Customizable Point-of-Interest Queries in Road Networks

Daniel Delling Renato F. Werneck
Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA
{dadellin,renatow}@microsoft.com

September 2013

Technical Report
MSR-TR-2013-90

We present a unified framework for dealing with exact point-of-interest (POI) queries in dynamic continental road networks within interactive applications. We show that partition-based algorithms developed for point-to-point shortest path computations can be naturally extended to handle augmented queries such as finding the closest restaurant or the best post office to stop on the way home, always ranking POIs according to a user-defined cost function. Our solution allows different trade-offs between indexing effort (time and space) and query time. Our most flexible variant allows the road network to change frequently (to account for traffic information or personalized cost functions) and the set of POIs to be specified at query time. Even in this fully dynamic scenario, our solution is fast enough for interactive applications on continental road networks.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Modern map services and other spatial systems must support a wide range of applications. Besides computing optimal point-to-point routes, advanced queries like "find the closest Thai restaurant to my current location" or "what is the best place to stop for grocery shopping on my way home" need to be supported as well. All these *location services* depend on a location and a set of points-of-interest (POIs) with certain properties (such as open times, category, or personal preferences). Given a location, we want to rank the POIs to decide which ones to report first. If one wants to order them by the actual driving (or walking) time from a given location, all these problems could be solved with one or more calls to a standard graph search algorithm, such as Dijkstra's [18]. For continental road networks, however, this takes several seconds for long-range queries [10], too slow for interactive applications. For better performance, more sophisticated solutions are needed.

Ordering POIs based on spatial information has been extensively studied [8, 9, 26, 28, 31, 33, 36, 37]. Independently, the algorithm engineering community has developed methods to rank POIs according to driving times by augmenting hierarchical speedup techniques for point-to-point queries [13, 20], such as contraction hierarchies [21] or hub labels [2, 3]. These speedup techniques divide the work into two phases. An offline *preprocessing* phase computes auxiliary data which is used during the online *query* phase to find the same path as Dijkstra's algorithm, but much faster. Extensions of such methods to POI-related queries use the fact that we are looking for shortest paths only (between the source and a subset of the available POIs), and that any shortest path must pass through a restricted set of "important" nodes, or *hubs*. An indexing step can associate POI information with these hubs, allowing for quick queries. Although these methods work reasonably well, they have major drawbacks, including nontrivial preprocessing effort and excessive space requirements. Most importantly, hierarchical methods are not robust to changes in the cost function: even small changes (such as setting high U-turn costs) can have a significant adverse effect on their performance [6, 10, 11, 16, 21].

To overcome these limitations, we take a different approach. We propose a unified framework for dealing with POI-related queries based on *multilevel overlays* [25]. This classical acceleration technique for computing shortest paths in road networks has a long history [27, 35, 29], but it has often been dismissed as uncompetitive with state-of-the-art hierarchical methods [14, 38]. Recently, however, the *customizable route planning* (CRP) variant [11, 16], has been shown to be quite competitive for point-to-point computations. Although CRP queries are orders of magnitude slower than the fastest hierarchical methods (such as hub labels [2, 3] or transit node routing [5]), they still take only a couple of milliseconds, which is more than good enough for interactive applications.

As long as queries are fast enough, considerations such as flexibility, low space consumption, predictable performance, realistic modeling, and robustness end up being more important for map services than raw speed. CRP excels in this regard because it moves the metric-dependent portion of the preprocessing to a metric *customization* phase, which runs in roughly a second on a continental road network using a standard server. This directly enables real-time traffic information and personalized cost functions. Unlike hierarchical approaches, CRP supports realistic modeling (turn costs and restrictions) with very little overhead and is much more robust to the choice of the optimization function. Not coincidentally, the underlying routing engine for Bing Maps is based on CRP.

In this paper, we show how CRP (and multilevel overlays in general) can be used to deal with point-of-interest queries efficiently, with no loss in robustness. We also show that the approach is quite flexible: the same basic setup enables different trade-offs between indexing effort and query times. We use extensive experiments to compare our method with the state of the art, and observe that our lightweight indexing techniques have a fraction of the time and space requirements of hierarchical approaches. Perhaps suprisingly, query times are still competitive with those of hierarchical methods.

In particular, we show that multilevel overlays are extremely practical for finding the closest POI, even in an online version (with no index). In the offline case (when indexing is allowed), although they are slower than the best hierarchical methods, they are still fast enough and have much lower space overhead. Finally, we present the first practical approach for finding the best via node on dynamic continental road

networks with tens of thousands of candidate POIs. While CRP can handle changes to the cost function in a second or less, hierarchical methods may take hours. This makes our methods the only practical approach for query scenarios that are dynamic in nature.

The remainder of this paper is organized as follows. Section 2 formally defines the problems we want to solve and gives background on the CRP algorithm. Section 3 then shows how this approach can be extended to handle POI-based queries in *online* fashion, with no significant indexing effort. Section 4 shows how one can use indexing to accelerate queries further. Section 5 puts our results in context by describing relevant related work in the area, including some on which our method builds. Section 6 presents a detailed experimental evaluation of our algorithms and compares them to existing state-of-the-art alternatives. Section 7 concludes with final remarks.

This article is the full version of an extended abstract [15].

2 Preliminaries

This section presents background for our contributions. We explain our representation of road networks (Section 2.1), define the problems we must solve (Section 2.2), then summarize the CRP algorithm (in Section 2.3).

2.1 Road Networks and Shortest Paths

A road network is usually modeled as a directed graph $G = (V, A)$, where each vertex $v \in V$ represents an intersection of the road network and each arc $(v, w) \in A$ represents a road segment. A *metric* (or *cost function*) $\ell : A \to \mathcal{N}$ maps each arc to a positive *length* (or *cost*). For a more realistic model, we also take turn costs (and restrictions) into account. We think of each vertex v as having one *entry point* for each of its incoming arcs, and one *exit point* for each outgoing arc. We extend the concept of metric by also associating a *turn table* T_v to each vertex v; $T_v[i, j]$ specifies the cost of turning from the i-th incoming arc to the j-th outgoing arc.

