Optimal Traversal Planning in Road Networks with Navigational Constraints*

Leyla Kazemi
Information Laboratory
Computer Science
Department
University of Southern
California
Los Angeles, CA 90089-0781
lkazemi@usc.edu

Cyrus Shahabi
Information Laboratory
Computer Science
Department
University of Southern
California
Los Angeles, CA 90089-0781
shahabi@usc.edu

Mehdi Sharifzadeh †
Google Inc.
Santa Monica, CA
mehdish@google.com

Luc Vincent
Google Inc.
Mountain View, CA
luc@google.com

ABSTRACT

A frequent query in geospatial planning and decision making domains (e.g., emergency response, data acquisition, street cleaning), is to find an optimal traversal plan (OTP) that traverses an entire area (e.g., a city) by navigating through all its streets. The optimality is defined in terms of the time it takes to complete the traversal. This time depends on the number of times each street segment is traversed as well as the navigation time such as the time spent on changing direction at each intersection.

While the problem roots in the classic problems of graph theory, real-world geospatial constraints of road network introduce new application-specific challenges. In this paper, we propose two algorithms to find OTP of a directed road network. Our greedy algorithm employs a classic graph traversal algorithm. During the traversal, it utilizes a set of heuristics at each intersection to minimize the total travel time. Our near-optimal algorithm, however, reduces an OTP problem to an Asymmetric Traveling Salesman Problem (ATSP) by extracting the dual graph of the original network in which each edge is represented by a graph node. Using an approximate solution for ATSP, our algorithm finds a near optimal answer. Our experiments using real-world road networks verify that our near-optimal algorithm outperforms the greedy algorithm in terms of the overall cost of its generated traversal by a factor of two, while its complexity is tolerable in real-world cases.

*This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), IIS-0238560 (PECASE), IIS-0324955 (ITR), and unrestricted cash gifts from Google and Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

†The work is completed when the author was studying PhD at USC's InfoLab

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

1. INTRODUCTION

In this paper, we study the problem of optimal traversal of real-world road networks with the presence of geographic constraints. Suppose an online mapping service wants to provide street level imagery of an area for online map viewers. To achieve this, a photographer must drive all the streets of that designated area to take pictures. A *blind* total traversal of the streets takes a considerable amount of time consisting of the travel time of each street segment as well as the *navigation* time such as the wait time at each intersection. While the former time component is unavoidable as each street segment *must* be driven at least once, the navigation time (wait or turn time) depends on the order in which segments are driven. These navigational constraints forced by relative positioning of the intersecting streets make the cost of traversing the same set of streets significantly different for different traversal orders. Given a road network, the focus of this paper is finding a traversal of minimum cost (e.g., time or distance) that visits the entire road network. This problem, termed as *Optimal Traversal Planning*[1] *(OTP)*, has applications in different areas. In emergency response, a responder might want to optimally decontaminate a polluted area by spreading neutralizer materials in each street. In urban planning, a planner looks for the fastest way of cleaning the streets of a neighborhood using a street sweeping machine. In sensing, a data collector wants to measure the precise geocoordinates of each end point of streets using a field vehicle that drives through the streets.

The literature on graph theory includes many variants

[1]Even though we are looking for the *optimal* traversal and hence the term *optimal* in the name of the problem, it is worth noting that our proposed solution is near-optimal.

of the problem of graph traversal with respect to different constraints. One of the classic problems in graph theory is the problem of *Chinese Postman Tour (CPT)* [6] in which a postman needs to travel along every road, with the least possible cost, to deliver mails. Another problem in graph theory is the *Traveling Salesman Problem (TSP)* [1, 4]. Given a collection of cities and the cost of travel between each city pairs, TSP is to find the cheapest way of visiting all of the cities and returning to the starting point (i.e., finding a Hamiltonian cycle with the least weight). If the distance between the cities is not symmetric (i.e., d(city1,city2) $\neq$ d(city2,city1)), the problem is referred to as the *Asymmetric Traveling Salesman Problem (ATSP)*.

However, the typical solutions to graph theory problems do not directly apply to road-networks. This is mainly due to the following two characteristics of road-networks that distinguish them from regular graphs. First, most of the graph-theory solutions, such as ATSP and CPT, assume a strongly connected graph. However, given a geographical area of interest, the graph of road-network is not usually strongly connected (unless the area is fully contained in, say, an island!). Second, the *navigational* properties of road-networks, such as *turns*, are not captured in their graph counterparts. For example, if a vertex is connected to two edges, as far as the graph model concerns, the cost of taking either of the edges only depends on the weight of that edge (e.g., its length). However, in a road-network, the cost of a *"left turn"* is higher than that of a *"right turn"*. This extra cost is not directly captured in the graph model of the road-network. In this paper, we provide solutions to both of the above issues on the way of solving the general OTP problem in road-networks. Note that even though our focus is on OTP, the two abovementioned issues arise in any other application that conceptualizes a road-network as a connected graph in order to utilize the graph theory solutions.

