Navigation Made Personal: Inferring Driving Preferences from GPS Traces[*]

Daniel Delling
Sunnyvale, USA
daniel.delling@gmail.com

Andrew V. Goldberg
Emerald Hills, USA
avg@alum.mit.edu

Moises Goldszmidt
Palo Alto, USA
moises.goldszmidt@gmail.com

John Krumm
Microsoft Research
jckrumm@microsoft.com

Kunal Talwar
San Francisco, USA
kunal@kunaltalwar.org

Renato F. Werneck
San Francisco, USA
rwerneck@acm.org

ABSTRACT

All current navigation systems return efficient source-to-destination routes assuming a "one-size-fits-all" set of objectives, without addressing most personal preferences. Although they allow some customization (like "avoid highways" or "avoid tolls"), the choices are very limited and require some sophistication on the part of the user. In this paper we present, implement, and test a framework that generates *personalized* driving directions by automatically analyzing users' GPS traces. Our approach learns cost functions using coordinate descent, leveraging a state-of-the-art route planning engine for efficiency. In an extensive experimental study, we show that this framework infers user-specific driving preferences, significantly improving the route quality. Our approach can handle continental-sized inputs (with tens of millions of vertices and arcs) and is efficient enough to be run on an autonomous device (such as a car navigation system) preserving user privacy.

Categories and Subject Descriptors

G.2.1 [**Combinatorics**]: Combinatorial algorithms; I.2.6 [**Learning**]: Parameter Learning

Keywords

shortest path, personalization, cost function, continental road networks, machine learning

1. INTRODUCTION

Navigation systems have become ubiquitous, available on the web, mobile devices, cars, and embedded systems from a variety of providers, including Apple, Baidu, ESRI, Garmin, Google, HERE, Microsoft, PTV, TomTom, and Yandex.

[*]This work was done while D. Delling, A. V. Goldberg, M. Goldszmidt, K. Talwar, and R. F. Werneck were at Microsoft Research Silicon Valley.

SIGSPATIAL '15 November 03 - 06, 2015, Bellevue, WA, USA
© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-3967-4/15/11...$15.00
DOI: 10.1145/2820783.2820808

Given an origin and a destination, these systems generate driving directions by computing shortest paths in a graph modeling the road network. Each arc of the graph represents a road segment and each vertex models an intersection with additional turn information. The notion of "shortest" is more general than one based on distance alone. It is standard practice for these systems to have a *cost function* that takes into account various considerations [23]. Each arc or turn is assigned a nonnegative *cost* according to which the routing engine computes shortest paths.

In current systems, the cost function is defined by the vendor and depends on dozens of static properties of each road segment (physical length, number of lanes, speed limit, historical traffic data, road category, and so on) or turn type (left, right, U-turn, etc.). The standard parameters of the cost function, which determine how much weight to give to each property of the cost function, are meant to be reasonable for the "average" driver. In order to address idiosyn-

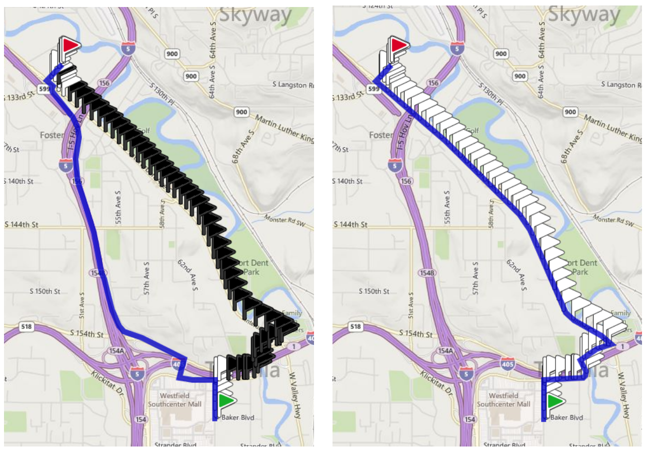

Figure 1: A GPS trace (flags) and routes (lines) computed before (left) and after (right) our optimization. The source of the trip is indicated by a green (lower) flag and the destination by a red (upper) flag. A white flag is a GPS point that can be reasonably matched to the computed route; a black flag cannot. The route computed with default parameters (before optimization) differs significantly from the trace, whereas the one computed after optimization is a perfect match.

cratic preferences, vendors sometimes expose a small subset of these parameters to users, often in binary terms (such as "avoid tolls" or "prefer highways"). This can be too limited, as drivers may have much finer preferences. Indeed, our results allow us to distinguish between several different driving habits. For example, delivery trucks famously avoid making turns against traffic. Others avoid turns at all, preferring simple paths. Ideally, users should be free to specify whatever tradeoffs they wish to make by adjusting all the "knobs" (parameters) available in the underlying data representation. Simply exposing these knobs, however, is not realistic or even desirable. Most users would be unable to express mathematically all the tradeoffs they make in practice, since this would require making quantitative comparisons between, say, making a U-turn and paying a toll. Also, as there are dozens of parameters, most users would be unwilling to tune them, even assuming that they could handle this vast combinatorial space.

In this paper, we propose an unsupervised learning framework that can *automatically* infer the preferences of a user based on an analysis of how he or she actually drives. In more formal terms, the problem we address is as follows: given a parameterized function that assigns costs to arcs and turns of a road network, together with a set of traces representing specific driven routes (for example, from an individual user), find a set of parameter values that best fits the traces. We define "best" so as to ensure that as many traces as possible correspond to shortest paths with respect to the arc and turn costs. Figure 1 gives an example. We note that our framework could be easily extended to take into account other factors (such as traffic situation or weather) if we were to have access to such data.