In the *point-to-point shortest path problem*, we are given a source location s and a target location t, and our goal is to find the minimum-cost path from s to t (considering both arc and turn costs). We denote the length of this path by $dist(s, t)$. Mimicking real-world road networks, s and t are not necessarily vertices, but points located anywhere along the arcs. These can be thought of as individual addresses within particular streets.

Without turns, this problem can be solved by Dijkstra's algorithm [18], which scans vertices in increasing order of distance from s and stops when t is processed. It runs in essentially linear time in theory and in practice [23]. We can run Dijkstra's algorithm on the turn-aware graph by associating distance labels to entry points instead of vertices [11, 22]. An alternative approach (often used in practice) is to operate on an *expanded graph* G', where each vertex corresponds to an entry point in G, and each arc represents the concatenation of a turn and an arc in G. This allows standard (non-turn-aware) algorithms to be used, but roughly triples the graph size. In contrast, the turn-aware representation is almost as compact as the simplified one (with no turns at all). For technical reasons, however, hierarchical methods such as contraction hierarchies [21] tend to have much worse performance on this representation [11].

2.2 Points of Interest

We focus on applications that deal with POIs, such as tourist attractions or store locations. In computational terms, each POI p is simply a location along an arc of the road network. We say that such an arc *contains* a POI, or simply that it is a *POI arc*. We denote the set of candidate POIs (for a given query) as $\mathcal{P}$. All problems are also parameterized by an integer k (with $1 \leq k \leq |\mathcal{P}|$) indicating the maximum number of POIs that are to be reported in any query.

We consider two problems with these inputs. In the *k-closest POI problem*, we are given a source s and must compute the set of k POIs p_i from $\mathcal{P}$ that minimize $dist(s, p_i)$. In the *k-best via problem*, we

are given a source s and a target t, and our goal is to compute the set of k POIs p_i from $\mathcal{P}$ that minimize $dist(s, p_i) + dist(p_i, t)$.

To solve these problems, we consider algorithms that work in up to four phases, each potentially taking the outputs of previous phases as additional inputs. The first phase is *metric-independent preprocessing*, which takes as input only the graph topology. The second phase, *customization*, takes as input the metric (cost function) that defines the actual cost of each arc. The third phase, *selection* (or *indexing*), processes the actual set $\mathcal{P}$ of candidate POIs, given k. Finally, the *query* phase takes as input a source s (and potentially a target t) and determines which are the best POIs among those in $\mathcal{P}$. Note that some algorithms may conflate some of these phases into one.

Different applications may impose different constraints on each phase. For example, in the *static* variant of our problems, the metric (cost function) is known in advance, and does not change often. In the *dynamic* version, the cost function can change frequently (to account for real-time traffic information or individual user preferences, for example). This enables a richer user experience, but restricts the amount of time the customization phase can spend.

Orthogonally, our problem may be *offline* or *online*, depending on when the set $\mathcal{P}$ of POIs is known. In the *offline* version, the set $\mathcal{P}$ is known in advance. This is what is available in a typical *store locator* feature found in websites for large chains: the set of all stores is known in advance, and users just want the closest to their current location. In the *online* version, the set of candidate POIs is revealed only at query time, enabling a richer set of queries. This is more typical of a search engine: one might want to focus on business with a certain keyword in their names. The offline version is more restricted but, because it does not preclude lengthy selection phases, it tends to be easier to solve.

As Section 5 will show, most work in this area has focused on the static offline version. This article provides a wider range of solutions. In particular, although the algorithms we introduce focus on the last two phases (selection and query), they reuse the customization phase of CRP, which is the fastest among point-to-poing algorithms. This makes our approach ideal for dynamic applications.

2.3 Customizable Route Planning

We now explain the multilevel overlays approach [25] for computing point-to-point shortest paths on road networks. These methods compute multiple nested overlay graphs, which consist of a subset of the original vertices with additional arcs created to preserve the distances between them. More concretely, we focus on the *customizable route planning* (CRP) [11, 16] algorithm, the fastest variant of this method.

CRP works in three phases: metric-independent preprocessing, customization, and queries. Preprocessing takes as input an unweighted version of the input graph (the sets of vertices and arcs), partitions it, and creates the topology of the overlay graph. Customization takes as input a metric (the original arc lengths) and uses it to compute the lengths of the overlay arcs. Finally, the query phase takes a source s and a target t as input and produces the shortest s–t path, using the output of the first two phases for speed. Next, we discuss each of these three phases in more detail.

The *preprocessing* stage defines a multilevel overlay of the graph and builds auxiliary data structures. A *partition* of V is a family $\mathcal{C} = \{C_0, \ldots, C_k\}$ of sets $C_i \subseteq V$ such that each $v \in V$ is in exactly one *cell* C_i. A *multilevel partition* of V of L *levels* is a family of partitions $\{\mathcal{C}^1, \ldots, \mathcal{C}^L\}$, where l denotes the level of a partition $\mathcal{C}^l$. To simplify notation and avoid special cases, assume that $\mathcal{C}^0$ consists of singletons (each vertex is associated with its own cell) and $\mathcal{C}^{L+1} = \{V\}$ (all vertices form a single top-level cell). Let U^l be the size of the biggest cell on level l. CRP deals only with *nested* multilevel partitions: for each $l \leq L$ and each cell $C_i^l \in \mathcal{C}^l$, there exists a cell $C_j^{l+1} \in \mathcal{C}^{l+1}$ with $C_i^l \subseteq C_j^{l+1}$; we say C_i^l is a *subcell* of C_j^{l+1} and C_j^{l+1} is a *supercell* of C_i^l. A *boundary arc* on level l is an arc with endpoints in different level-l cells; its endpoints are *boundary vertices*. A boundary arc on level l is also a boundary arc on all levels below.

The preprocessing phase of CRP uses PUNCH [12], a graph-partitioning heuristic tailored to road networks, to create a multilevel partition. Given an unweighted graph and a bound U, PUNCH splits the graph into cells with at most U vertices while minimizing the number of arcs between cells. We find a

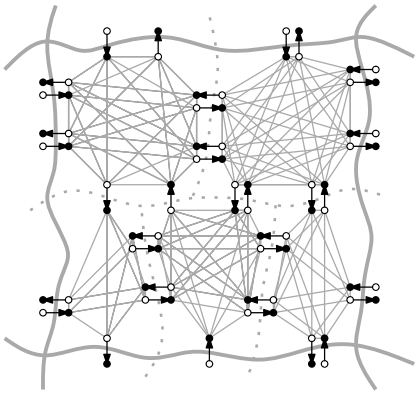


Figure 1: Overlays of five cells.

multilevel partition in top-down fashion: we first partition the full graph, then recursively apply PUNCH to each resulting cell.