Therefore, first we propose a general technique to transform a partial road-network to a fully connected graph. Next, we propose two different approaches for OTP, building on top of the classical solutions in graph theory. Our first approach extends a solution to the classic problem of *Chinese Postman Tour* in order to incorporate the additional navigational costs. This classic algorithm first transforms the graph into an Eulerian graph, a graph in which at least one Eulerian cycle exists. Subsequently, it finds the Eulerian cycle with the minimum cost (i.e., the optimal traversal plan). We modified this algorithm to also consider the navigational cost when making local planning decisions, in a greedy manner. Consequently, the final traversal plan is no longer optimal due to this local optimization. Our second algorithm, however, results in a near-optimal solution by taking a global optimization approach to find the optimal traversal. With this approach, we first reduce the original network to a dual graph in which each original edge is represented by a vertex. Now we can add the navigational costs (e.g., *turns*) as edges between these vertices. Consequently, in this dual graph, the objective is changed to visiting all the vertices (as opposed to traversing all the edges). Hence, the solution to Asymmetric Traveling Salesman Problem (ATSP) of this dual graph is equivalent to the solution to the OTP problem in the original graph. However, since ATSP is NP-Complete we use a classic approximation algorithm for ATSP to find a near-optimal answer to the OTP problem. Note that by this problem reduction, we also show that OTP is NP-complete.

Our empirical experiments with real-world datasets show that our greedy approach is very fast even for large datasets (it operates in about 13 seconds for our largest dataset). Our near-optimal approach requires more time to build the traversal (about 3 minutes for the largest dataset which is an area consisting of 1600 intersections). The reason is that it aims to solve a global optimization problem comparing to the greedy approach in which only heuristics are applied to achieve local optimizations. This expensive CPU cost in the off-line process of building the traversal pays off when our near-optimal algorithm outperforms the greedy algorithm in terms of the overall cost of its generated traversal by a factor of two (For the same dataset stated above, our greedy approach generates a 62-hour plan, while the near-optimal produces a plan which takes only 32 hours). That is, using the near-optimal solution significantly saves the time of the actual traversal of the road network. We also show that this time is very close to a lower-bound for the time of any traversal of the same network.

The reminder of the paper is organized as follows. Section 2 formally defines the OTP problem. In Section 3, we propose our two different solutions and prove their correctness. Section 4 includes our experimental results and Section 5 discusses the conclusions.

2. DEFINITIONS

Consider the weighted directed graph $G = (V, E)$ as the two sets V of vertices, and $E \subseteq V \times V$ of edges. We represent each edge of E, directly connecting vertices u and v, as the ordered pair $[u, v]$. Each vertex v represents a 2-d point $(v.x, v.y)$ in a geometric space. Hence, each edge is also a line segment in the space. A numeric weight (cost) w_{uv} is associated with the edge $[u, v]$.

Definition 1: Given a graph G, a *path* P from v_1 to v_n is an ordered set of vertices P=$\{v_1, v_2, \ldots, v_n\}$ such that v_i is connected to v_{i+1} by the edge $[v_i, v_{i+1}]$ for $1 \leq i < n$. The length of this path is $n - 1$ edges. As shown in Figure 1, $\{b, c, f\}$ is a path of length two from b to f. We refer to a path as *cycle* iff we have $v_1 = v_n$.

Definition 2: Given the consecutive edges $[u, v]$ and $[v, w]$, we define *navigation*, $nav(u, v, w)$, as the relative positioning of the second edge $[v, w]$ with reference to the first edge $[u, v]$. Graphs, in their classic definition, do not hold such network related features. Thus, these features should be incorporated into the graph model. Without loss of generality, we assume four kinds of navigation: left, right, straight, and u-turn which are easily computed based on the coordinates of u, v, and w. Alternatively, if the turn restrictions were available from the dataset, we can utilize them as well.[2]. The *navigational cost*, denoted as $ncost(u, v, w)$, is the cost assigned to the navigation $nav(u, v, w)$. Throughout the paper, without loss of generalization and for the purpose of illustration, we assume the costs 60, 10, 0, and 80 seconds for the left, right, straight, and u-turn navigations, respectively. These costs are used only as proof of concept and might be different from the real costs in the road networks. In Figure 1, we have $nav(b, c, f) = Right$ with $ncost(b, c, f) = 10$ *secs*

[2]For example, the NAVTEQ dataset has additional information about the network restrictions

as f is located on the right side of the edge $[b, c]$.

Definition 3: Given a path P=$\{v_1, v_2, \ldots, v_n\}$ where $n > 1$, we define *Path Cost* of P, *pcost*(P), as the sum of all the edge costs and the navigational costs of all pairs of consecutive edges in P. Formally, for the path P, we have,

$$pcost(P) = w_{v_1 v_2} + \sum_{i=1}^{n-2}(ncost(v_i, v_{i+1}, v_{i+2}) + w_{v_{i+1} v_{i+2}})$$

In Figure 1, consider the cycle $C(e) = \{e, f, i, h, e\}$. There are three right turns in this cycle. Hence, the cost of $C(e)$ is calculated as follows:
pcost$(C(e))$ = 50+10+30+10+80+10+90=280.
For vertices u and v, we use P_{uv} to denote the *shortest path* from u to v in G; the path P=$\{u, \ldots, v\}$ with minimum path cost *pcost*(P). We also refer to P_{uu} as the shortest path (cycle) from u to itself which goes through at least one edge.

Problem Definition: Given a weighted directed graph $G = (V, E)$, and a start vertex s, we want to find the cycle $C(s)$ of minimum cost that traverses (includes) all edges of E. We refer to this cycle as the **Optimal Traversal Plan (OTP)** of the graph G. In Figure 1, the following cycle $C(a)$ starts from a and traverses all the edges with the cost of 1950: $C(a)$=$\{a, b, c, f, i, h, e, f, i, h, g, d, e, b, c, f, i, h, e, h, g, d, a\}$. For a directed graph G, such a cycle exists iff G is *strongly connected*; there exists a path between any two vertices u and v. Figure 1 shows an example of such a graph. In our real-world motivating application, the graph G represents the real road network of a city. Hence, it is reasonable to assume that all the road segments are reachable from each other. Otherwise, there would be a region that is isolated from other places (such as an island) that could not be reached by any means. However, there are still other problems in the connectivity of the graph: since an area of road network is assumed to be selected, some streets might get cut off. Therefore, there is a high probability that the built-in graph would not be strongly connected. The solution to this will be addressed in the next section.