The main contributions of this work are as follows. First, we customize a version of stochastic coordinate descent [4, 8, 27] in order to learn the parameters of our cost function. Second, we engineer the framework to make it scalable to the large graphs required to represent continental road networks (tens of millions of vertices) and to thousands of traces. Third, we present an extensive experimental evaluation of our techniques using real-world map data from Bing Maps, as well as a large number of real traces from 85 volunteer drivers. We show that we can (a) successfully recover a cost function in a few minutes, and (b) generate *personalized* cost functions (improving the route quality in almost all cases) for individual users in about 30 seconds on a server.

2. PRELIMINARIES

As is standard in this domain, we model a road network as a directed graph $G = (V, A)$, where each vertex $v \in V$ represents an intersection and each arc $a \in A$ corresponds to a (directed) road segment. In addition, each intersection has associated turns between its incoming and outgoing arcs. A *cost function* F maps each arc a or turn t into a nonnegative *cost* reflecting the effort to traverse it. A *path* in the graph is a sequence of arcs of the form (v_0, v_1), (v_1, v_2), (v_2, v_3), $\ldots$, (v_{k-1}, v_k). The *cost* of a path is the sum of the costs of its arcs and of the turns between them. The *shortest path* problem takes as input the graph G and two arcs a_s and a_t, and returns the shortest (minimum-cost) path that starts at a_s and ends at a_t in G.

The cost function F maps the static properties of any arc (road category, number of lanes, speed, etc.) or turn type (left, right, U-turn, etc.) into the *cost* of traversing it. Each vendor uses a particular proprietary F to provide routes to its users. Regardless of the particular form and shape of F, we will make the reasonable assumption that it is defined by a vector of parameters β. To make this dependence clear, we refer to a cost function F instantiated by a vector β as F_β.

As an example, assume each road segment has a physical length (in meters) plus indicator (0/1) variables for three road categories (local, arterial, freeway). Moreover, there exist four types of turns (left, right, straight, U-turn). We can define the function F to use average speeds for each road category and fixed values (times) for each turn type. A vector β can then specify the numerical values of the corresponding coefficients in F. As an example of β, we can set 20, 40, and 60 kph for the three road categories; for the turns, we could use 0 s for straight, 5 s for right, 10 s for left, and 15 s for U-turns. This vector of parameters would be a reasonable choice for an average driver. For ambulances, we could have a different β' with higher average speeds (say, 30, 50, and 80 kph) and fixed small costs for all turns (3 s). For trucks, we could have a third vector β'' with reduced average speeds and higher costs for left and U-turns. Note that these examples are simplified and not necessarily realistic; a real-world data set has many more static properties (and coefficients to set).

In this work, we use the cost function F actually used by Bing Maps. It has a few dozen parameters β_i and is nonlinear; its precise form is proprietary. It correlates well with driving times, using additional penalties to avoid, in varying degree, undesirable elements (such as unpaved roads, U-turns, and tolls). Only a subset of all parameters is "active" for any individual arc or turn. For example, the cost of a freeway arc does not depend on the parameter associated with the category "unpaved roads". We note that we use the real-world cost function used by Bing Maps as of 2014, with no simplifications. This function has one order of magnitude more parameters compared to previous work [22, 23]. The flipside of using such a function is that some specifics are proprietary. However, our algorithm works for other reasonable cost functions.

A *trace* is a sequence of points, given by latitude and longitude. Road networks are embedded into the same geometric space E, and each vertex in G has an associated point in E. An arc (v, w) is represented as a polyline (sequence of straight segments) between the points corresponding to v and w. Besides v and w themselves, the polyline may have additional interpolation points to model curved roads. The closest arc to a given point is the arc whose polyline has minimum distance (in the geometric space) to the point. The *point-to-point shortest path* problem takes as input the graph G, the embedding E, and two points p_s and p_t. It first finds the closest arcs a_s and a_t to p_s and p_t, respectively, and then solves the (arc-to-arc) shortest path problem.

2.1 CRP

Our framework relies on the *customizable route planning* (CRP) technique [15] for computing shortest paths. Although CRP can be used as a black box within our algorithm, we outline how it works for completeness. (The reader can find more details in the original article [15].) The distinguishing feature of CRP is that it splits the usual preprocessing phase in two subphases.

The first subphase creates multiple (typically no more than five) levels of nested partitions. Within each level, CRP partitions the vertices into *cells* of bounded size (number of vertices) while minimizing the number of edges between them. Crucially, this phase does not depend on the cost function, and therefore only needs to be run once (in a few minutes) on continental road networks.

The second preprocessing subphase (called *customization*) then takes as input the partition (from the first phase) as well as a cost function (F_β in our case), and builds an *overlay*. For each cell in the partition, it computes *shortcuts* representing the shortest paths between its boundary vertices. Although the customization subphase must be rerun for every new vector β we test, it is quite fast, particularly when run on GPUs [16].

A query from a source a_s to a target a_t runs a bidirectional version of Dijkstra's algorithm, but using shortcuts to skip cells that contain neither a_s nor a_t. On continental road networks, even long-range queries visit only a couple of thousand vertices on average.

3. BASIC FRAMEWORK

In a nutshell, our learning task is as follows. We are given as input a road network G, a cost function F with an initial vector β^0 of parameters, and a set of traces T. Our goal is to find a vector β^* such that the paths produced by a navigation engine using F_{β^*} "match" T as well as possible. The real-life cost function we have access to has over 50 parameters; this is the number of components in the vector β^* we must learn.

