite each other on a graph face. Any such sequences has the property that a shortest path contains no more than one edge from the sequence. We will demonstrate that identifying such sequences can be done very easily. Once we have identified all such sequences, the problem of embedding the graph into a hypercube is straightforward.

5. CUTS AND CODE ASSIGNMENT

In this section, we describe our scheme to assign codes to the nodes in road network.

We first accommodate weighted edges in $\mathcal{G}$ as follows. If edge (u, v) has weight k, we introduce $k-1$ virtual nodes into $\mathcal{G}$ between u and v, and assign a weight of 1 to each of these new edges. This trick preserves the shortest distances between the nodes in the original version of $\mathcal{G}$, while ensuring that the edges in the new graph all have weight 1. For the rest of this section, we assume that all the edges of our graph have unit weight.

5.1 Definitions and Notation

As before, a road network is represented as a planar graph $\mathcal{G}(V, E, W)$ with a drawing in $\mathbb{R}^2$. The orientations clockwise and anti-clockwise are defined with respect to this drawing. We begin with some definitions.

Interior Face $\mathcal{F}$: An interior face is a cycle of $\mathcal{G}$ that bounds a connected region. $\mathcal{F}[n]$ denotes the n^{th} edge of $\mathcal{F}$, when the edges of $\mathcal{F}$ are arranged in clockwise order. $\mathcal{F}[0]$ can be an arbitrary edge of the face. Every edge appears in two faces, not necessarily distinct.

Outer Face $\mathcal{C}$: The outer face of $\mathcal{G}$ is the unbounded face in the embedding of $\mathcal{G}$ in $\mathbb{R}^2$.

Odd (Even) Face: A face with an odd (even) number of edges.

Opposite Edges: Edges $e = (u, v)\ and\ e' = (u', v')$ are opposite in face $\mathcal{F}$ if $d(u, u') = d(v, v')$ and equal the diameter of the cycle $\mathcal{F}$. If $\mathcal{F}$ is an even-face then every edge $e \in \mathcal{F}$ has unique opposite edge. If $\mathcal{F}$ is an odd-face than every edge e has two opposite edges e^+ and e^-.

Cut $\mathcal{L}$: The concept of a cut is central to our method. Formally, a cut is a sequence of edges $\{e_1, e_2, e_3, \ldots, e_k\}$ with the following properties.

1. Either $e_1 = e_k$ or $e_1 \in \mathcal{C}$ and $e_k \in \mathcal{C}$
2. $\exists \mathcal{F}[e_i, e_{i+1} \in \mathcal{F}]$
3. e_{i+1} is an opposite edge of e_i in face $\mathcal{F}$

If the edges of a cut $\mathcal{L}$ are deleted from the graph $\mathcal{G}$, the graph gets partitioned into two components $\{\mathcal{G}/\mathcal{L}\}_0$,$\{\mathcal{G}/\mathcal{L}\}_1$.

Alternating Cut An alternating cut is a cut that alternates over odd faces. By this we mean that if the cut takes a right (left, respectively) turn on one odd face, then on the next odd face it encounters it takes a left (right, respectively) turn. An alternating cut can be visualized as a line through the graph, intersecting select edges only. An example of an alternating cut is shown in Figure 2 .

Core: A maximal two-connected subgraph of $\mathcal{G}$.

Core Tree: It is a well known result [2] that the interconnection graph of all the maximal two connected subgraphs of a graph is a tree. We call this the *core tree* (see Figure 5).

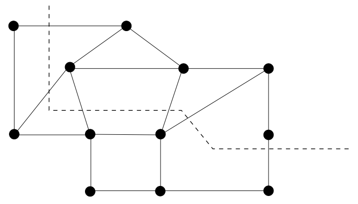


Figure 2: Alternating cuts alternate on odd cycles

We call a cut $\mathcal{L}$ *convex* if the shortest path between any two vertices in $\{\mathcal{G}/\mathcal{L}\}_0$ lies entirely within $\{\mathcal{G}/\mathcal{L}\}_0$. A graph satisfies the *generalized isometry property* if all alternating cuts are convex.

5.2 The Encoding Scheme

We now describe the full encoding scheme for the planar graph $\mathcal{G}$. The cores of $\mathcal{G}$ are obtained from $\mathcal{G}$ by removing all edges which have only one adjacent face. The resulting components are all the cores of the graph. Hypercube embeddings are a lot more efficient for two-connected graphs (as we shall see in Section 7). We therefore choose to encode each core into a hypercube separately. Thus the distance computation will involve finding a distance along the core tree (which can be done very efficiently), as well as computing distances within two cores using the hypercube embeddings. Note that shortest distances along a tree can be very efficiently computed after an initial round of preprocessing. Although this is a two-stage scheme, the resulting distances are not hierarchical. Our technique still calculates *the* shortest distance between any two nodes. In describing how we encode the cores, we will treat each core as an independent planar graph. Henceforth $\mathcal{G}$ will denote the core on which the algorithm is currently being run.