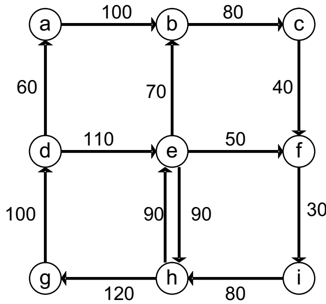


Figure 1: Weighted directed graph $G = (V, E)$

3. SOLUTIONS

3.1 Greedy Solution(GR)

In this section, we propose our greedy algorithm utilizing the following classic results. Given a directed graph G_e, suppose that there exists a cycle C in which each edge of G_e is visited exactly *once*. In graph theory, such a graph G_e is termed as an *Eulerian graph* and the cycle C is denoted as an *Eulerian cycle* of G_e [3]. Given an Eulerian graph, the classic algorithm of Fleury [3] easily finds the Eulerian cycle C. Our greedy algorithm to find OTP of a graph G employs these results as follows. First, with minimum changes, it transforms the graph G to an Eulerian graph G_e (graph balancing in Section 3.1.1). That is, the algorithm adds *virtual* edges to the graph to facilitate the possibility of traversing all the edges only once. Subsequently, it employs Fleury's algorithm to find the Eulerian cycle C of G_e as the OTP of G (greedy traversal in Section 3.1.2).

While our greedy algorithm is similar to the classic solution of CPT problem (see Section 1), there are major differences: 1) Comparing to CPT, OTP is a road network problem which requires more information to be incorporated into a simple graph. This includes navigations as well, while there is no such notion in the classic definition of CPT. 2) Once G is transformed into an Eulerian graph G_e, any Eulerian cycle of G_e is a solution of CPT. However, because of the presence of navigational costs in OTP, our algorithm requires to find an Eulerian cycle of *minimum* cost. That is, solving OTP requires solving an optimization problem. Hence, our greedy algorithm tries to employ heuristics during the iterations of Fleury's algorithm to visit those vertices that require navigations of less cost. This policy results in a close-to-minimum-cost cycle.

3.1.1 Graph Balancing

The first phase of the greedy algorithm transforms the graph G to an Eulerian graph. To describe the definition of such a graph, we first define the following notations. Given a vertex v in a directed graph G, the number of edges $[u, v]$ going into v is called the *in-degree* of v, denoted by $d^-(v)$. Similarly, the number of edges $[v, u]$ coming out of v is the *out-degree* of v, denoted by $d^+(v)$. We define the *degree* of v, $\delta(v) = d^-(v) - d^+(v)$, as the difference of in-degree and out-degree of the vertex v. In Figure 1, we have $d^-(f) = 2$, $d^+(f) = 1$, and $\delta(f) = 1$. We refer to the vertex v as a *balanced* vertex iff we have $\delta(v) = 0$ (e.g., vertex a in Figure 1). Based on the above notations, the definition of an Eulerian graph follows:

THEOREM 1. *A directed graph G is Eulerian iff G is strongly connected and all of its vertices are balanced [3].*

As stated in Section 2, our graph G might not be strongly connected because it is built based on a randomly selected region of road network (some road segments get cut which can create several disconnected areas).This happens mostly for the intersections on the boundary of the selected region. In order to solve this problem, we use a simple *Pruning* approach to remove unreachable vertices of the graph. The pruning is based on the fact that there should be no intersection with only incoming road segments or just outgoing road segments. Thus, we remove any such intersection when building the graph G.

After building the strongly connected graph G,to make it Eulerian we only need to make its vertices balanced. Figure 2 shows the pseudo-code of this phase. Let D^+ and D^- be the sets of vertices of G with positive and negative degrees, respectively (lines 1-2). Note that D^+ and D^- may not have the same size but the sum of their vertex degrees is zero.

Algorithm Graph Balancing (graph G=(V, E))
01. let $D^+ = \{$all $v \subset V$ where $\delta(v) > 0\}$;
02. let $D^- = \{$all $v \subset V$ where $\delta(v) < 0\}$;
03. set $P = \{$find a matching between D^- and D^+ with the least total cost so that all the vertices become balanced$\}$;
04. for each vertex pair $(u, v) \subset P$
05. create a virtual edge $[u, v]$ with the weight $w_{uv} = pcost(P_{uv})$;
06. add virtual edge $[u, v]$ to G;

Figure 2: Graph Balancing Algorithm

The idea is to add some virtual edges between the vertices of these two sets, so that all of them become balanced. Then, the weight associated to each of these virtual edges would be the cost of the shortest path between the endpoints of the edges in G. Furthermore,the specific choice of virtual edges between D^+ and D^- must be added to G that add minimum cost to the edges of G. The reason is to add minimum possible extra cost to the cost of the ultimate solution to our main OTP problem. Therefore, line 3 in Figure 2 involves an optimization problem. Once, the algorithm finds the set of optimal virtual edges, it adds them to G. The new graph G that includes the original edges of G as well as the virtual edges is the corresponding transformed Eulerian graph.