Our basic approach to learn these parameters is to use a version of stochastic coordinate descent [4, 8, 27]. We maintain a current vector β, which we steadily try to improve as the algorithm progresses. The main building block of our approach is a *local search* procedure, which systematically explores the parameter space to find improvement steps, and keeps moving towards improving solutions until it reaches a local optimum. To escape local optima, we use two strategies: *perturbation* and *specialized sampling.* The first allows the algorithm to explore "plateaus" in the search space, i.e., different parameter values that do not lead to strictly better solutions. Those are quite common in our application, as multiple vectors β can lead to the same shortest path between a given source and a given destination. (Intuitively, although the β_i parameters are continuous, the actual paths they induce are discrete.) To escape deeper valleys in the search space, we rely on specialized sampling to focus the attention of the algorithm on traces that are not well matched by the current β value.

Note that we have kept the notion of "matching" measured traces to computed paths deliberately vague in our problem formulation, since our fitting procedure is in general agnostic to this choice. All it assumes is the availability of a *quality oracle* $Q(G, T, F_\beta)$ capable of evaluating a set of parameters β on a set of traces T. Our convention is that higher scores mean greater quality; our objective is thus to maximize the sum of the scores of all traces. Note that if we want to bias the cost function towards certain traces, we could assign a weight to each trace and maximize the weighted sum of the scores of all traces.

4. LEARNING PROCEDURE

We next describe in detail the elements of our learning procedure: coordinate descent and the two procedures to escape local minima. Then we present an overview of how these three parts interact.

4.1 Local Search

Following a basic stochastic coordinate descent approach, local search proceeds in rounds and terminates when several consecutive rounds fail to improve the quality score. Each round first selects a random permutation of the parameters (components) in β, then searches for a better value for each one in the order given by the permutation. After processing a parameter, we update β (if needed) and proceed to the next one.

Coordinate descent has been developed in the context of nonlinear optimization [4, 8, 27], where a change in a given coordinate is computed using line search. In our case, the objective function is discrete (and thus non-differentiable), so we use sampling instead of line search to find a coordinate change that improves the objective function. Our experiments show that the sampling approach works well for our problem.

Formally, to find a better value for β_i, we take a sample set of alternative values for β_i, biased towards the current value. For each value in the sample, we compute the quality score of the solution obtained from β by changing β_i to the sample value. If the *maximum* score value is greater than the current one, we change β_i to a sample value that gives the maximum score. Otherwise β_i remains unchanged.

4.2 Perturbation

Since many parameter values may induce the same shortest path, there are many plateaus in the multi-dimensional surface defined by F_β (we verified this experimentally). If β is on a plateau, no change in a single parameter value leads to a score improvement. However, a simultaneous change in several parameters may yield a better score. The *perturbation* phase attempts to escape a plateau. It is similar to the local search phase, but allows "sideways" moves: changes in a parameter value with no change in quality score.

More precisely, the perturbation phase processes the parameters in random order. For each parameter β_i, it samples a few values (biased towards the initial value), just as in the local search phase. Let v_a and v_b be the minimum and maximum sampled values that achieve the *same* quality score as the initial β_i value. We sample uniformly at random from $[v_a, v_b]$ until we find a value that also matches this score (on rare occasions, values in $[v_a, v_b]$ can lead to a worse score), and set β_i to it. Occasionally, a value tested during this procedure will lead to an improved quality score. In such cases, we set β_i to that value and end the perturbation phase.

We experimented with combining the local search and perturbation phases into one, by allowing the local search phase to move sideways when an improvement is not found for a parameter. Separating these two phases led to slightly better results.

4.3 Specialized Sampling

The goal of this phase is to escape deep valleys in the search space in order to explore alternative local minima. To perform a "drastic" move and explore a different section of the search space, we identify the traces that are not fully matched (i.e., that are not assigned the highest possible score by the quality oracle). To emphasize these traces when evaluating the overall score, we increase their weights and maximize the weighted sum of the scores of all traces. We found that this extreme perturbation is most successful when we increase the weight by a factor of 10. For simplicity, we refer to this specialized sampling as *boosting* in the remainder of this paper, although it is technically not the same as the usual definition [30].

4.4 The Algorithm in Full

Our final algorithm has three parameters (ML, MS, MP). It repeatedly performs rounds of local search (stochastic coordinated descent) until it sees ML consecutive rounds in which the quality score does not improve. The algorithm then uses up to MP rounds of perturbation to try to escape the local optimum; if it succeeds, it restarts from the current solution. Otherwise, the algorithm uses specialized sampling, i.e., it penalizes unmatched traces and restarts as soon as an improvement is found for the weighted trace set. At most MS rounds of specialized sampling are allowed. Eventually, the algorithm returns the solution with the highest score (under the non-boosted sets evaluated) found during the entire process. By default, we set MS = 10, ML = 3, and MP = 10. We also consider a faster *economical* mode of our algorithm, which uses MS = 3, ML = 2, and MP = 3.

5. QUALITY ORACLE

A basic operation of our algorithm (executed in various places) is to evaluate the quality score $Q(G, T, F_\beta)$ of the cost function F with a set of parameter values β with respect to a set of traces T. We first define the precise score function we use in our experiments, then explain how it can be computed efficiently.

5.1 Quality Score

A fundamental operation is to assign a quality score to each individual trace $t \in T$, given a vector β of parameters. We do so by determining how many points of the trace can be mapped to the corresponding shortest path (according to the cost function F_β).

More precisely, to evaluate the trace t, we first run a point-to-point shortest-path query from its first to its last point. Let P be the corresponding shortest path. We say that a point p in t is *matched* if its (Euclidean) distance to P is within a certain threshold x, which depends on the quality of the map data and the device used to obtain traces. For our data, we found that setting x to 10 meters works well. The quality score for track t is the fraction of its points that are matched.