5.3 Encoding Cores:Some Insights

We start by stating some properties of alternating cuts, upon which our algorithm is based. These properties are proved in later sections. For any alternating cut $\mathcal{L}$, consider the two components $\{\mathcal{G}/\mathcal{L}\}_0$ and $\{\mathcal{G}/\mathcal{L}\}_1$. For any two nodes u,v with $u \in \{\mathcal{G}/\mathcal{L}\}_0$ and $v \in \{\mathcal{G}/\mathcal{L}\}_1$, the shortest path from u to v has one edge in $\mathcal{L}$. Furthermore, every edge of the shortest path from u to v is part of two cuts $\mathcal{L}_1$ and $\mathcal{L}_2$. We take advantage of these facts in our algorithm.

Our code-assignment algorithm proceeds as follows. First we initialize all codes to the null string. For every alternating cut $\mathcal{L}$, we find the connected components $\{\mathcal{G}/\mathcal{L}\}_0$ and $\{\mathcal{G}/\mathcal{L}\}_1$. We append 0 to the code of each node in $\{\mathcal{G}/\mathcal{L}\}_1$ and 1 to the code of each node in $\{\mathcal{G}/\mathcal{L}\}_1$. On termination of the algorithm the code of each node is a bit string of length equal to the total number of alternating cuts. We later show that for any two nodes (u,v), the Hamming distance between their codes gives the shortest distance between (u,v). Furthermore, there is a correspondence between alternating cuts and bit positions in the final codes.

To find an alternating cut containing edge e, the procedure follows naturally from the definition. Starting at edge e, we proceed in both directions, taking opposite edges on all even faces, until we come across the first odd face in both directions. Then, on one odd face we take a right turn, and the other odd face we take a left turn (by changing the choice of odd faces, we can get one more alternating cut). From then on, we proceed in both directions alternating at odd faces until we reach the outer face. We call a cut produced by such a procedure starting at edge e a cut *seeded* at e. Note that an alternating cut that consists of edges on even faces only, should

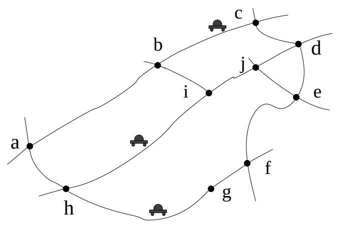


Figure 3: A simple road network

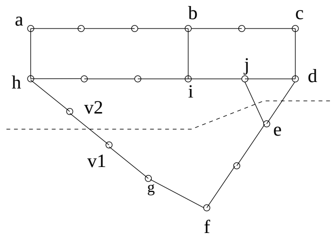


Figure 4: Converting a road map into its corresponding planar graph.

be counted as two cuts, and two bits should be alloted in the code based on this cut (for reasons that will be explained later).

5.4 Encoding the Cores: An Example

We now illustrate our algorithm through an example. Figure 3 shows a road network, which we transform into the corresponding planar graph. Distances may be assigned by simple discretization of the road lengths according to the precision requirements. Figure 4 shows the development of one of the cuts of the graph. In Table 1 we list the codes for each of the nodes induced by the cuts of the graph.

The road network consists of 7 roads with 10 nodes, which form a planar graph in Figure 1. Virtual nodes are inserted at every unit distance between nodes. For example, the distance between nodes a and b is 3 units. Thus two virtual nodes are inserted between them. A total of eight virtual nodes are added to make all edges of unit length. For the rest of this example we will not differentiate between original nodes and virtual nodes. All faces will be described by their bounding vertices in an anti-clockwise fashion. Now that we have constructed the planar graph, we find all the alternating cuts of the graph.

We describe a procedure for finding an alternating cut through the edge (d, e), in the example graph of Figure 4. Since face (e,d,j) has an odd number of edges, the cut has to make a turn. We start by making a left turn to cut the edge (j,e) (another valid alternating cut would have been formed if we had taken a right instead).

It then enters face (j, e, f, g, $v1$, $v2$, h, i, j), which has an odd number of edges again. Since the cut alternates its turn at odd faces, it makes a right turn in this face. The right opposite edge of (j,e) is ($v1$,$v2$). The cut leaves the graph and enters the outer face. The entire cut is given in Figure 4.

After finding all the cuts, we assign codes to the nodes. Each cut partitions the planar graph in two disjoint components. The cut shown here induces two components, namely (a,h,i,j,d,c, b) and (g,f,e). We append '1' to the codes for nodes in one component and '0' to the nodes in the other. Since a bit is appended to the code for each cut, the code size equals the number of cuts graph.

Node	Code
a	01011111111011
b	00000001011011
c	00000000000111
d	00000000000100
e	10000000000000
f	11111000000000
g	11111110000000
h	01011111111000
i	00000001011000
j	00000000001000

Table 1: The embedding in H_{14} of the example road map

Table 1 list the codes for all nodes. As remarked earlier, the Hamming distance between the codes is twice the length of the shortest path between the corresponding vertices of the graph. For example, the shortest distance on the graph between nodes a and f is 5. The corresponding codes for a and f are 01011111111011 and 11111000000000. These have a Hamming separation of 10, which is exactly twice the distance between the graph nodes. The validity of this encoding for other pairs of points may be verified.

5.5 Encoding the Cores: Detailed Description

Here we describe our algorithm in detail. The pseudo code is given in Algorithms 1, 2, 3, 4, 5, 6. All algorithms assume that the planar graph $\mathcal{G}$ is available as a global variable. Furthermore all of the interior faces $\mathcal{F}$ are given. What follows now is a short description of each part of the algorithm. In section 5.5 we put all pieces together to present the complete picture.