We illustrate the graph balancing phase using the graph G of Figure 1. In the figure, we have D^+=$\{b, f\}$, D^-=$\{d, e\}$, and the degrees of each of unbalanced vertices are either 1 or -1. There are two alternatives to choose the virtual edges: 1) connect b to d and f to e, or 2) connect b to e and f to d. Here, both choices add equal costs to graph G: $pcost(P_{bd})$+$pcost(P_{fe})$=480+220=$pcost(P_{be})$ +$pcost(P_{fd})$=350+350. Hence, our algorithm can choose either of them to be added to G. Figure 3 shows the resulting Eulerian graph when we add virtual edges $[b, e]$ and $[f, d]$ (shown as dotted lines) to the graph of Figure 1.

3.1.2 Greedy Traversal

In the first phase, we transformed our original graph to an Eulerian graph. In this section, with a greedy approach we find an Eulerian cycle of close-to-minimum cost in the transformed graph. Here, we extend Fluery's classic algorithm to find an Eulerian cycle of close-to-minimum cost. To keep the result close to minimal, we incorporate heuristic guidelines to the original algorithm. That is, when we have more than one alternate edge to traverse, we traverse the one that imposes the navigation of minimum cost on the final result.

Figure 4 shows the pseudo-code of our algorithm. It starts from the starting vertex s and iteratively traverses the graph by visiting an outgoing edge of the current vertex. Each visited edge is added to the partial result OTP. Subsequently, the partial cost *pcost* of OTP is also updated. We describe the algorithm using the transformed graph shown in Figure 3. Table 1 shows the trace of the algorithm. Starting from the vertex a, it adds the only unvisited outgoing edge of a (i.e., $[a, b]$) to the partial result OTP. The cost of OTP is updated to $w_{ab} = 100$ (lines 2-7). The next iteration examines the last visited vertex b. Both outgoing edges of b, $[b, c]$ and $[b, e]$, have not been visited. To choose the edge to visit, the algorithm first computes the shortest path P_{ba} from b to the starting vertex a. To perform this computation, it employs Dijkstra's algorithm. Throughout the iterations, at every vertex v, we refer to the first edge of P_{va}(i.e., a is the starting vertex) as the *bridge edge*. As an example, the edge

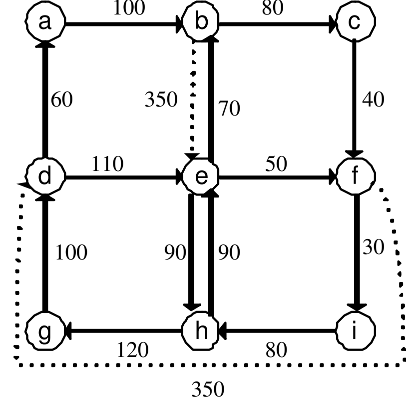


Figure 3: G_e, Eulerian graph of graph G

$[b, c]$ is the first edge of P_{ba} and therefore is a bridge edge. The idea is to leave the bridge as the last edge to be traversed from each vertex. Therefore, if there is a *non-bridge* unvisited edge from b, the algorithm visits that edge (lines 8-19). Otherwise, it visits the bridge edge. Here, the algorithm visits $[b, e]$ and adds it to OTP. Now, to update the cost, we need to add the navigational cost $ncost(a, b, e)$ and w_{be}. While the latter value is simply the weight of the virtual edge $[\mathbf{b}, \mathbf{e}]$ (i.e., 350), computation of the former value is tricky. As $[\mathbf{b}, \mathbf{e}]$ represents the shortest path P_{be}, the navigation between $[a, b]$ and the virtual edge $[\mathbf{b}, \mathbf{e}]$ is in fact the navigation between $[a, b]$ and the first non-virtual edge in P_{be} (here $[b, c]$). Hence, the cost $ncost(a, b, e)$ is equal to $ncost(a, b, c) = 0$.

The fourth iteration examines e and all its outgoing unvisited edges $[e, b]$, $[e, f]$, and $[e, h]$. Computing the shortest path P_{ea}, it finds that $[e, h]$ is the bridge edge and must be skipped for now. Now, either $[e, b]$ or $[e, f]$ must be visited. To choose, the algorithm applies our heuristics. It first computes the navigational costs $ncost(b, e, b)$ and $ncost(b, e, f)$ where $[\mathbf{b}, \mathbf{e}]$ is the last visited edge. Again, as $[\mathbf{b}, \mathbf{e}]$ is a virtual edge, the navigation between $[\mathbf{b}, \mathbf{e}]$ and $[e, b]$ (or $[e, f]$) is actually the navigation between the last edge in P_{be} (namely, $[h, e]$) and $[e, b]$ (or $[e, f]$). Hence, we have $nav(b, e, b)$=$nav(h, e, b)$= *Straight* and $nav(b, e, f)$= $nav(h, e, f)$= *Right*. Therefore, as $ncost(b, e, b)$=$ncost(h, e, b)$ = 0, the algorithm visits $[e, b]$ instead of $[e, f]$. It updates the cost by adding $ncost(b, e, b)$=0 and w_{eb}=70.

The next 12 iterations of the algorithm use the same heuristics to traverse the unvisited edges of the graph. The algorithm stops when it revisits the starting vertex a and all edges of G_e are visited (line 10). In our example, at the end of the 16th iteration the final OTP is computed as: OTP=$\{a, \mathbf{b}, \mathbf{e}, b, c, f, i, h, e, \mathbf{f}, \mathbf{d}, e, h, g, d, a\}$. The final step of our algorithm, expands the virtual edges of computed OTP to their corresponding shortest paths (line 21). For example, it replaces $[\mathbf{b}, \mathbf{e}]$ in the above OTP with the vertices of the path P_{be}=$\{b, c, f, i, h, e\}$ as there is no actual direct connection from b to e in G. Hence, the greedy algorithm returns the following cycle as the result: $\{a, \mathbf{b}, \mathbf{c}, \mathbf{f}, \mathbf{i}, \mathbf{h}, \mathbf{e}, b, c, f, i, h, e, \mathbf{f}, \mathbf{i}, \mathbf{h}, \mathbf{g}, \mathbf{d}, e, h, g, d, a\}$.