The overall quality score $Q(G, T, F_\beta)$ is the average score over all $t \in T$. (If specialized sampling is used, the average is weighted appropriately.) A score of 1.0 (or 100%) means that all traces in T can be perfectly matched. For convenience, we sometimes refer to the *matching error*, defined as $(1 - Q(G, T, F_\beta))$, instead of to the score itself.

5.2 Efficient Shortest-Path Computation

To compute the quality score for each trace t, we must first compute the point-to-point shortest path (according to the cost function induced by the current β) between the first and last points of t. One could do so by simply running Dijkstra's algorithm [18]. Starting from the source, it visits vertices in increasing order of distance until the target is processed. Although this is reasonably fast if traces are short, it is not robust enough for our purposes: a single long-range shortest path computation on a continental road network can take seconds, since it visits almost the entire graph [31].

Many recent techniques can bring worst-case (exact) query times down to a microsecond or less [1, 7] after a few minutes of preprocessing; see Bast et al. [6] for a survey of such methods. Unfortunately, however, the preprocessing routines of most methods must know the cost function in advance. Since our application changes the cost function frequently, it cannot benefit from these accelerations. A recent technique can set the cost function at query time [22], but its queries are too slow in our setting.

The best fit for our application is the *customizable route planning* (CRP) approach [14, 17, 15] outlined in Section 2.1. It still uses preprocessing, but it can incorporate a new cost function for a full continental road network (with tens of millions of vertices) in seconds on CPUs [15] or fractions of a second on GPUs [16]. The time to incorporate a new cost function depends linearly on graph size, which makes it much faster on smaller networks, such as metropolitan areas. Arbitrary queries take a couple of milliseconds (even on continental scale) on a commodity CPU, which is fast enough for our needs. We use CRP as a black box.

5.3 Efficient Matching

We now consider the problem of deciding whether each point p from an input trace t can be matched to a given path P in the graph, typically resulting from a shortest-path computation. For this purpose, we interpret the path P as a polyline, i.e., a sequence of adjacent straight line segments.

Our goal is to determine, for each point p in the trace, whether the distance from p to the polyline P is at most x. This can be trivially accomplished by computing the distance from each p to each segment in P, but this would be too slow. We therefore propose some optimizations.

First, we use simpler geometry. Instead of computing geographical distances (along the surface of the Earth), we assume an embedding on the plane, with the longitude of a point indicating its x-coordinate and the latitude its y-coordinate. This avoids expensive trigonometric functions, and our preliminary tests indicate that the number of mismatches is negligible, since we deal with very small distances. To avoid computing square roots, we evaluate the square of distances instead and set our threshold accordingly.

Even with this acceleration, explicitly computing the distances from each point p in the trace to each segment of the polyline P would be too expensive, considering that we would have to do this for every point in each trace. To reduce the number of segments of P, we run the Douglas-Peucker [20] polyline simplification algorithm, while still retaining accuracy within one meter. To further reduce the time per point to logarithmic (in practice), we use *bounding box indexing.* For a polyline P with $|P|$ segments, we use $\log(|P|)$ levels of axis-aligned bounding boxes. The highest level has one bounding box containing the full polyline; the second highest level has two boxes, one for each half of P; and so on. This bounding box index can be built by one sweep over all segments of P. When querying a point, we use this index top-down, quickly discarding large chunks of the polyline. If the distance from the point to a bounding box is larger than x, all segments in the box can be discarded. If paths do not fold too much onto themselves (as is usually the case for real shortest paths), the effect is very similar to a binary search.

Note that the index is built once for each path P, and reused for each point in the trace. With all accelerations, evaluating all points of a trace is roughly as expensive as computing the shortest path with CRP.

5.4 Oracle Approximation

The quality oracle described so far is fast enough to handle a large number of traces on continental-sized road networks, but we can improve efficiency further.

The idea is to keep for each trace t a *candidate path pool* C_t, which is a collection of paths between the endpoints of t. Whenever we evaluate a new β within our two subroutines (local search or perturbation), we evaluate it on the pool, as follows. For each trace t, we check which path P in C_t has the lowest cost according to β; we take the score of this path as the score for t. Then, after running the full subroutine (which evaluates many different β), if we find a β' that improves the overall score with respect to the candidate pools, we evaluate β' with the "exact" oracle we described before (which invokes shortest-path computations on the graph and matching points to the resulting paths). We only make the move if β' also improves the score according to the exact oracle.

The pool for each trace t is initialized with two paths. The first is the shortest path according to β^0, the initial parameters. For the second, we split t into five equal-sized subtraces, compute the shortest paths between their start and end points according to β^0, and concatenate them. Whenever the exact oracle produces a previously unseen path for a trace t, we add it to the pool C_t. In our experiments, a candidate pool typically ends up with about ten paths.

Since it does not require shortest-path computations, the approximate oracle is much faster than the exact one. For further acceleration, we store the quality score of each candidate path in C_t added to the pool, thus avoiding subsequent redundant matching computations.

Finally, we keep a summary of each candidate path (how many meters on freeways, how many left turns, etc.), which allows us to determine its cost for a new β without looking at the entire path. Note that this requires two assumptions on the cost function: (1) the contribution of each edge/turn type is independent (i.e., they can be added up); and (2) for a particular edge type, its contribution depends linearly on the length (which means storing the total length is enough). These assumptions hold for the Bing cost function (and maybe for most reasonable functions), but we stress that some cost functions cannot be expressed with this summary.

Overall, using the approximate oracle within our subroutines accelerates our framework by three orders of magnitude, with a negligible effect on quality. We therefore use it by default in our experiments.