Besides partitioning, the CRP preprocessing phase sets up the topology of the overlay graph. Consider a cell C on any level, as in Figure 1. Every incoming boundary arc (u, v) (i.e, with $u \notin C$ and $v \in C$) defines an *entry point* for C, and every outgoing arc (v, w) (with $v \in C$ and $w \notin C$) defines an *exit point* for C. The *overlay* for cell C is simply a complete bipartite graph with directed *shortcuts* (gray lines in the figure) between each entry point (filled circle) and each exit point (hollow circle) of C. The overlay of level l consists of the union of all cell overlays, together with all boundary arcs (black arrows) on this level.

During the *customization stage*, CRP computes the *lengths* of the shortcuts on the overlay. A shortcut (p, q) within a cell C represents the shortest path (restricted to C) between p and q. To process a cell, the customization phase must compute, for each cell, the distances between each entry point and each exit point.

The natural approach [11] to accomplish this is to process cells bottom-up, starting at level one. One can run Dijkstra's algorithm from each entry point p to find the lengths of all shortcuts starting at p. Processing level-one cells requires running Dijkstra on the original graph, taking turn costs into account. Higher-level cells can use the overlay graph for the level below, which is much smaller and has no explicit turns, since turn costs are incorporated into shortcuts.

The customization phase can be even faster [16] if one replaces Dijkstra's algorithm with the Bellman-Ford [7, 19] algorithm, which works better for small graphs (such as those representing individual cells). It has better locality and can be easily extended to compute more multiple shortest path trees (rooted at different entry nodes) at once using instruction-level parallelism [40] (SSE instructions). Additional speedups are possible using contraction [21] instead of graph searches during customization, although at the price of much higher preprocessing space [16].

A point-to-point CRP *query* runs bidirectional Dijkstra on the overlay graph, but only entering cells containing either the source s or the target t. We call this a *multilevel Dijkstra* (MLD) query.

3 Online Queries

We now show how to generalize the multilevel approach to deal with queries involving more than two endpoints (s and t). In particular, we consider generic *one-to-many* and *many-to-many* queries in Section 3.2, finding closest POIs in Section 3.3, and best via queries in Section 3.4. In all cases, the algorithms we suggest work in the online setting, in which the set of POIs is only revealed at query time.

3.1 Generalized Multilevel Dijkstra

Before getting to specific applications, we introduce a general framework to analyze elaborate queries. We consider queries in which distances to several points (not just s and t) are relevant; these are the POIs. To take them into account, we use a generalized version of the multilevel-Dijkstra (MLD) query used by CRP.

To establish general results that can be applied to various specific problems, we consider an abstract setting in which each nontrivial cell (on level 1 or higher) is labeled either *safe* or *unsafe*. (Level-0 cells, corresponding to individual vertices, are always safe.) We define a *safety assignment* to be *valid* if no unsafe cell has a safe supercell. (In other words, all ancestors of an unsafe cell must be unsafe as well.) We say that a cell is *active* if it is safe but has an unsafe supercell. We say that a vertex is active if it is a boundary vertex of an active cell, and that an arc is active if both of its endpoints are active. The *active graph* consists of all active vertices and arcs. Note that every vertex in the original graph belongs to exactly one active cell.

In this setting, a *generalized MLD search* consists of running Dijkstra's algorithm on the active graph. We define forward, backward, and bidirectional versions of this in the natural way. Note that a standard s–t point-to-point query is indeed a special case: it is a generalized MLD search whose assignment only marks as unsafe the nontrivial cells that contain s and t.

To help analyze more complex queries (beyond point-to-point), we first show that the active graph is an overlay for all active vertices.

Theorem 1 *For any valid safety assignment, the active graph preserves the distances between all active vertices.*

Proof. Let G be the original graph, and $G' = (V', A')$ the active graph for a given valid safety assignment. We must show that, for any two vertices v and w in V', $dist'(v, w) = dist(v, w)$ (the distance in G' is the same as the distance in G). Let P be the shortest v–w path in G; it suffices to show that G' has a v–w path of the same length (it cannot have a shorter path). Let $C_0, C_1, \ldots, C_q$ be the sequence of active cells containing at least one vertex of P, in the order induced by P. (Note that a cell may appear more than once in this sequence, i.e., we may have $C_i = C_j$ for some $i \neq j$.) These cells decompose P into a sequence of subpaths, each between two cells or within a distinct cell. The transition from C_i to C_{i+1} happens at a single boundary arc between two active cells; this arc belongs to A' by construction. Consider the maximal subpath a_i–b_i that is entirely contained within cell C_i. Because the subpath is maximal, a_i and b_i must be boundary vertices of C_i, and therefore they are active vertices. Moreover, because P is a shortest path in G, this subpath must be a shortest path as well, and the length of the (a_i, b_i) shortcut in G' will be equal to $dist(a_i, b_i)$. By concatening the appropriate boundary arcs and shortcuts in G', we obtain the desired path. □

This implies that a generalized MLD search will find the distances from its source to all active vertices (since it is just Dijkstra's algorithm on the active graph). In particular, a standard s–t MLD query (in which only cells containing s or t are marked unsafe) is correct. This theorem is stronger, however, and will allow us to generalize the basic algorithm to handle more sophisticated queries, as we will see next.

3.2 One-to-Many Queries

We first consider the *one-to-many* problem: we are given a source s and a set of POIs $\mathcal{P}$, and want to compute the distance from s to *all* vertices in $\mathcal{P}$ (not just the closest).

The trivial solution to this problem is to run Dijkstra's algorithm from s (in G) until all vertices in $\mathcal{P}$ are scanned. We can use the multilevel approach to accelerate this algorithm, with the help of Theorem 1. During the selection phase, we mark all nontrivial cells containing POIs (or s) as unsafe; we then run a generalized MLD search from s with this assignment. The final distance labels of the POIs are the answer to the one-to-many problem.