3.2 Near-Optimal Solution (nOpt)

```
Algorithm Greedy Traversal(graph G_e,
                          starting vertex s)
01. OTP={s}; Cost=0;
02. [s, u] = bridge edge of P_ss
03. if there exists an unvisited edge [s, v] such that v ≠ u
04.   append v to OTP; add w_sv to Cost;
05.   mark [s, v] as visited;
06. else
07.   append u to OTP; add w_su to Cost;
08.   mark [s, u] as visited;
09. p, v = the last two vertices of OTP;
10. while v ≠ s or there exists an unvisited edge
11.   [v, w] = bridge edge of P_vs
12.   from all unvisited edges [v, x] such that x ≠ w
13.     [v, n] = the edge with minimum ncost(p, v, x);
14.   if [v, n] is not null
15.     append n to OTP;
16.     add ncost(p, v, n) + w_vn to Cost;
17.     mark [v, n] as visited;
18.   else
19.     append w to OTP;
20.     add ncost(p, v, w) + w_vw to Cost;
21.     mark [v, w] as visited;
22.   [p, v] = the last two vertices of OTP;
23. replace any virtual edge [u, v] with the vertices of
    shortest path P_uv
24. return OTP;
```

Figure 4: Greedy algorithm

Itr.	OTP	Cost
01	a	0
02	ab	$\mathbf{0} + 100 = 100$
03	a**be**	$\mathbf{100} + 0 + 350 = 450$
04	a**be**b	$\mathbf{450} + 0 + 70 = 520$
05	a**be**bc	$\mathbf{520} + 10 + 80 = 610$
06	a**be**bcf	$\mathbf{610} + 10 + 40 = 660$
07	a**be**$bcfi$	$\mathbf{660} + 0 + 30 = 690$
08	a**be**$bcfih$	$\mathbf{690} + 10 + 80 = 780$
09	a**be**$bcfihe$	$\mathbf{780} + 10 + 90 = 880$
10	a**be**$bcfihef$	$\mathbf{880} + 10 + 50 = 940$
11	a**be**$bcfihe$**fd**	$\mathbf{940} + 10 + 350 = 1300$
12	a**be**$bcfihe$**fd**e	$\mathbf{1300} + 10 + 110 = 1420$
13	a**be**$bcfihe$**fd**eh	$\mathbf{1420} + 10 + 90 = 1520$
14	a**be**$bcfihe$**fd**ehg	$\mathbf{1520} + 10 + 120 = 1650$
15	a**be**$bcfihe$**fd**$ehgd$	$\mathbf{1650} + 10 + 100 = 1760$
16	a**be**$bcfihe$**fd**$ehgda$	$\mathbf{1760} + 0 + 60 = 1820$

Table 1: Trace of greedy algorithm

The problem of using the previous solution is that it tries to find OTP of a graph G by a greedy approach, which might not be optimal. The reason is that at each step, the algorithm tries to employ some heuristics to explore those edges that appear to have navigations of less cost (i.e., locally optimal), which might not necessarily results in a global optimal answer. In this section, we propose our near-optimal algorithm, where we try to reduce our problem to ATSP (see Section 1). Similar to our greedy solution, this algorithm also first transforms the given graph G to an Eulerian graph G_e. Subsequently, it transform G_e to a particular graph G_l^* (*Extended Line Graph* of G) in which every vertex v in G_l^* represents a possible turn between any two consecutive edges in G_e. Once such graph G_l^* is generated (Section 3.2.1), our OTP algorithm requires to traverse all the *vertices* in G_l^* to equivalently traverse all the *edges* in G_e. Based on definitions introduced in Section 1, this is an ATSP problem. Since ATSP is an NP-Complete problem [7], and therefore, a polynomial time solution is unlikely to exist, applying some heuristics to the problem can give a near-optimal solution. Utilizing such heuristics, we try to solve the OTP problem as well (Section 3.2.2). We use a classic $O(\log n)$ approximation algorithm (n is the number of vertices), termed *Patched Cycle Cover* [2, 5], to find this solution. In sum, our near-optimal solution consists of the following phases:

1. Building the Eulerian graph G_e of G using graph balancing algorithm (Section 3.1.1).
2. Building the extended line graph G_l^* of G_e (Section 3.2.1) by incorporating the navigational properties of G_e into G_l^*.
3. Solving ATSP for the graph G_l^* (Section 3.2.3).

Here, we discuss the last two phases of this approach.

3.2.1 Building the Extended Line Graph

The second phase of our near-optimal solution generates the extended line graph G_l^* from the Eulerian graph G_e of our original graph G. To define the extended line graph, we first define the following. In graph theory, the *Line Graph* G_l of a directed graph G is a graph such that each vertex of G_l represents an edge $[v, w]$ of G; and for any pair of vertices s and t in G_l, the edge $[s, t]$ shows that their corresponding edges in G are consecutive.