Other Face As already mentioned, every edge has two adjacent faces. Given an edge e and a face $\mathcal{F}$ adjacent to it, this function gives the other face adjacent to it.

Algorithm 1 $other\_face(edge\ e, face\ \mathcal{F})$

```
Require: $\mathcal{F}$ is an incident face of e
1: $(\mathcal{F}1, \mathcal{F}2) = incident\ faces\ of\ e$
2: if $\mathcal{F}1 = \mathcal{F}$ then
3:     return $\mathcal{F}2$
4: else
5:     return $\mathcal{F}1$
6: end if
```

Opposite Given a face $\mathcal{F}$ and an edge e on it, this function returns an edge opposite to e in face F.

Algorithm 2 $opposite(edge\ e, face\ \mathcal{F}, bool\ \tau)$

```
1: $n = sizeof(\mathcal{F})$ {number of edges $\mathcal{F}$}
2: $p = index(e, \mathcal{F})${position of e in cyclic order for edges in $\mathcal{F}$}
3: if $sizeof(\mathcal{F}) \bmod 2 = 1$ then
4:     if $\tau$ then
5:         return F$[(\lfloor n/2 \rfloor + p) \bmod n]$
6:     else
7:         return F$[(\lceil n/2 \rceil + p) \bmod n]$
8:     end if
9: else
10:    return F$[(n/2 + p) \bmod n]$
11: end if
```

Allocate Codes Each time this function is called it appends a bit to the code of each vertex. Line 4 finds the two components $\{\mathcal{G}/\mathcal{L}\}_0$ and $\{\mathcal{G}/\mathcal{L}\}_1$ induced by the cut $\mathcal{L}$ on graph $\mathcal{G}$. It then appends 0 to the code of each vertex in $\{\mathcal{G}/\mathcal{L}\}_0$ (line 5-7) and 1 to the code of each vertex in $\{\mathcal{G}/\mathcal{L}\}_1$ (line 8-10).

Algorithm 3 $allocate\_code(cut\ L)$

```
1: for all edge $e \in L$ do
2:     $delete(e, \mathcal{G})$
3: end for
4: $(\mathcal{G}1, \mathcal{G}2) = connected\_component(G)$
5: for all node $v \in \mathcal{G}1$ do
6:     $append(v.code, '1')$
7: end for
8: for all node $v \in \mathcal{G}2$ do
9:     $append(v.code, '0')$
10: end for
11: for all edge $e \in L$ do
12:     $add(e, \mathcal{G})$
13: end for
```

Half Cut

Conceptually, the cut starts at edge e, goes through face $\mathcal{F}$ incident on it, and extends further till it meets the outer face $\mathcal{C}$ (line 5-10). A cut alternates its turn at every odd face (line 8-10), where the direction of the first turn is determined by τ. It returns the list of all edges that lie on this cut.

Alternate Cut

To find an alternating cut through edge e, the algorithm first finds the faces adjacent to e: $\mathcal{F}1, \mathcal{F}2$ (line 2). It then finds two half cuts, $\mathcal{L}1$ and $\mathcal{L}2$, starting at edge e, one going through $\mathcal{F}1$, the other through $\mathcal{F}2$ (line 4,5). The variable τ makes sure that the turn made at first odd face in $\mathcal{L}1$ alternates with the turn made at the first odd face in $\mathcal{L}2$. The two half cuts are merged to form a complete alternating cut. To find the other cut passing through this edge, we just change the value of τ.

Putting Everything Together This section combines all the functions mentioned above to produce the encoding of the vertices of the given planar graph. Line 2 initializes a stack S of edges. We then initialize the variables of the edges. All edges are then pushed into the stack (line 3 -5).

For every edge e, variable $e.num\_cut$ stores the number of cuts that include e that have been found already. Since all edges have exactly two cuts through them (this will be shown), an edge is removed from the stack once both its cuts are found. Algorithm 4 increments this variable for edge e when the cut that it is exploring crosses the edge e.

The algorithm terminates when all alternating cuts are discovered. For each cut discovered we call the function $allocate\_code$, which appends bits to each vertex of the graph (line 14-15).

5.6 Encoding the Cores: Proof of Correctness

The correctness of the hypercube embedding follows from the following theorems, whose proofs are omitted due to space limitations. The following theorems hold for graphs that satisfy the generalized isometry property.

THEOREM 5.1. *The shortest path between any two vertices u and v intersects any alternating cut in at most one edge.*

Sketch of a proof: This result follows from the isometric cycle property. Since an alternating cut goes from one edge to its opposite edge, alternating on odd faces ensures that if any path p contains two edges on an alternating cut, then one can find a path that is shorter than p between the same vertices. For a detailed proof, the reader is referred to [3].