Note that the search starts from s as in Dijkstra's algorithm, but will only descends into (i.e., enters) cells that contain POIs, using shortcuts to skip over other cells. For this reason, we call this strategy

automatic descent. Since both s and the POIs are active vertices, the correctness of this approach follows from Theorem 1.

Note that the selection phase consists of simply marking all unsafe cells. This can be implemented by a bottom-up traversal of the multilevel hierarchy. In a first pass, we mark all level-1 cells that contain POIs, looking at each POI exactly once. We then process the remaining levels in increasing order, marking all level-i cells that contain at least one marked subcell at level $i-1$. If there are L levels and $|\mathcal{C}|$ cells (abusing notation) in total, this takes no more than $O(\min\{L|\mathcal{P}|, |\mathcal{P}| + |\mathcal{C}|\})$ time. As our experiments will show, in practice the selection phase takes only a few milliseconds even when there are tens of thousands of POIs to evaluate, making it ideal for online queries.

The running time of the query itself (the generalized MLD execution) depends not only on the number of POIs, but also on their distribution. For fixed $|\mathcal{P}|$, it will be faster if the POIs are close together, since the algorithm can skip large areas using higher-level cells. In contrast, if targets are spread over the graph, the algorithm must explore more low-level cells. The worst case happens when each level-zero cell has at least one target: we will visit exactly the same vertices as Dijkstra's algorithm. Even in this case, however, the algorithm has very little overhead.

A natural application of one-to-many queries is computing distance tables: given a set of sources S and targets T, compute an $|S| \times |T|$ table with the distances between them. This can be solved by using T as the set of POIs and running $|S|$ one-to-many queries. Note that it suffices to run the selection phase only once for all $|S|$ queries.

3.3 Finding Closest POIs

One-to-many queries can also be trivially applied to the k-closest POI problem. We simply run the selection phase on the set $\mathcal{P}$ of POIs, run a one-to-many query, then pick the k POIs with the smallest distances.

In practice, we can do better by running a restricted version of the one-to-many query. Since we scan POIs in increasing order of distance from s, we can stop as soon as we are about to scan the k-th POI. As our experiments will show, this technique is surprisingly effective in practice, and POI queries (for small values of k) tend to be even faster than point-to-point queries, since they are usually local. Although one can construct adversarial POI distributions where the pruned search is not much faster than a full one-to-many query, our experiments will show that automatic descent is quite efficient for distributions that actually occur in practice.

3.4 Best Via POIs

We now address the k-best via POIs problem. Given a source s, a target t, and a set $\mathcal{P}$ of POIs, we must find the k POIs $p_i \in \mathcal{P}$ that minimize $dist(s, p_i) + dist(p_i, t)$.

This problem can be trivially solved with two one-to-many queries. We use a forward query to find the distances from s to $\mathcal{P}$, and a backward query to find the distances from $\mathcal{P}$ to t. This provides all the information necessary to compute $dist(s, p_i) + dist(p_i, t)$ for every POI p_i.

An equivalent, but somewhat more efficient, version of this approach is to run both forward and backward searches at the same time, using two priority queues. We maintain a candidate set with the k best POIs seen so far (i.e., those whose corresponding via distances are minimized). Whenever we scan a POI arc for the second time, we update the list appropriately. We can stop as soon as the radii of both searches (given by the top values in their respective priority queues) are at least as high as the lengths of the k best paths found so far. To maximize the effectiveness of this criterion, we always advance the search (forward or backward) with minimum radius.

Note that this pruning is much less tight than the one mentioned in Section 3.3, since we must wait until all potentially good POIs are found by both searches. As a result, these queries are fast only for very local queries. With enough POIs, long-range queries can be quite slow.

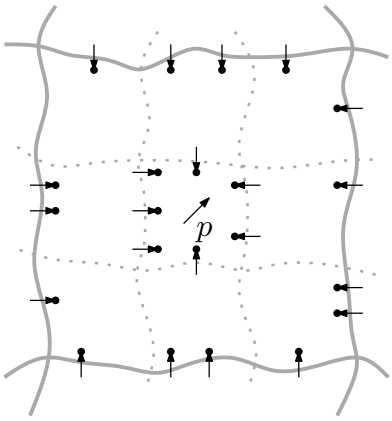


Figure 2: Example of single-source indexing. We add p to the buckets of the entry vertices of p's cells. Level-1 cells are indicated by dotted lines, solid lines show level-2 cells.

4 Index-Based Approaches

To overcome the limitations of automatic descent, we now consider an index-based approach. It provides a different trade-off: substantially faster queries, but worse selection times and space requirements. As our experiments will show, however, selection is still reasonably fast (a few seconds sequentially), making the indexed-based approach applicable not only in offline scenarios, but also some online ones.

Recall that the automatic descent approach can be slow because it must visit all cells that contain POIs, and they may be numerous. The index-based approach avoids this by precomputing (at selection time), for every cell containing a POI, the information that the automatic descent approach would learn at query time. This information (the *index*) is then associated with elements (arcs or boundary vertices) of the cell itself. The query algorithm can then gather all the information it needs without descending into the cell. Since the selection phase no longer needs to mark POI cells as unsafe, the number of vertices visited during the POI query is similar to that of a point-to-point query, regardless of the number (or location) of POIs in the system.

We propose two variants of this approach: *single-source indexing* applies to the closest POI problem, while *double-source indexing* applies to the best via path problem.

4.1 Single-Source Indexing

We first consider *single-source indexing*, an acceleration to the closest POI problem. This idea is related (but somewhat different) to the *bucket-based approach* [30] developed in the context of one-to-many computations using hierarchical speedup techniques. This section will discuss our approach as applied to multilevel overlays; the relationship to the hierarchical variant is deferred to Section 5.

To index a cell C, we associate with each entry point v of C a *bucket* $B(v)$ containing the k POIs $p \in \mathcal{P}$ within cell C minimizing the distance $dist(v, p)$, together with the distances themselves; if C has fewer than k POIs, the bucket includes all. See Figure 2.