To incorporate navigational costs of graph G into the line graph G_l, we introduce the **Extended Line Graph** G_l^*. The intuition is to add new vertices in G_l^* such that we can generate a new edge for each possible turn between any two edges in graph G. For a graph G, we build G_l^* as follows. For each edge $[v, x]$ (or $[x, v]$) connecting to vertex v, we create a vertex v_{vx} (or v_{xv}) in G_l^*. In the example of Figure 3, for vertex e we create six vertices $e_{eh}, e_{he}, e_{ef}, e_{eb}, e_{be}$, and e_{de} in G_l^* (Figure 5).

Once we created all vertices of G_l^*, we start adding corresponding edges to G_l^*. For each edge $[u, v]$ in G, we add an edge $[u_{uv}, v_{uv}]$ from u_{uv} to v_{uv} in G_l^*. The weight of this new edge is the same as that of $[u, v]$. Figure 6a illustrates different edges created in G_l^* corresponding to the edges connecting to the vertex e of graph of Figure 3. For example, the edge $[d_{de}, e_{de}]$ with the weight 110 corresponds to the edge $[d, e]$ of the Eulerian graph shown in Figure 3.

The final step in building the extended line graph creates edges in G_l^* that represent navigational costs (correspond-

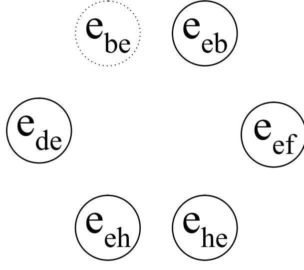


Figure 5: Vertex representation in G_l^*

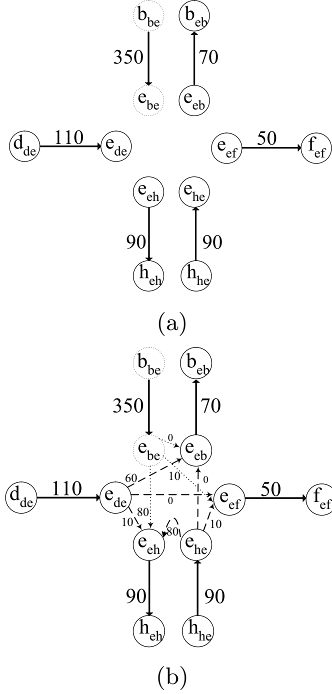


Figure 6: Adding edges in G_l^*

ing to *turns*) in G. Hence, for each two consecutive edges $[x, v]$ and $[v, y]$ in G, we create an edge $[v_{xv}, v_{vy}]$ from v_{xv} to v_{vy} in G_l^*. We assign the weight of this edge to the navigational cost $ncost(x, v, y)$ in G. For the vertex e in Figure 3, we have the following set of 9 pairs of consecutive edges passing through e:

$[d, e], [e, b]$, $[d, e], [e, f]$, $[d, e], [e, h]$, $[h, e], [e, b]$, $[h, e], [e, f]$, $[h, e], [e, h]$, $[\mathbf{b}, \mathbf{e}], [e, b]$, $[\mathbf{b}, \mathbf{e}], [e, f]$, $[\mathbf{b}, \mathbf{e}], [e, h]$

For each of these edge pairs, we create one edge in G_l^* as stated above (see dotted edges in Figure 6b). For instance, corresponding to the edges $[d, e]$ and $[e, b]$, we create an edge $[e_{de}, e_{eb}]$ from e_{de} to e_{eb} with the weight $ncost(d, e, b) = 60$ as the navigation from $[d, e]$ to $[e, b]$ is Left. Notice that similar to Section 3.1.2, computing the navigational cost $ncost(x, v, y)$ is tricky when either of $[x, v]$ or $[v, y]$ is a virtual edge in G. For example, for the edge pairs $[\mathbf{b}, \mathbf{e}], [e, b]$, $[\mathbf{b}, \mathbf{e}], [e, f]$, $[\mathbf{b}, \mathbf{e}], [e, h]$ corresponding to e, the edge $[\mathbf{b}, \mathbf{e}]$ is a virtual edge and represents the shortest path P_{be}. Hence, the navigation between the virtual edge $[\mathbf{b}, \mathbf{e}]$ and $[e, b]$ is in

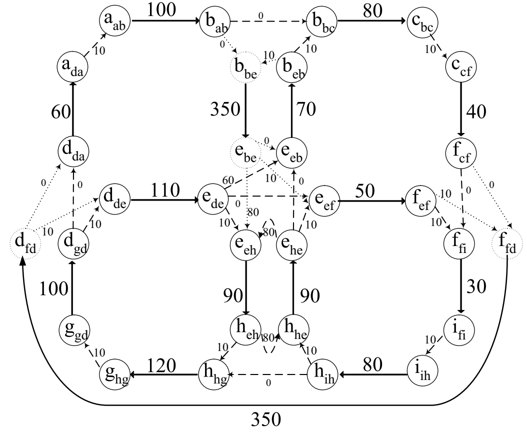


Figure 7: G_l^*, extended line graph of G_e shown in Figure 3

fact the navigation between the last non-virtual edge in P_{be} (here $[h, e]$) and $[e, b]$. Hence, the cost $ncost(b, e, b)$ is equal to $ncost(h, e, b) = 0$. Therefore, we create an edge from e_{be} to e_{eb} with the weight zero.

By applying the above steps to the entire graph of Figure 3, we build the graph of Figure 7. It is clearly seen that the extended line graph G_l^* of a graph G represents all the information of edge costs and navigational costs. That is, there is a one-to-one correspondence between any path in G and a path in G_l^*. Therefore, our algorithm is a lossless complete transformation of G.