Algorithm 4 $half\_cut(edge\ e, face\ \mathcal{F}, turn\ \tau)$

```
1: cut $\mathcal{L} = NULL$
2: edge $e' = NULL$
3: face $\mathcal{C} = unbounded\_face(G)$
4: $\mathcal{L}.insert(e)$
5: while $e' \notin \mathcal{C}$ do
6:     $e' = opposite(e', \mathcal{F}, \tau)$
7:     $e'.num\_cut = e'.num\_cut + 1$
8:     $\mathcal{F} = other\_face(e', \mathcal{F})$
9:     if $\mathcal{F}\ is\ an\ odd face$ then
10:        $\tau = \neg\tau$
11:    end if
12:    $\mathcal{L}.insert(e')$
13: end while
14: return $\mathcal{L}$
```

Algorithm 5 $alternate\_cut(edge\ e)$

```
1: cut $L = NULL$
2: $[\mathcal{F}1, \mathcal{F}2] = faces\ incident\ on\ e$
3: $\tau = RIGHT$
4: $\mathcal{L}_\vdash = half\_cut(\mathcal{F}1, e, \tau)$
5: $\mathcal{L}_\dashv = half\_cut(\mathcal{F}2, e, \neg\tau)$
6: $\mathcal{L}1 = \mathcal{L}1_\vdash + \mathcal{L}1_\dashv$
7: $\tau = LEFT$
8: $\mathcal{L}_\vdash = half\_cut(\mathcal{F}1, e, \tau)$
9: $\mathcal{L}_\dashv = half\_cut(\mathcal{F}2, e, \neg\tau)$
10: $\mathcal{L}2 = \mathcal{L}1_\vdash + \mathcal{L}1_\dashv$
11: return $[\mathcal{L}1, \mathcal{L}2]$
```

THEOREM 5.2. *Each edge occurs in exactly* 2 *alternating cuts.*

PROOF. Define a relation $\sim$ on E by the following: $e_1 \sim e_2$ if and only if e_2 lies on a cut seeded by e_1. We claim that $\sim$ is a symmetric relation on E.

To show that $e_1 \sim e_2 \Rightarrow e_2 \sim e_1$, we first note that if e_2 is on the primitive segment of e_1, the statement is trivial. The result follows by an easy induction on the number of odd faces between e_1 and e_2 along the cut seeded by e_1. The induction uses the following stronger induction hypothesis: a cut seeded by e_1 that passes through e_2, is also a cut seeded by e_2.

Algorithm 6 $find\_code()$

```
1: $\mathcal{C} = unbounded\_face(\mathcal{G})$
2: stack S
3: for all edges $e \in \mathcal{G}$ do
4:     $e.num\_cut = 0$
5:     $S.push(e)$
6: end for
7: for all nodes $v \in G$ do
8:     $v.code = \phi$ { assign empty codes}
9: end for
10: while $S.not\_empty()$ do
11:    $e = S.pop()$
12:    if $e.num\_cut < 2$ then
13:        $[\mathcal{L}1, \mathcal{L}2] = alternate\_cut(e, G)$
14:        $allocate\_code(\mathcal{G}, \mathcal{L}1)$
15:        $allocate\_code(\mathcal{G}, \mathcal{L}2)$
16:        $S.push(e)$
17:    end if
18: end while
```

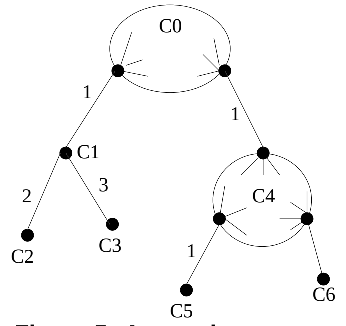


Figure 5: A sample core tree

Since $\sim$ is a symmetric relation, the only cuts that pass through a given $e \in E$ are the cuts seeded by e. Thus there are exactly two cuts per edge. $\square$

Now it is fairly straightforward to see that any shortest path will intersect exactly twice as many cuts as there are edges on the path. Thus the number of bits that change from the codes of one vertex to the codes of another will be exactly twice the length of the shortest path between the two vertices.

6. SPATIOTEMPORAL QUERIES

In this section, we show how to apply our shortest distance scheme to efficiently answer several types of spatial and spatiotemporal queries. Our scheme is asymptotically better than any existing method, and also has vastly superior practical performance. Existing methods must depend upon an slow shortest distance algorithms.

Representing Road Networks: Each vertex v_i is assigned a code $c(v_i)$ using the encoding technique presented earlier, The shortest distance between vertices v_i and v_j is simply the Hamming distance between codes $c(v_i)$ and $c(v_j)$, which we represent as $|(c(v_i), c(v_j)|$. For each vertex v_i we maintain a set $\mathcal{S}_i$ of its adjacent nodes. To improve query processing efficiency, we also store information about the core tree in the network.

Representing Points: Since objects move only along graph edges, their positions can be defined by distances along edges. A position p_i on edge e_j is assigned coordinates of the form (v_{j_1}, v_{j_2}, d_i), where $e_j = (v_{j_1}, v_{j_2})$ and d_i indicates how far along e_j the position p_j is located. The shortest distance between two points can now be trivially found with just four computations of shortest distance between vertices.

Representing Routes: Since earlier methods have only allowed linear motion, they have sought to represent object paths as a series of line segments, an approach that is awkward and often expensive. Trajectories in real roads datasets [15] must be composed from many line segments, which can significantly affect the overall performance of these techniques.

Let $\langle i, j\rangle$ represent the shortest path between vertices v_i and v_j. Our routes are shortest paths between the source and destination vertices, so $\langle i, j\rangle$ would be the normal route between v_i and v_j. To accommodate routes that differ from this shortest path, we define *waypoints*, and require objects to follow the shortest path between waypoints. The route for an object that moves between vertices v_{i_1} and v_{i_m}, passing through waypoints $v_{i_2}, \cdots, v_{i_{m-1}}$, is represented as $\langle i_1, i_2, \cdots, i_{m-1}, i_m\rangle$, and is obtained by concatenating the shortest paths $\langle i_1, i_2\rangle$, $\langle i_2, i_3\rangle, \cdots, \langle i_{m-1}, i_m\rangle$.