5.5 Alternative Oracles

We have focused on one particular score function, based on the fraction of trace points that are close enough to the shortest paths between its endpoints. We note, however, that our learning procedure only needs "oracle access" to the score function: it only needs the quality score itself, and assumes nothing about how it is computed. Therefore, we could easily consider other functions as well.

In particular, in our preliminary experiments we considered a continuous quality measure that depends on how close each point is to the shortest path. Even if a point is not perfectly matched to the path, being nearby should intuitively be better than being far away. This measure, however, turned out to be less effective than our default quality score function, in which a point is either perfectly matched or not. The main reason is the discrete nature of shortest paths: by completely disregarding points that are not matched (rather than trying to break ties among them), the learning procedure is more focused.

Another score function we considered is as follows. Instead of trying to match the trace to a single path (the shortest), we could consider a small number of alternative paths as well. If a system can offer (say) three alternatives [5, 2, 24] in response to a query (as many commercial systems do), the user will be happy as long as at least one of them reflects his preferences. It is trivial to extend our approach to handle alternatives; we simply consider a point from the trace as "matched" if it is close enough to at least one line segment from any of the alternative paths. This would capture users that take different paths depending on traffic conditions, for example.

6. RELATED WORK

To the best of our knowledge, we are the first to use GPS traces to learn a cost function of multiple variables that can be used by a navigation engine. Up to now, GPS traces have mainly been used to reconstruct the path a user has taken [10, 28], or to improve (or even construct) maps [29, 12, 33, 32, 9]. Most closely related to our work are previous projects for personalized routing. The Coolest router [13] allows users to manually set the importance of distance, time, nearby points of interest, and path simplicity. Duckham and Kulik [21] developed a technique to compute the simplest path between two points as an alternative to the shortest or fastest route, which they argue could be preferable. Agrawala and Stolte [3] showed how to visualize so-called "wedding maps," which are spatially distorted route maps that emphasize details at turns and landmarks. After examining routes from GPS logs, Letchner et al. [26] found that drivers took the fastest route, as given by a commercial routing engine, only 35% of the time. They went on to create a router that better matched drivers' chosen routes by taking into account actual measured road speeds and by biasing the router to prefer a driver's more familiar roads. Similarly, Chang et al. [11] created a personalized router by noting which roads were most familiar to the driver, producing routes that use these familiar roads. Compared to previous work, ours is more general. We infer preferences even in areas not previously visited by a user.

7. EXPERIMENTS

We implemented all algorithms in C++ and CUDA, and compiled them with Visual C++ 2012 and CUDA 5.5. We performed all timed tests on a workstation running Windows 8.1. It has an Intel Core-i7 4770 CPU (4 cores, 8 threads, 3.4 GHz, 4x64 KB L1, 4x256 KB L2, and 8 MB L3 cache) and 32 GiB of 1600-DDR3 RAM. It also has four EVGA NVIDIA GTX 780 Ti OC GPUs, each with 15 multiprocessors, 2880 CUDA cores, 1.2 GHz core clock rate, and 3 GiB of 7 GHz memory. The GPUs are solely used for accelerating the customization phase of CRP [16]. We use, unless otherwise mentioned, the default parameters for our learning procedure (MS = 10, ML = 3, and MP = 10; see Section 4.4).

We use the road network of North America as input, with about 32 million vertices. We also use a smaller subgraph of this network representing the state of Washington, with 810 thousand vertices and 955 thousand road segments. CRP needs 7 ms to incorporate a new cost function in Washington and 70 ms in North America.

The input road network data of Bing Maps (based on Navteq data) has several dozen parameters, which can be grouped into three types: speed-related (about three dozen), delay-related (about a dozen), and turn-related (about a dozen). We cannot reveal the exact parameters, but its default cost function correlates well with travel times in light traffic.

7.1 Controlled Experiments

Our first set of experiments is in a controlled environment. The idea is to generate traces from shortest paths computed using a cost function instantiated with known parameters, then hide these parameters from our algorithm. This setup is meant to evaluate the extent to which we can reconstruct the original parameters of the cost function. The advantage of this approach is that we know that a perfect solution (ground truth) exists, allowing us to calibrate our algorithm.

7.1.1 Methodology

We generate a set of k traces by running k arc-to-arc shortest-path queries (with the source and target arc chosen uniformly at random), using the default cost function of Bing Maps. We produce a trace from each resulting route by sampling it at regular intervals (measured in seconds). Our controlled experiments use 10 000 traces to evaluate the quality of a parameter set β. These *evaluation* traces are never seen by the learning procedure, which uses a separate set of *training* (or *learning*) traces.

The speed parameters in β give the average speed for each of the possible road types, such as arterial, highway, and ramp. Our algorithm has two modes, depending on whether or not these average speeds are known in advance. In the *known-speeds* mode, we initialize our framework with a cost function where all delay parameters are set to 1.0 (neutral) and turn costs are set to 0, but provide the correct average speeds and do not allow the learning procedure to change them. This leaves us with about two dozen parameters to optimize. Since average speeds can be obtained from other sources (such as speed limits or GPS tracks with timing information), the known-speeds setup is very relevant in practice.

Under some conditions, however, it might be useful to learn the speed-related parameters as well. In the *unknown-speeds* mode, we initialize each speed parameter with the same arbitrary value (10 kph), and allow them to be adjusted by the learning function. We still provide the algorithm with a hint regarding the *order* among the roads (from slowest to fastest). So we create a parameter β_0 corresponding to the average speed on the slowest roads but, for any other speed parameter i, β_i represents the (nonnegative) *speed difference* between the i-th and $(i-1)$-th slowest road type. So, except for the slowest roads, the algorithm must learn the "deltas" rather than the absolute value.