LEMMA 1. *Any path P in a graph G can be mapped to a path P^* in G_l^*, the extended line graph of G and vice versa. The path cost of P is equivalent to the sum of weights of edges in P^* (length of P^*).*

3.2.2 Reduction to ATSP

In this section, we show that if we build the extended line graph G_l^* of the Eulerian graph G_e of a graph G, we can reduce the OTP problem on G to the ATSP problem on G_l^*. Then, in Section 3.2.3, we discuss one of approximation algorithms to solve ATSP that we use to solve our OTP problem. First, we show that there is a path that passes through all vertices of G_l^*. That is, the graph G_l^* is *Hamiltonian*.

THEOREM 2. *If a graph G is Eulerian, then the extended line graph of G, G_l^*, is Hamiltonian.*

PROOF. We show that there is a path that passes through all the vertices of G_l^*. As graph G is Eulerian, there is a cycle C that passes through all the edges of G. Hence, as Lemma 1 states, there is a cycle C^* in G_l^* corresponding to C in G with the same cost. Each edge of G corresponds to two unique connected vertices in G_l^*. Therefore, C^* includes the corresponding vertices of each edge of G. As these are the only vertices of G_l^*, the cycle C^* passes through all vertices of G_l^*. That is, G_l^* is Hamiltonian. □

THEOREM 3. *Given a starting vertex s on a graph G, finding an OTP is equivalent to finding a cycle of shortest length that passes through all vertices in the graph G_l^*, the extended line graph of the Eulerian graph G_e of G.*

PROOF. Let C^* be a cycle of shortest length that passes through all vertices of G_l^*. First, similar to the proof of Theorem 2, we can show that C^* corresponds to a cycle C that passes through all edges of G_e. As the length of C^* is the same as path cost of C, the path cost of cycle C is also minimum over all cycles that pass through all edges of G. □

According to theorems 2 and 3, G_l^* is Hamiltonian. Thus, solving OTP on G is equivalent to finding a hamiltonian cycle on G_l^* which is a graph-based analogue of Asymmetric Traveling Salesman problem. Since ATSP is NP-Complete, the following theorem is entailed:

THEOREM 4. *OTP is NP-Complete.*

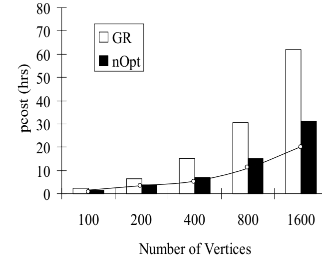
3.2.3 Solving ATSP

Based on the above theorems, solving OTP on G is equivalent to solving ATSP on G_l^*. Since ATSP is an NP-Complete problem, finding its optimal solution is expensive specially for large graphs. However, there are few approximation algorithms that can find a near-optimal solution by utilizing some heuristics. We apply the *Patched Cycle Cover* algorithm. The first step of the algorithm employs the *Assignment Problem (AP)*. Given a directed graph $G = (V, E)$, AP seeks to assign a vertex v to each vertex u, with w_{uv} as the cost of this assignment, so that all the vertices are assigned and the total cost of the assignments is minimized. If the assignment was a single cycle, the optimal result is achieved and the algorithm terminates. Otherwise, the solution is a set of subcycles. Then, the second step uses a patching algorithm by selecting the two biggest subcycles and joining them with the minimum cost. This process is repeated until a single cycle is achieved. When the algorithm terminates, it gives us a hamiltonian cycle of G_l^*, which is easily mapped to an Eulerian cycle of the graph G_e. This cycle is an approximate answer to the OTP problem on graph G.

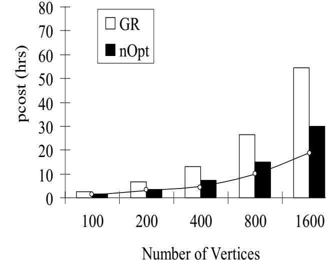
4. PERFORMANCE EVALUATION

We conducted several experiments to evaluate and compare the performance of our proposed approaches. The effectiveness of Greedy (GR) and near-optimal (nOpt) approaches is determined in terms of 1) the CPU cost which is the time it takes to compute the OTP for a given graph, and 2) the path cost (pcost) of the computed OTP. For our experiments, we used a real-world data set obtained from NAVTEQ covering large areas in Los Angeles and San Jose, California. Since our comparing parameter is the data size (number of intersections or vertices of the graph), in our experiments we selected subsets of each area with 5 different sizes (i.e., 100, 200, 400, 800, and 1600 intersections). For each data size, we randomly selected 10 spatially bounded regions of that size in different parts of the two cities. Results are then averaged for each data size. Experiments were run on an AMD Opteron 3.20 GHz processor with 3 GB of RAM.

Our first set of experiments focuses on the effect of graph size on the cost of OTP generated by each of our two approaches. Figure 8a and 8b show this cost in seconds for the regions of different size in cities of Los Angeles and San Jose, respectively. Both figures compare GR and nOpt approaches as the number of the intersections of the input graph increases exponentially. According to these figures,