For reasonable vehicular paths, the number of way-points is likely to be low. Further, if several shortest paths exist between the given (source, destination) pair, one can force a choice between them by using waypoints.

6.1 The Route Hashing Optimization

We describe a hashing scheme that places shortest paths into bins that capture the manner in which they traverse the graph, allowing significant query optimizations. For example, given a query region, we are able to discard all routes in bins that don't appropriately match the codes for the vertices defining the region. The method works as follows.

Each cut $\mathcal{L}$ corresponds to a bit position (say $k_{\mathcal{L}}$) in the encoding. All vertices in the first partition $\{\mathcal{G}/\mathcal{L}\}_0$ induced by $\mathcal{L}$ have $k_{\mathcal{L}} = 0$, and all vertices in the other partition $\{\mathcal{G}/\mathcal{L}\}_1$ have $k_{\mathcal{L}} = 1$. If both the source and destination of a given route belong to the same partition, their codes will agree in the bit position corresponding to $k_{\mathcal{L}}$, and the shortest path between this source and destination will use no edge outside this partition.

Let subgraph R be defined by the vertices $(v_{i_1}, v_{i_2}, \cdots, v_{i_k})$. Suppose we can find a cut $\mathcal{L}$ so that all these vertices belong to $\{\mathcal{G}/\mathcal{L}\}_0$, i.e., their codes have $k_{\mathcal{L}} = 0$. Any shortest path $\langle v_i, v_j\rangle$ where both v_i and v_j have $k_{\mathcal{L}} = 1$ will lie outside $\{\mathcal{G}/\mathcal{L}\}_0$, and hence outside R. In this case, we say that $\mathcal{L}$ *excludes* $\langle v_i, v_j\rangle$ from region R. Cuts which exclude many routes from R reduce the search space significantly.

This idea can be generalized to the the concept of *route hashing*. Let a cut be called a *bisector* if it partitions the graph into two halves with roughly the same number of vertices. A few bisector cuts $\mathcal{L}_1, \mathcal{L}_2 \ldots \mathcal{L}_m$ are selected as the basis for creating bins. The key to route hashing lies in considering only the bit positions $k_{\mathcal{L}_1}, \cdots, k_{\mathcal{L}_m}$ corresponding to these bisectors. We call these bit positions the *bisector bits*. These m bits are used to define 2^m bisector bins, corresponding to regions on the planar graph, and labeled by binary strings of length m. The idea is to place a route in a bin if it is not excluded from entering the corresponding region of the planar graph. Specifically, the shortest path $\langle v_i, v_j\rangle$ is placed in all bisector bins b_l whose labels match the codes for either v_i or v_j in *all* of the bit positions $k_{\mathcal{L}_1}, \cdots, k_{\mathcal{L}_m}$. Using bisectors ensures that the bins are roughly equal-sized and routes are uniformly distributed across them. In a preprocessing step, we hash the set of routes $\langle o_i\rangle$ for the objects o_i in the system into these bisector bins. Route hashing can be used to reduce the search space for spatial and spatiotemporal queries, boosting their performance.

6.2 Queries

Observe that a vertex v lies on a shortest path $\langle i, j\rangle$ if and only if $|c(v_i, v_j)| = |c(v_i, v)| + |c(v, v_j)|$. In this case, we say $\langle i, j\rangle$ *includes* v. Similarly, $\langle i_1, i_2, \cdots, i_m\rangle$ includes v if and only if there are two consecutive way points $v_{i_k}, v_{i_{k+1}}$ such that $\langle k, k+1\rangle$ includes v.

When we do not wish to refer explicitly to waypoints, we denote the route taken by object o_i simply as $\langle o_i\rangle$. The distance of vertex v along the route $\langle o_i\rangle$ is denoted by $dist(\langle o_i\rangle, v)$. The speed of object o_i is denoted by $u_i(t)$. If o_i passes through vertex v at time t_v, we will have $dist(\langle o_i\rangle, v) = \int_0^{t_v} u_i(t)dt$.

6.2.1 *Incidence Queries*

Given a query vertex v_q and time interval $[t_1, t_2]$, an incidence query requests all objects that pass through v_q during $[t_1, t_2]$. We first pick the bisector bins that agree with the code $c(v_q)$ on the bisector bits. For each route $\langle o_i \rangle$ in these bins, we first check if $\langle o_i \rangle$ includes v_q, and second, whether o_i could passed v_q during $[t_1, t_2]$. In other words, we check whether $\int_0^{t_1} u_i(t)dt \leq dist(\langle o_i \rangle, v_q) \leq \int_0^{t_2} u_i(t)dt$.

6.2.2 *Shortest Path Queries*

To find the shortest path $\langle v_i, v_j \rangle$ between vertices v_i and v_j, we start at v_i, and find the neighboring vertex w of v_i such that $|c(v_i), c(v_j)| = |c(v_i), c(w)| + |c(w), c(v_j)|$. Clearly, $\langle v_i, v_j \rangle$ includes w, which is hence the first node on this shortest path. We repeat the process until we arrive at v_j.