Assuming such an index is in place (its efficient construction will be discussed shortly), consider how it can be used to accelerate k-closest POI queries from *any* source s.

The query is a forward generalized MLD search from s in which only the cells containing s are considered unsafe. It is a standard search, with minor adjustments.

First, we maintain a list L (initially empty) containing the best k candidate POIs found by the algorithm so far. Moreover, before scanning each vertex v, we examine each entry $(p_i, dist(v, p_i))$ in the associated bucket $B(v)$. We compute the tentative distance from s to p_i through v (given by $dist(s, v) + dist(v, p_i)$),

and add p_i to L if it is among the lowest k seen so far. (Note that this may involve simply adding a new entry or replacing another, possibly associated with p_i itself.)

Theorem 2 *Indexed single-source queries are correct.*

Proof. Let p_i be the i-th closest POI to s (for $i \leq k$). Take the active cell C containing p_i. Consider the largest suffix of P entirely contained in this cell, and let v be the starting vertex of this suffix. Since v is a boundary vertex of an active cell, it is an active vertex. Theorem 1 ensures that a forward MLD search will find the exact s–v distance. When v is scanned, we claim $B(v)$ contains p_i; if it did not, C would contain k other POIs q_j with $dist(v, q_j) < dist(v, p_i)$, contradicting the optimality of p_i. The shortest s–p_i path will thus be considered, completing the proof. □

Note that the generalized MLD search can stop as soon as it is about to scan a vertex v whose distance label is greater than the k-th best POI found so far. No further improvements to the candidate set are possible at this point.

Having established that queries are correct, we now consider the selection phase, which builds the index itself.

We propose three approaches to compute the buckets. To analyze them, we consider the time to process a cell C containing m_C edges, p_C POI arcs, and b_C boundary vertices, and assume we index no more than k POIs. For simplicity, let $k \leq p_C$; if not, just redefine k as p_C.

With *forward indexing*, we perform a forward search (restricted to the cell C) from each entry vertex, finding the distance to all POI arcs within C. This approach visits $O(b_C m_C)$ edges in the worst case. In practice, each search can stop as soon as k POIs are visited.

This approach is applied to cells in bottom-up order. To process cells at level 1, we only visit arcs of the original graph (using the expanded representation). For level $i > 1$, we only visit arcs (and their buckets) at level $i - 1$.

Note that the operations performed by forward indexing are similar to the standard customization phase of CRP [11, 16]: as in Section 2.3, we simply run a search from each boundary vertex of each cell, bottom-up.

A second natural approach is *reverse indexing*. We run a backward search from each POI arc, adding its POIs to the buckets of all entry points visited. This visits $O(p_C m_C)$ arcs in the cell, and may be faster than the previous approach when the cell has fewer POIs than boundary nodes. Note that reverse indexing does not need to be run level by level; it suffices to run a single MLD search from each POI arc until all cells that contain it are visited. The search can be pruned at the boundary vertices of the top-level cell.

We propose a third approach, *bulk indexing*, which can be significantly more efficient in practice than the previous two. As in reverse indexing, the idea is to perform backward searches starting from the POIs. Instead of performing p independent searches, however, we perform a single search that starts from all POI arcs within a cell at once, as we explain next.

Each vertex v maintains a label $L(v)$ consisting of a set of k pairs $(p, dist)$, representing the k (tentatively) closest POIs reachable from v. (For entry vertices, the labels will eventually correspond to the buckets.) All labels are initially empty, except those corresponding to vertices incident to POI arcs. We say an entry in the label of a vertex v is *stale* if it was already present (with the same distance) when v was last scanned by the algorithm; otherwise, we say that the entry is *fresh*. During the algorithm, we maintain a heap (priority queue) of vertices, each with priority corresponding to the smallest (lowest-distance) fresh entry in its label.

Each step of the algorithm works as follows. First, we pick the top vertex v from the priority queue, with label $L(v)$. Consider an incoming arc (u, v). We update $L(u)$ in the obvious way, by extending every entry of $L(v)$ by $\ell(u, v)$ and checking if it is better than any entry in $L(u)$. If any entry in $L(u)$ is updated (thus becoming fresh), we add u to the heap (or update it) with priority equal to the lowest fresh distance. The algorithm stops when the heap runs dry, at which point the labels associated with the entry points will correspond to the appropriate buckets.

This is a standard *label-correcting* algorithm [39], and its correctness is immediate. To bound its running time, consider the following. Since all arcs have nonnegative lengths, scanning an incoming edge (u, v) (from v) cannot lead to a smaller entry in $L(u)$ than in $L(v)$. This means that we scan vertices in nondecreasing order of distance labels; when we scan a vertex v, its minimum-distance fresh entry is final. After at most k scans, all entries in $L(v)$ will have no more fresh entries. Since the total degree of all vertices is $O(m_C)$, the total number of edges scans is bounded by $O(km_C)$.

Since each scan must merge two sorted $O(k)$-sized arrays and may result in one heap operation, the total running time per cell is $O(k^2 m_C \log m_C)$. (As long as k is small at selection time, the overhead of processing several entries at once is actually quite small, since they are represented cache-efficiently.) Recall that we are assuming that $k \leq p_C$; if not, we can replace k by p_C in the running time, for even better time bounds.

As in the forward approach, bulk indexing is applied one level at a time. On level 1, we start from the POI arcs; on level $i + 1$ (for $i > 1$), we start from the buckets computed on level i.

4.2 Double-Source Indexing

We now discuss *double-source indexing*, a strategy to accelerate k-best via node queries. Since we are interested in paths, we must associated buckets with *arcs* instead of vertices.

More precisely, double-source indexing associates a bucket $B(v, w)$ to each arc (v, w) in the graph. If (v, w) is an arc of the original graph, $B(v, w)$ contains the POIs assigned to the arc in G. If (v, w) is a shortcut arc for a cell C, $B(v, w)$ has the best k POIs p_i within C for (v, w), i.e., it contains what would be the k via POIs for v–w if C were the entire graph. In either case, the entry corresponding to POI p in bucket $B(v, w)$ also holds the actual length of the shortest v–p–w path (restricted to the cell). See Figure 3 for an illustration. As Section 5 will explain, this approach is related to the double-hub indexing strategy [1] used for hierarchical speedup techniques.