For both modes, we provide the algorithm with upper and lower bounds for each parameter. Turn-related parameters cannot exceed more than 10 minutes, delay factors are at least 1.0 and at most 5.0, and the minimum and maximum traversal speeds are 5 kph and 150 kph, respectively.

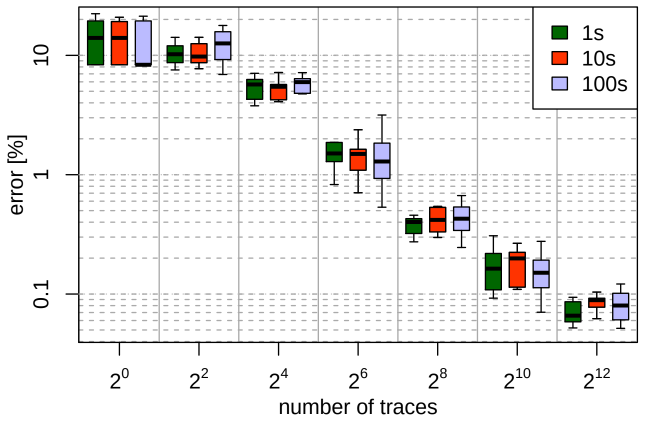

Figure 2: Quality of the learned cost function, depending on the sample size (number of traces). Speeds are known, but the sampling rate for each trace varies.

We note that a real system would likely start from a better initial set of parameters, such as the default one used by the vendor, but our goal here is to stress-test the algorithm.

7.1.2 Learning Requirements

We first evaluate the tradeoff between the amount of learning data we have and the quality of the solution we find. We vary both the number of training traces and the sampling rates for each one.

Figure 2 reports, over 10 runs, the median as well as the 75% (box) and 95% (whisker) confidence intervals for the matching error $1-Q$ of the parameters we learn as a function of the number of learning traces (1, 4, 64, 256, 1024, 4096) with different sampling rates (1, 10, 100 seconds) used to generate synthetic traces. This experiment uses the known-speeds mode.

The figure shows that, with as few as 64 traces, we can find a cost function that is almost 99% similar to the ground truth (as given by the 10 000 evaluation traces). More training traces help. With 4096 traces, we bring the matching error well below 0.1% compared to the 10 000 evaluation traces. In fact, in almost all cases we can match 100% of the training traces (as opposed to the evaluation traces).

The experiment also suggests that a sampling rate of 100 s is sufficient to achieve good results. If drivers log their location only every 100 seconds, we can reconstruct the underlying cost function almost as well as when sampling every second. Note that we vary the sampling rate only for the training traces; the evaluation traces have a sampling rate of 1 second in all cases. Note that our algorithm does not use the timestamps of the GPS points at all. For the remainder of this paper, we use a sampling rate of 10 seconds for training traces by default. The real-world traces we evaluate later have a similar sampling rate.

7.1.3 Learning Speeds

Next, we evaluate how much it helps to know average speeds. Figure 3 compares the quality of the cost function we learn for our two modes (known- and unknown-speeds). Although having the correct speeds as input leads to a better learned cost function, the overall solution quality is still

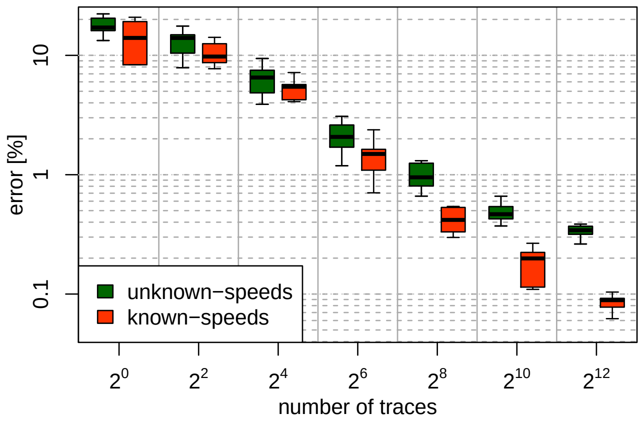

Figure 3: Quality of the learned cost function, depending on the sample size (number of traces). The sample rate is fixed to 10 s, but speeds either are given (and fixed) or must be learned (initialized to 10 kph).

quite high even if we have to learn the speeds. For 1024 traces, we obtain scores of about 99.5%.

7.1.4 Other Inputs

Table 1 reports the quality of the parameters we compute if we use different cost functions to generate the training traces. The parameterizations we use correspond to travel times (all delay factors neutral, no turn costs), trucks (lower top speed, turns more expensive), and ambulances (higher average speed on all arcs and lower turn costs). In this experiment, we also test the faster economical (MS = 3, ML = 2, and MP = 3; see Section 4.4) mode of our algorithm. Finally, we test our default cost function for the full North America data set.

We observe that in all cases we can reconstruct the underlying hidden cost function quite accurately. Moreover, the economical variant is two to four times faster, with little loss in quality. In fact, it learns fully-fledged cost functions from a large number of traces on continental-sized networks in a few minutes.