6.2.3 *Range Queries*

A range query requests all objects o_i that pass through some query region R during some time interval $[t_1, t_2]$. R is intended to bound object positions, so R is typically a rectangle when objects move linearly in the plane. In our case, however, objects are constrained to graph edges, so the proper way to bound their positions is by points on graph edges. In Figure 6, for example, the region R is specified by the set of positions $\{r_1, r_2, r_3, r_4, r_5, r_6\}$ on graph edges. To answer range queries, we first identify the vertex set B_R that bounds R. In Figure 6, this bounding set is $\{c_1, c_2, \cdots, c_6\}$.

Our next step is to select a subset of bisector bins. For each vertex $v_i \in B_R$, consider the encoding $c(v_i)$, and let β_i be the substrings of $c(v_i)$ obtained by choosing only the bisector bit positions. Let γ be the set of bisector positions where all the β_i match. We only need to look at the bisector bins which match $c(v_i)$ at positions γ. Let this set of bisector bins be called the set of γ-bins.

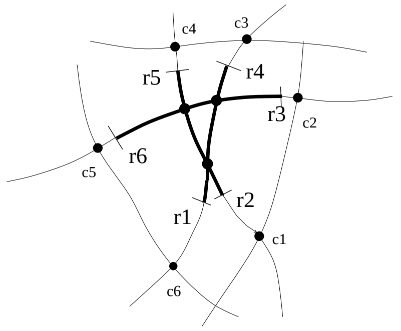
We observe that any object entering or leaving R during time interval $[t_1, t_2]$, must cross at least one of the vertices in B_R during that interval. For each route $\langle o_i \rangle$ in the γ-bins of B_R, we check if $\langle o_i \rangle$ is incident on a vertex in B_R (see Section 6.2.1) during $[t_1, t_2]$.

The other case that remains is when an object o moves such that its route $\langle o \rangle$ lies completely within the region. This can be detected by checking if the shortest path between the source and a vertex p lying within the region intersects B_R.

6.3 Join Queries

Given a time interval $[t_1, t_2]$, we are required to find all pairs of (o_i, o_j), such that o_i and o_j are on the same edge during $[t_i, t_2]$. Since routes in different bins do not intersect, so only routes that are within same bin need be compared. To find all routes $\langle o_j \rangle$ that intersect route $\langle o_i \rangle$ during $[t_1, t_2]$, first enumerate the vertices $u_1, u_2, \ldots, u_k$ included by $\langle o_i \rangle$, and the times $t_{i_1}, t_{i_2}, \ldots, t_{i_k}$ that o_i arrives at each of them. Discard all u_j for which $t_{i_j} \notin [t_{q1}, t_{q2}]$. For each of the remaining u_i, we find all objects o_j with $\langle o_j \rangle$ incident on u_i during the time interval $[t_{i-1}, t_{i+1}]$ (see Section 6.2.1). This procedure is repeated for each object o.

To find all pairs of objects that are within distance d of one another at a given time t_q, we use the the standard all pairs join method. This method is feasible because shortest distance computation is now fast.



– Road Segments representing Query Region R
● Nodal Points c1, c2, ..., c6 bound query region R

Figure 6: Highlighted segment represents query region

6.4 Intercept Queries

An object o_p (the "pursuer") starts at a specified node, and is required to catch a second object o_q (the "quarry"), whose route $\langle o_q \rangle$ is given. The speeds $u_p(t), u_q(t)$ of both vehicles are known, and we are required to determine whether o_p can catch o_q.

Let $\{v_1, v_2, \cdots, v_k\}$ be the set of vertices included by o_q's route $\langle o_q \rangle$ (see Section 6.2.2). Let the distances of each of these vertices from the starting positions of o_q and o_p be $\{d_1^q, d_2^q, \cdots, d_k^q\}$ and $\{d_1^p, d_2^p, \cdots, d_k^p\}$ respectively. These are trivially obtained as the Hamming distances between their codes. Since the speeds $u_q(t)$ and $u_p(t)$ are known, we can compute the times $\{t_1^q, t_2^q, \cdots, t_k^q\}$ and $\{t_1^p, t_2^p, \cdots, t_k^p\}$ at which o_q and o_p are incident on each of these vertices (see Section 6.2).

An interception is possible if and only if $t_i^p \leq t_i^q$ for some vertex v_i. That is, o_p gets to v_i first.

7. CODING PERFORMANCE

In this section, we prove that the performance of the algorithm given in the previous section is asymptotically good. Our asymptotic results are in terms of the number of nodes n in the original graph, not including the virtual nodes. Our experiments include these virtual nodes as well.

To make the analysis tractable, and provide reasonable performance estimates for realistic scenarios, we make a few assumptions that do not affect the correctness or execution properties of our algorithm. We assume that the vertex degree and edge weights are both bounded from above by constants. In practice, it is unreasonable to have intersections where an unbounded number of roads meet, or to have roads of unbounded length.

Since the complexity results for encoding trees are already well-known [4], we only present complexity results for the encoding of cores. The space and time requirements for constructing and coding the core tree are all asymptotically very good. Proofs are omitted due to lack of space.