Section 4.2.1 describes how such an index can be used to accelerate via POI queries. Section 4.2.2 explains how the index can actually be built.

4.2.1 Queries

With the index in place, running an s–t via query is straightforward. First, we mark only the nontrivial cells containing s and t as unsafe, then execute two simultaneous generalized MLD searches: a forward search from s and a backward search from t (we can alternate between the two searches in any way). For

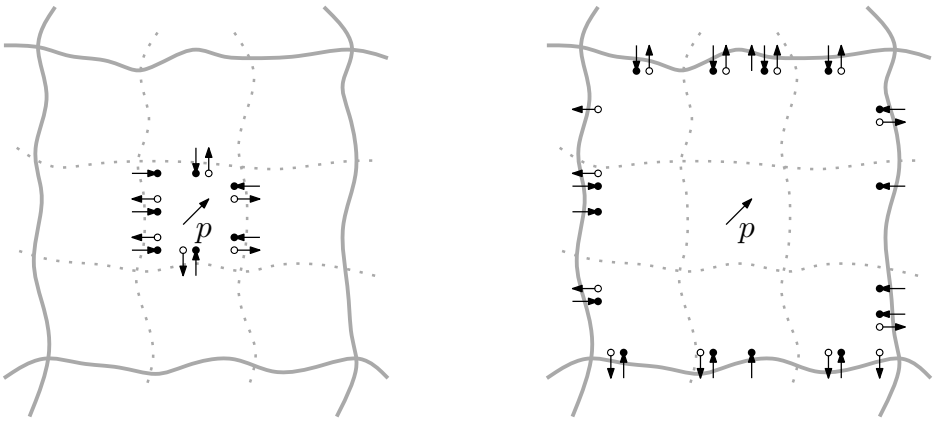


Figure 3: Double-source indexing. POI p contributes to all the shortcuts between entry and exit vertices of the cell containing p on levels 1 (left) and 2 (right).

each active vertex v, we will have two distance labels ($d_s(v)$ and $d_t(v)$), representing the exact distances from s and to t, respectively. These labels are initially infinite, except for $d_s(s)$ and $d_t(t)$ (which are zero). During the algorithm, we maintain a list $\mathcal{L}$ with the k POIs leading to the shortest via paths found so far.

Consider what happens when we process a vertex v in the forward direction (the backward search works similarly). We must process all outgoing arcs (which may include shortcuts, boundary arcs, and original arcs, depending on where v is relative to s and t). Consider one such arc (v, w) and its associated bucket $B(v, w)$. As in the standard generalized MLD search, we update w's (forward) distance label considering the actual cost of the arc. In addition, if w is already scanned by the backward search, each POI in $B(v, w)$ will give us a valid via path; if it is among the k shortest known via paths, we update $\mathcal{L}$ appropriately. After the search is complete, $\mathcal{L}$ will contain the k best via POIs between s and t.

Theorem 3 *Indexed double-source queries are correct.*

Proof. We consider the case $k = 1$ for simplicity; the same argument applies to higher k. We must show that, for any source s and target t, we will find the POI p^* that corresponds to the optimum (minimum length) via path s–p^*–t. Let this path by P^* in the original graph G. We have to prove that the algorithm will find a path with the same cost in the active graph.

Let (v, w) be the arc (in the original graph) to which p^* is assigned. Let a be the last active vertex on the s–v subpath of P^*, and let b be the first active vertex on the w–t subpath. We claim that the active graph must contain an arc (a, b) and that $p^* \in B(a, b)$. If a and b belong to different active cells, then $(a, b) = (v, w)$ is a boundary arc, and p^* is assigned to (v, w) itself. Otherwise, let C^* be the active cell containing both a and b. By our choice of a and b, a must be an entry vertex and b an exit vertex, and (a, b) is a shortcut within C^*. Moreover, p^* must be the best via node within C^* for the path a–p^*–b (if there was a better via node, it would give us a better solution than P^*, contradicting its optimality). Therefore, $p^* \in B(a, b)$. Since a and b are active, Theorem 1 ensures that the forward search will find $dist(s, a)$ and the backward search will find $dist(b, t)$. Assume that a is scanned after b (the symmetric case is similar): while scanning a, the forward search will look at $B(a, b)$ and consider a valid via path with the same length as P^*. Since no shorter path exists, this path will be returned. □

Note that the algorithm can stop as soon as the top values in the priority queues are both larger than all k best paths found. To maximize the effectiveness of this criterion, we alternate between the forward and backward search by radius, always picking the side with the smallest value.

4.2.2 Indexing

Having established that queries are correct, we now turn our attention to the selection (indexing) phase. Recall that, for each shortcut (v, w) in the overlay graph, we must build a bucket $B(v, w)$ containing the k best via points between v and w within (v, w)'s cell.

The straightforward approach is *POI-based* indexing. We initialize all buckets as empty. We then process each original POI arc (a, b) by running a forward MLD search from b and a backward MLD search from a; both searches can be pruned at the boundary of the level-L cell containing (a, b) (the top level does not need to be visited in full). For each cell C that contains (a, b) (of which there are up to L), we consider all pairs (v, w) of entry and exit points in C, adding (a, b) to $B(v, w)$ with value $dist_C(v, a) + \ell(a, b) + dist_C(b, w)$. (Here $dist_C$ indicates the distance restricted to cell C.) Since it requires a separate search from each POI arc, POI-based indexing can be costly.

We therefore propose a potentially faster *level-based* indexing strategy, which processes each cell containing a POI only once. During initialization, we create all non-empty buckets at level zero by simply adding to $B(v, w)$ all POIs that are located at arc (v, w). It then processes non-trivial cells in bottom-up fashion, from level 1 to level L.

Consider a cell C at level $i \geq 1$. Let N be its set of entry points, and let X be its set of exit points. Let R be the set of all *relevant* level-$(i-1)$ arcs in C (those with nonempty buckets). Let T be the set of their tail vertices, and let H be their head vertices. To process the cell C, we create two temporary distance

tables, one with all distances from N to T, and the other with all distances from H to X. These tables can be built in a straightforward way, by running Dijkstra's searches on the level below. For the $|N| \times |T|$ table, we run $|N|$ forward searches if $|N| \leq |T|$, or $|T|$ backward searches otherwise. The $|H| \times |X|$ table is handled similarly.