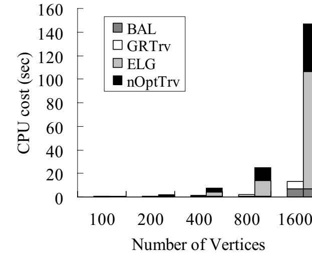


(a) Los Angeles

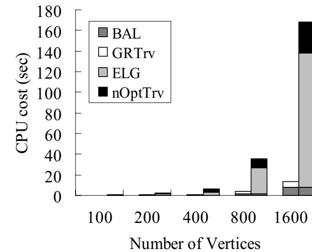


(b) San Jose

Figure 8: pcost versus data size



(a) Los Angeles



(b) San Jose

Figure 9: CPU cost versus data size

the nOpt approach outperforms GR for all graph sizes by an approximate factor of two. In Figure 8a, for a graph of size 1600, GR gives an OTP which takes about 62 hours to traverse, while nOpt generates a 32-hour plan (1.94 times shorter). Here, we assume there is no extra restriction if the plan is more than a workday (e.g., the car needs to park overnight). However, if such restriction exists, it should be either incorporated into the plan or we only provide daily planning. The figures also show the total sum of all edge costs of each graph (the time it takes to traverse each road only once). This value is a lower-bound on the path cost of any cycle that passes through all edges of the input graph. As shown, the cost of OTP generated by nOpt is always close to this lower-bound value (only 1.9 times longer in the worst case). Notice that this bound does not consider the cost of directions participating in any path. Hence, it is not a tight lower-bound. We also conducted the experiment with San Jose dataset. For a dataset of similar size, results shown in Figure 8b comply with those of Los Angeles dataset. GR generates an OTP which takes about 54 hours to traverse, while nOpt returns only a 29-hour plan (a factor of 1.86 improvement).

Our last set of experiments studies the effect of graph size on the CPU cost. Figure 9a and 9b show the CPU cost for both GR and nOpt approaches as the number of the intersections increases exponentially. Tables 2 and 3 show the same results in more detail. This cost is divided into different computation tasks involved with each approach: 1) **BAL** is that portion of the cpu cost used for graph balancing that transforms the original graph G into an Eulerian graph G_e (Section 3.1.1). This time consists approximately 3% of the total cpu cost. 2) **GRTrv** is the cpu cost of finding the greedy traversal (almost 4% of the total cpu cost). 3) **ELG** is the cpu cost of constructing the extended line graph G_l^*. 4) **nOptTrv** is the cpu cost of finding the near-optimal traversal by solving the corresponding ATSP problem. **BAL** is common between the two approaches and does not contribute to the difference in cpu cost. For small datasets, the cpu cost of both approaches are equal and negligible. However, as the datasize increases, the gap between the two approaches increases as well. For a graph of 1600 vertices in Los Angeles, GR computes the OTP in 13 seconds while nOpt requires two minutes and 40 seconds. As in many real-world applications OTP can be pre-computed, the importance of this off-line computation time would be less significant than the actual traversal time shown in the first experiment. For the largest dataset of 1600 vertices, cpu cost is about three minutes which is still reasonable for generating a 32-hour travel plan as the extra two minute off-line computation saves about 30 hours of traversal time. On the other hand, if we want to compare these results with those computed by optimal solution, with the $O(n!)$ complexity of ATSP, it will take us years of computation.

CPU cost (sec)		Number of Vertices				
		100	200	400	800	1600
Greedy	BAL	0.01	0.03	0.4	0.48	6.6
	GRTrv	0.4	0.62	0.8	1.62	6.51
nOpt	BAL	0.01	0.03	0.4	0.48	6.6
	ELG	0.31	0.68	3.81	13.62	100
	nOptTrv	0.59	1.16	3.17	10.94	40.18

Table 2: CPU cost versus data size in Los Angeles

CPU cost (sec)		Number of Vertices				
		100	200	400	800	1600
Greedy	BAL	0.03	0.2	0.36	1.69	8.11
	GRTrv	0.23	0.36	0.48	2.03	5.14
nOpt	BAL	0.03	0.2	0.36	1.69	8.11
	ELG	0.23	1.29	3.12	24.83	130
	nOptTrv	0.5	1.23	2.97	9.1	30.32

Table 3: CPU cost versus data size in San Jose

5. CONCLUSIONS

In this paper, we studied the problem of optimal traversal of real-world networks in the presence of geographic constraints. We proposed two alternative solutions to find OTP of a directed network, the Greedy solution (GR) and the Near-Optimal solution (nOpt). GR extends a solution for CPT by utilizing a set of heuristics to minimize the total traversal cost considering the navigational constraints. On the other hand, nOpt reduces the OTP problem to ATSP and tries to find the near optimal solution using an approximate answer. Our experiments showed the superiority of nOpt over GR in terms of the overall cost of the generated traversal (a factor of two), while its complexity is tolerable in real-world scenarios.

In future, we intend to focus more on a dynamic plan for the real- world applications. As an example, consider a driver who is traversing the streets based on a proposed plan. However, he might fail to follow the given plan because of an unexpected street closure or a traffic pattern change which would render the plan inefficient. The algorithm should have the ability to change its plan accordingly in a reasonable time.

6. REFERENCES

[1] N. Biggs, E. K. Lloyd, and R. J. Wilson. *Graph Theory, 1736-1936.* Clarendon Press, 1986.

[2] J. Cirasella, D. S. Johnson, L. A. McGeoch, and W. Zhang. The asymmetric traveling salesman problem: Algorithms, instance generators, and tests. *Lecture Notes in Computer Science*, 2153:32–59, 2001.

[3] R. Diestel. *Graph Theory.* Springer-Verlag, 2000.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[5] G. Gutin and A. Punnen. *Traveling Salesman Problem and Its Variations.* Kluwer Academic Publishers, 2002.

[6] H. W. Thimbleby. The directed chinese postman problem. *Softw., Pract. Exper.*, 33(11):1081–1096, 2003.

[7] W. Zhang. Truncated branch-and-bound: A case study on the asymmetric tsp. *In Working Note of AAAI 1993 Spring Symposium: AI and NP-Hard Problems*, 15(6):160–166, 1993.