LEMMA 7.1. *For road networks, the total number of alternating cuts is the same as the length of the outer face.*

PROOF. Clearly, every alternating cut either starts and ends at outer edges, or forms a loop. We will argue that for the kinds of graphs that we are looking at, no alternating cuts will form loops.

An alternating cut through edge e will form a loop if (1) Both the faces adjacent to e are not distinct, or (2) The alternating cut passes through several faces before turning around and passing through the same edge again. Our encoding methods operate on two-connected subgraphs, and so the first case does not arise. Planar graphs in which alternating cuts form loops (as in the second case) are highly distorted [3], because if two edges are opposite in a face, they also tend to be spatially opposite. For a loop to form, the cut has to turn a complete 360 degrees, which is completely against the spatially opposite principle.

Thus, in a typical road network we would expect that no alternating cuts would form loops. So, all alternating cuts are from an outer edge to an outer edge. Further, there are exactly 2 cuts per outer edge, and every cut is counted twice, once at each end. Thus, the total number of cuts is exactly the same as the length of the outer face. $\square$

THEOREM 7.2. *The code size in the encoding of a planar graph by our scheme is exactly equal to the length of the outer face. In terms of the number of nodal points n, the code size grows as $O(\sqrt{n})$, on an average.*

Proof Sketch:

In a road network, the bounded degree condition together with the integer distance condition implies that there cannot be an arbitrarily large number of nodes in a given area. Further, the bounded edge weight condition means that there cannot be an arbitrarily small number of nodes in a given area (given that there is at least one node in the area). This means that the $nodes/area$ is bounded above and below by real numbers. Thus the total area occupied by an embedding of the planar graph grows as $O(n)$. Thus, since the planar graph is 2-connected, the average number of edges on the outer face (the perimeter) is $O(\sqrt{n})$. So, for a road network, where the length of an edge is bounded, the length of the outer face is $O(\sqrt{n})$. The code size is exactly equal to the total number of cuts. From Lemma 7.1, the total number of cuts is equal to the length of the outer face. Thus the code size is equal to the length of the outer face and grows as $O(\sqrt{n})$.

COROLLARY 7.3. *The total space required to store the encodings of the points is $O(n\sqrt{n})$.*

COROLLARY 7.4. *The time required for computation of shortest distance is $O(\sqrt{n})$.*

THEOREM 7.5. *The time required for the preprocessing phase is $O(n\sqrt{n})$.*

PROOF. The total time required for the preprocessing phase is the sum of the time required for finding the cuts and the time required for allotting the codes. There are $O(\sqrt{n})$ cuts. The number of edges is linear in the number of vertices (by Euler's formula for planar graphs [2]). Since each edge has exactly two cuts through it, the average number of edges per cut is $O(\sqrt{n})$. Thus the total time required for finding all the cuts is $O(\sqrt{n}) \times O(\sqrt{n}) = O(n)$. Then for allotting the actual codes, for each cut one would allot one bit to each vertex of the graph. Thus time required for this is n times the number of cuts, which is $O(\sqrt{n})$. Thus the asymptotic time complexity of the entire preprocessing phase is $O(n) + O(n\sqrt{n}) = O(n\sqrt{n})$. $\square$

	Time to Generate Encodings (seconds)					
Number of Nodes	Edge Density in Graph (% of maximal)					
	50	60	70	80	90	100
40,000	0.638	1.397	1.198	1.357	1.609	1.048
50,000	1.744	1.142	2.014	1.618	1.321	1.925
60,000	2.246	1.431	1.836	1.635	1.406	1.957
70,000	1.619	1.941	2.340	1.831	1.942	2.418
80,000	1.821	2.726	2.774	2.333	2.578	1.985
90,000	2.290	2.846	3.213	2.997	2.614	2.559

Table 2: Time (secs) to generate codings for planar graphs

Although our encoding scheme grows sublinearly with respect to the number of nodes, the actual performance depends upon the length of the roads. Therefore, we conducted experiments on graphs, simulated to be abstraction of actual road networks with varying edge lengths. Our experiments in Section 7.1 demonstrate superior performance over different lengths of road segment.

7.1 Experimental Evaluation

We implemented our coding method in C++ using the LEDA graph library [10], and conducted a series of experiments on a Linux box with an Intel Pentium IV 1.4 GHz processor and 1 GB of RAM. We generated a large number of random road networks, and measured both, the time required for generating the encodings for these graphs, as well as their size. The edge weights were randomly distributed, with an average lengths of 5 and 10 units.

First, we demonstrate the efficiency of our coding algorithm for graphs of varying sizes and edge densities. A maximal planar graph with n nodes has $3n - 6$ edges [2] (represented here as M_n). We varied the edge density from 50% of M_n to 100% of M_n. We varied the total number of nodes (real and virtual) from 40,000 to 100,000.

Table 2 shows that it takes less than 3 seconds to process 90000 nodes (real and virtual) with approximately 180000 edges. This implies that even for large road networks, we can find all the cuts in extremely short order. Figure 7 represents length of the resulting codes in bytes for these same graphs. The size of the encoding varies from under 250 bytes to under 550 bytes.

The efficiency of our method is now readily apparent. Given a road map with 90,000 nodal points in all (corresponding to roughly 9000 real nodes), the shortest distance can be obtained by computing the XOR of around 450 pairs of bytes. It is also clear from the graph that the scheme works uniformly well for all edge densities.