For additional efficiency, we actually pick the graph search algorithm we use depending on the number of sources we need to compute distances from. If there are more than two left, we run the Bellman-Ford algorithm (starting from multiple sources at once [16]) instead of Dijkstra's.

Once the tables are in place, we fill each of the $|N| \times |X|$ buckets (one for each pair of entry and exit points). To fill $B(v, w)$, we go over all arcs $(a, b) \in R$ and compute the lengths (by evaluating $B(a, b)$) of the appropriate via paths (taking $dist_C(v, a)$ and $dist_C(b, w)$ from the distance tables), adding to $B(v, w)$ any via paths that are among its best k.

4.3 Hybrid Approaches

The indexing techniques introduced in Sections 4.1 and 4.2 have faster queries than the automatic descent approach introduced in Section 3, at the cost of significant more effort spent at selection time and higher space usage. For a smoother trade-off, we can use a *hybrid* approach: index only the lower q levels, and use automatic descent above that. The query algorithm is still generalized MLD, but with POI cells marked as unsafe only if they are above level q. Indexed cells are safe. This approach works for the k-closest POI and k-best via POI problems.

5 Related Work and Discussion

The POI-related problems have been extensively studied in the literature [8, 9, 26, 28, 31, 33, 36, 37]. One popular approach is to exploit the fact that road networks have associated coordinates, and use spatial data structures to filter relevant points of interest. Approaches such as these rely on there being sufficiently high correlation between Euclidean distances (as the crow flies) and graph-based shortest-path lengths, and they often rely on approximations. Both features severely limit our freedom to set the cost function.

Recently, there has been a greater push for exact solutions building on hierarchical speedup techniques for computing point-to-point shortest paths on road networks. The prototypical hierarchical technique is contraction hierarchies (CH) [21], but other methods can be used as well [4, 24, 34]. Conceptually, a typical hierarchical s–t (point-to-point) query is simply a pruned version of bidirectional Dijkstra's algorithm, in which the forward search (from s) and the backward search (from t) only traverse arcs that lead to "more important" vertices. These so-called *upward* searches typically visit a very small number of vertices (a few hundred for CH on continental road networks), orders of magnitude less than Dijkstra's algorithm. Preprocessing ensures that some important vertices are visited by both searches, and that the best such meeting point is on the shortest path. The search spaces can either be determined at query time or precomputed and stored as *labels*, as in the hub labels (HL) algorithm [2, 3].

Knopp et al. [30] introduced the *bucket-based* technique to leverage hierarchical methods to solve the one-to-many and many-to-many problems; it was later extended to the closest POI problem by Geisberger [20]. The selection phase of this algorithm performs an upward search (in the reverse graph) from each candidate POI p, with p itself added to the bucket $B(v)$ of every vertex v scanned by the search. The query stage runs an upward search from the source s; each entry of each bucket it finds gives a candidate path to a nearby POI. The shortest such path is returned.

The same technique could be applied as is to multilevel-based algorithms, since they have a hierarchical component. Since the upward search is much bigger for MLD, however, this would not be practical. The single-source indexing approach we introduced in Section 4.1 solves this problem by leveraging the fact that we have nested partitions. As we have shown, it is enough to add a POI p only to the buckets associated with the boundary vertices of the cells that contain p, which are typically a small fraction (less than 10%) of the entire search space. Moreover, the fact that we have partitions allows us to index the POIs with more ellaborate algorithms, such as forward and bulk indexing.

The *best via node* problem was not as well studied in the context of speedup techniques, since it is much less amenable to indexing. Abraham et al. [1] have recently proposed the notion of *double-hub indexing*, which (once again) builds on hierarchical methods. For this problem, buckets are associated with *pairs* of vertices. More precisely, given a POI p, let $S^+(p)$ be the forward search space from p and $S^-(p)$ be its backward search space. For each pair $(v, w) \in S^-(p) \times S^+(p)$, we insert p into bucket $B(v, w)$ with value $dist(v, p) + dist(p, w)$. The intuition here is that the optimum via path is a concatenation of two shortest paths (into p and out of p). An s–t via query then performs upward searches from s and t and looks at the cross product of the vertices $v \in S^+(s)$ and $w \in S^-(t)$. For each such pair (v, w), we check whether the bucket $B(v, w)$ gives a better via path through a POI. As our experiments will show, queries are fast with this method, but space usage can be quite high and customization is slow.

Once again, the double-source indexing approach we described in Section 4.2 has the same flavor, but leverages the fact that we have a nested partition to obtain a much more limited set of buckets. We do not have to consider the cross product of all vertices in the backward and forward search space. Instead, a POI p must only be indexed by the actual cells that contain it (and only within each level). This leads to much fewer bucket entries, and even queries can be quite fast. Again, the fact that we have a nested partition allows us to compute these buckets in a more efficient way.

Another approach building on CH is PHAST [10], which can perform one-to-all computations in road networks much faster than Dijkstra's algorithm. Although computing distances to all vertices is overkill for POI-based queries, the related RPHAST [13] algorithm uses a selection phase to allow faster one-to-many computations. For POI queries, we can then run a full one-to-many query and select the best candidate POIs. Unfortunately, it is hard to prune the search algorithm for local queries, even for the k-closest POI problem.

Finally, the fastest point-to-point algorithm, HL, computes the distance between two random points in well below one microsecond. Since these queries are so fast, it is often feasible to run HL queries from s (and t) to all POIs, and then, like for RPHAST, pick the k best POIs among those.

6 Experiments

We now present an experimental evaluation of the algorithms described in this paper. Our code is written in C++ and compiled with Microsoft Visual C++ 2012. Our test machine runs Windows Server 2008 R2 and has 96 GiB of DDR3-1066 RAM and two 6-core Intel Xeon X5680 3.33 GHz CPUs, each with 6×64 KB of L1, 6×256 KB of L2, and 12 MB of shared L3 cache. All executions are single-threaded.