We note that, even if we did not use the GPUs in our system at all, the learning system would be only slightly slower. Over a complete run (with GPUs), about 75% of the time is spent in evaluating the candidate pool, 12% in computing shortest paths (using CRP queries), 10% in evaluating how

			KNOWN-SPEEDS			UNKNOWN-SPEEDS		
			quality		time	quality		time
input	function	eco.	start	end	[s]	start	end	[s]
WA	default	×	0.9166	0.9983	519	0.4474	0.9948	577
	default	✓	0.9166	0.9975	205	0.4474	0.9916	175
	truck	✓	0.7690	0.9904	186	0.4212	0.9863	224
	time	✓	1.0000	1.0000	0	0.4736	0.9881	111
	ambul.	✓	0.9438	0.9970	104	0.4565	0.9911	160
NA	default	×	0.8370	0.9977	1329	0.3074	0.9915	2476
	default	✓	0.8370	0.9960	342	0.3074	0.9883	662

Table 1: Results using 1024 traces as learning sample, 10,000 traces to evaluate, frequency of the learning sample set to 10 seconds. Note that we can recover any cost function in a few minutes.

many GPS points match them, and only about 3% in running the CRP customization phase (updating the CRP data structures on the GPU). Even if we ran customization on the CPU, it still would not be the bottleneck of our framework. However, the system would be considerably slower if we used Dijkstra instead of CRP. With Dijkstra, random queries on Washington would be 350 times slower (174 ms per arc-to-arc query for Dijkstra, as opposed to 0.52 ms for CRP). For North America, CRP is 3100 times faster (13750 ms for Dijkstra, 4.4 ms for CRP).

7.2 User-Specific Traces

We now evaluate how well we can learn the driving habits of real drivers. For this purpose, we evaluated GPS traces from 85 volunteers from the Washington state area. Each user was observed for a median of 57 days (ranging from 15 to 1 748 days), with actual GPS data for a median of 48 days (ranging from 13 to 1 252 days). To avoid under- or over-segmentation, we need to split each driver's GPS log into discrete trips. As suggested by previous work [19], we end a trip if a driver stayed within 17.2 meters for at least 90 seconds. Krumm and Rouhana [25] show that this gives a recall rate of 99% in detecting a non-moving GPS logger. According to data from the 2009 U.S. National Household Travel Survey (http://nhts.ornl.gov/), 99% of driving trips are either at least 483 meters long or 3 minutes long. Thus, we eliminate any trips shorter than these amounts. After processing the trips from our 85 drivers, the median number of trips per driver is 188 (ranging from 37 to 8 608), and the total number trips for all drivers is 34 708.

For these 85 drivers, we evaluate the route quality with respect to the standard cost function (provided by Bing Maps) and compare it with the cost function our algorithm computes. We used the Washington instance as the underlying road network. We initialize our framework with the parameters of Bing's standard cost function.

Figure 4 shows the quality score of the parameters for each driver before (x-axis) and after (y-axis) optimization. We report the median as well as the 75% (box) and 95% (whisker) confidence intervals of a 10-fold cross-validation. We consider our unknown-speeds scenario. For 95% of the users, the median is above the main diagonal, indicating that our method indeed leads to routes that fit the driving habits of each user better. On average, we increase the route quality by an additive factor of 0.054, which is an improvement of 7.5% over Bing's cost function. In the best case, the average quality score increases from 0.479 to 0.713, an improvement of 49%.

An interesting observation is that we can identify different driver types from analyzing the individual cost function we obtain. Some drivers prefer freeways, some minimize their travel distance, some do not mind making left turns, and others avoid turns as much as possible.

Recall that in this experiment we initialize our framework with a reasonable cost function. Combined with the fact that real-world traces normally are shorter than our synthetic ones, the algorithm here is much faster than reported in Table 1. The running time stays below 30 seconds for almost all users. Moreover, in a closed environment like a car, new traces could be processed incrementally (each new trace can start from the set of parameters learned from previous traces), reducing the computational effort even further.

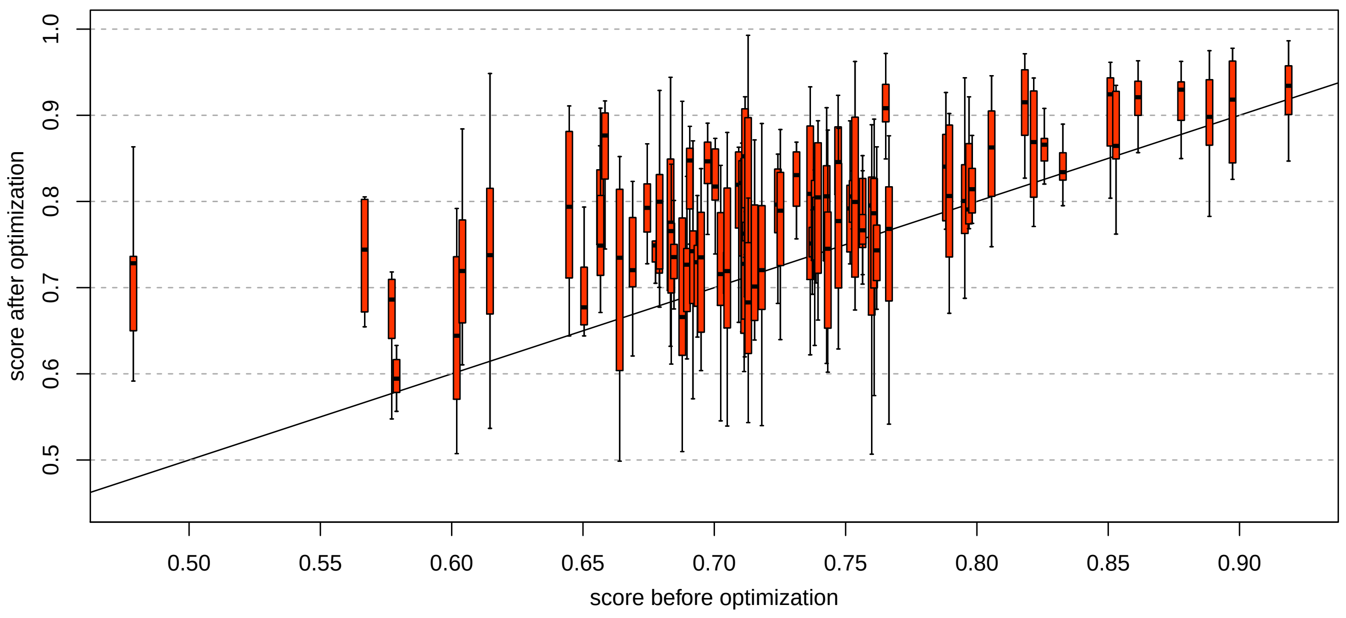

Figure 4: Route quality before and after our optimization, using as input the parameters provided by Bing Maps. Note that 95% of the medians are above the diagonal, indicating a significant improvement.