7.1.1 Comparison With Table Lookups

An alternative approach would be to store precomputed shortest distances, doing table lookups as needed. Figure 8 compares the time required by our method with the time to find all pairs of shortest paths in the planar graph. Since shortest path computations run in $O(n^2 logn)$ time [4], it scales poorly, and is impractical even for small road networks. Our algorithm runs in time $O(n\sqrt{n})$, and takes just a few seconds to assign codes. Experimentally we also see that it scales nicely with the number of nodes in the graph.

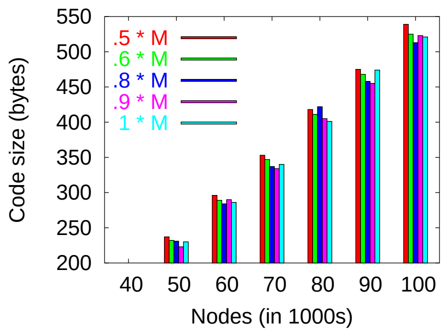


Figure 7: Code size v/s total nodes (real + virtual)

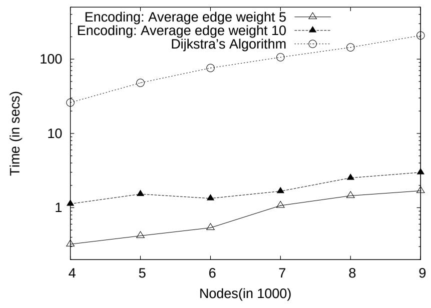


Figure 8: CPU requirements: Comparison with Table Lookup. Note logarithmic scale.

Figure 9 compares the space required for a table-lookup scheme with the space required by our method. Even assuming that table entries are only 2 bytes long, we see that space requirement for table lookup rises dramatically relative to the space requirement for our method. Our methods appears to win hands-down in terms of both preprocessing time and the size of the database required. Yet it can still compute shortest distances extremely quickly.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have addressed the problem of answering spatiotemporal queries on objects moving along constrained trajectory systems that closely simulate road networks. The solution is extremely efficient and practical, and enhances the current state of the art significantly. It is based on a very efficient method for computing shortest distances on a large class of planar graphs, based on hypercube embedding. The method for assigning encodings to the graph nodes is very fast, and our encodings are very space-efficient.

Based on our encoding scheme, we provide elegant and efficient data structures and algorithms for handling a vast array of spatial and spatiotemporal queries over very large road-like networks. We have shown experimentally that our algorithms perform very well.

Possibilities for future work include generalizations of these techniques and those in [6] to handle more complex classes of graphs, and ultimately, to real road networks.

9. ACKNOWLEDGEMENTS

This work was supported in part by grants from the Tata Consultancy Services, Inc. and the Digital Media Innovations Program of the University of California, and by award FTN F30602-01-2-0535 from DARPA.

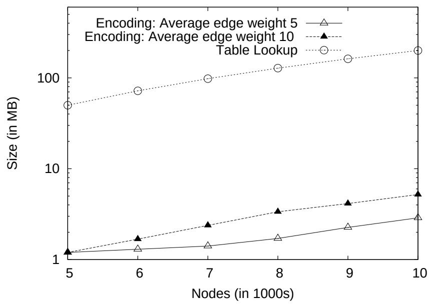


Figure 9: Storage requirements: Comparison with Table Lookup. Note logarithmic scale.

10. REFERENCES

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *In Proc. of the 19th ACM Symp. on Principles of Database Systems (PODS)*, pages 175–186, 2000.

[2] Bondy and Murthy. Graph theory with applications. 1976.

[3] V. Chepoi and M. Deza. Clin d'oeil on l_1-embeddable planar graphs. *In Discrete Appl. Math. 80*, pages 3–19, 1997.

[4] Cormen, Lieserson, and Rivest. Introduction to algorithms. 1990.

[5] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *In SIAM J. Computing 16(6)*, pages 1004–1022, 1987.

[6] C. Gavoille, D. Peleg, S. Perennes, and R. Raz. Distance labeling in graphs. In *Symposium on Discrete Algorithms*, pages 210–219, 2001.

[7] R. Guting, M. Bohlen, M. Erwig, C.Jensen, N. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *In ACM TODS, Vol. 25, No 1*, pages 1–42, 2000.

[8] G. Kollios, D. Gunopulos, and V. Tsotras. On Indexing Mobile Objects. *In Proc. of the 18th ACM Symp. on Principles of Database Systems (PODS)*, pages 261–272, June 1999.

[9] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15:215–245, 1995.

[10] K. Mehlhorn, S. Naher, and C. Uhrig. The LEDA platform of combinatorial and geometric computing. In *Automata, Languages and Programming*, pages 7–16, 1997.

[11] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases, 2003.

[12] S. Saltenis, C. Jensen, S. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. *In Proceedings of the ACM SIGMOD*, pages 331–342, May 2000.

[13] C. Shahabi and M. R. K. M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *In the 10th ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS'02), McLean, VA*, 2002.

[14] Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. of the VLDB*, 2003.

[15] US Census Bureau. TIGER. *http://tiger.census.gov/*.

[16] M. Vazigiannis and O. Wolfson. A spatiotemporal model and language for moving objects on road networks. In *Advances in Spatial and Temporal Databases*, 2001.