8. CONCLUSION

We have introduced a framework for personalizing driving directions by automatically analyzing the GPS traces from the driver. To our knowledge, this is the first approach of this kind that is validated by controlled experiments on a continental road network, as well as by using real-life personal traces from a relatively large population.

Our results indicate that, given the structure of the cost function used by a routing engine, our algorithm can recover the parameters of the function with close to 100% accuracy. Moreover, GPS trace sampling can be as infrequent as once every 100 seconds. For personalization, we obtain an improvement of 7.5% over Bing's cost function on average, with some cases reaching 49%. This provides strong evidence of the benefits of our approach. Preliminary tests indicate that we can further increase the route quality if we store multiple cost functions per user, depending on the time of the day.

Moreover, the approach is efficient enough to be performed in a closed environment, such as a car or a standard PC, ensuring that highly sensitive personal information does not leave the control of the user.

9. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 18(1):1–17, 2013.

[3] M. Agrawala and C. Stolte. Rendering effective route maps: improving usability through generalization. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pages 241–249. ACM, 2001.

[4] A. Auslender. *Optimisation Métodes Numériques*. Masson, Paris, 1976.

[5] R. Bader, J. Dees, R. Geisberger, and P. Sanders. Alternative Route Graphs in Road Networks. In A. Marchetti-Spaccamela and M. Segal, editors, *Proceedings of the 1st International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS'11)*, volume 6595 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2011.

[6] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route Planning in Transportation Networks. *CoRR*, abs/1504.05140, 2015.

[7] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

[8] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, Massachusetts, second edition, 1999.

[9] J. Biagioni and J. Eriksson. Map inference in the face of noise and disparity. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pages 79–88. ACM, 2012.

[10] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 853–864. VLDB Endowment, 2005.

[11] K.-P. Chang, L.-Y. Wei, M.-Y. Yeh, and W.-C. Peng. Discovering personalized routes from trajectories. In *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Location-Based Social Networks*, pages 33–40. ACM, 2011.

[12] N. Cohn. Real-time traffic information and navigation.

Transportation Research Record: Journal of the Transportation Research Board, 2129(1):129–135, 2009.

[13] E. R. da Silva, C. de Baptista, L. Menezes, and A. Paiva. Personalized path finding in road networks. In *Proceedings of the Fourth International Conference on Networked Computing and Advanced Information Management*, volume 2, pages 586–591. IEEE, 2008.

[14] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

[15] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning in Road Networks. *Transportation Science*, 2015. online preprint.

[16] D. Delling, M. Kobitzsch, and R. F. Werneck. Customizing driving directions with GPUs. In *Proceedings of the 20th International Conference on Parallel Processing (Euro-Par 2014)*, volume 8632 of *Lecture Notes in Computer Science*, pages 728–739. Springer, 2014.

[17] D. Delling and R. F. Werneck. Faster customization of road networks. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 2013.

[18] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.

[19] M. Dimond, G. Smith, and J. Goulding. Improving route prediction through user journey detection. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 466–469. ACM, 2013.

[20] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10:112–122, 1973.

[21] M. Duckham and L. Kulik. "simplest" paths: Automated route selection for navigation. In *Spatial Information Theory. Foundations of Geographic Information Science*, pages 169–185. Springer, 2003.

[22] S. Funke, A. Nusser, and S. Storandt. On k-path covers and their applications. In *Proceedings of the 40th International Conference on Very Large Databases (VLDB 2014)*, pages 893–902, 2014.

[23] R. Geisberger, M. Rice, P. Sanders, and V. Tsotras. Route Planning with Flexible Edge Restrictions. *ACM Journal of Experimental Algorithmics*, 17(1):1–20, 2012.

[24] M. Kobitzsch. HiDAR: An alternative approach to alternative routes. In *Proceedings of the 21st Annual European Symposium on Algorithms (ESA'13)*, volume 8125 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2013.

[25] J. Krumm and D. Rouhana. Placer: semantic place labels from diary data. In *Proceedings of the 2013 ACM international Joint Conference on Pervasive and Ubiquitous Computing*, pages 163–172. ACM, 2013.

[26] J. Letchner, J. Krumm, and E. Horvitz. Trip router with individualized preferences (trip): Incorporating personalization into route planning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1795. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[27] R. Maxunder, J. H. Friedman, and T. Hastie. SparseNet: Coordinate descent with nonconvex penalties. *Journal of the American Statistical Association*, 106(495), 2011.

[28] P. Newson and J. Krumm. Hidden markov map matching through noise and sparseness. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 336–343. ACM, 2009.

[29] S. Rogers, P. Langley, and C. Wilson. Mining GPS data to augment road models. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 104–113. ACM, 1999.

[30] R. E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[31] C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46:547–560, 2014.

[32] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 99–108. ACM, 2010.

[33] B. D. Ziebart, A. L. Maas, A. K. Dey, and J. A. Bagnell. Navigate like a cabbie: Probabilistic reasoning from observed context-aware behavior. In *Proceedings of the 10th International Conference on Ubiquitous Computing*, pages 322–331. ACM, 2008.