Most of our experiments use a standard [17] benchmark instance made available by PTV AG representing the European road network. It has 18 million vertices and 42 million arcs, and its cost function represents travel times. Since the the network lacks real turn costs and restrictions, we enrich it with U-turn costs of 100 seconds [11]. We use arcs picked uniformly at random as locations for our POIs. We also performed tests on the road network of North America used by Bing Maps, together with realistic POI data sets (see Section 6.3); since these datasets are proprietary, however, most of our experiments use the European road network.

We report the effort (space and time) of the metric-independent preprocessing, the (metric-dependent) customization phase, and the POI selection. Query times are averaged over 1 000 sources (and targets, if needed) picked uniformly at random from the full graph. We compare our method against existing code for the fastest previous techniques [13]: RPHAST, HL with buckets (BHL), and plain HL (which computes all necessary distances with point-to-point hub label queries). Note that all these algorithms must operate on the expanded graph to deal with turns properly; for Europe, this graph has 42 million vertices and 115 million arcs.

For CRP, we use a 5-level setup with maximum cell sizes 2^8, 2^{11}, 2^{14}, 2^{17}, and 2^{20}. We use contraction for customizing the lowest level, and Bellman-Ford search for the levels above [16].

Table 1: Main results for finding POIs from a single location ($|\mathcal{P}| = 16384$).

	PREPROCESS		CUSTOM		SELECTION		4-CLOSEST		ALL	
algorithm	space [MiB]	time [s]	space [MiB]	time [s]	space [MiB]	time [s]	#scan nodes	time [ms]	#scan nodes	time [ms]
Dijkstra	—	—	—	—	—	—	7096	1.95	29297531	13367.92
RPHAST	1519	8170.1	—	—	32.18	0.28	651974	11.98	651974	11.95
HL	64396	11652.3	—	—	—	—	—	7.26	—	7.23
BHL	64396	11652.3	—	—	13.72	0.21	—	0.01	—	2.16
CRP no index	3119	5190.6	71.0	3.9	0.00	0.01	1626	0.64	12319771	8081.23
CRP reverse index	3119	5190.6	71.0	3.9	33.27	9.12	443	0.17	3933	6.77
CRP bulk-4 index	3119	5190.6	71.0	3.9	5.73	2.20	443	0.17	—	—
CRP reverse index-2	3119	5190.6	71.0	3.9	3.53	1.78	451	0.17	198602	99.31
CRP bulk-4 index-2	3119	5190.6	71.0	3.9	3.53	1.69	451	0.17	—	—

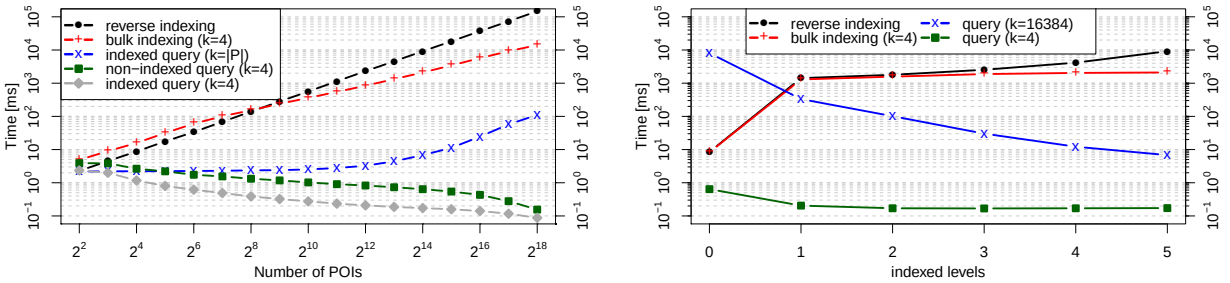
6.1 Single-Source Queries

In our first experiment, we consider two variants of closest POI queries (with $|\mathcal{P}| = 16384$): 4-closest POIs and $|\mathcal{P}|$-closest POIs (one-to-many). Table 1 compares our methods to competing ones. For CRP, we test many variants: no index, reverse indexing, and bulk indexing with $k = 4$. (Recall that bulk indexing requires limiting k.) Both indexing techniques are tested in full and up to level 2 (in a hybrid version).

We observe that CRP can find the 4 closest POIs in well below one millisecond, even without indexing. (While technically there is a selection phase, it takes only 8 ms to mark all relevant cells.) With an index, CRP is faster than HL and RPHAST, but has slower selection. However, only CRP can handle metric updates quickly (its customization phase takes less than 4 seconds). Surprisingly, even Dijkstra's algorithm is fast enough for finding the 4 closest POIs: choosing POIs uniformly at random from the graph makes it very likely that there are many POIs close to any source, allowing Dijkstra's algorithm to stop very soon. As will see in Section 6.3, this is not true for more realistic inputs.

When computing distances to all POIs, Dijkstra's algorithm is much slower, since it visits almost the full graph. Without indexing, CRP is slow as well (around 8 seconds), since it must explore many cells on the lowest level. Indexing all levels reduces query times to well below 10 ms. Similarly, BHL takes much longer to compute distances to all POIs than to the closest 4. Other methods depend less on the query type: for RPHAST and HL, both queries take the same time. Interestingly, although RPHAST scans about 200 times more vertices than with indexed CRP, its queries are only twice as slow, since it has (by design) much better locality [13]. Again, the main advantage of CRP is that it is applicable in dynamic scenarios, and queries are still fast enough.

We next evaluate the impact of varying the number of POIs. Figure 4(a) reports the selection (using reverse and bulk indexing with $k = 4$) and query times for finding the 4 closest POIs (with and without an index), as well as the time to compute the distance to all POIs (with an index). As expected, reverse



(a) Varying the number of POIs with $k = 4$.

(b) Varying the number of indexed levels with $|P| = 16384$.

Figure 4: Closest POI queries.

indexing time is proportional to the number of POIs, and bulk indexing is faster when $|\mathcal{P}|$ is large. Closest POI queries, both indexed and non-indexed, become faster with the number of POIs, since the searches can stop sooner. When computing the distances to all POIs, query times are rather stable up to 2048 POIs, then become slower; even though the number of buckets scanned is the same (about 4000), they eventually become too large to fit in cache. When $|\mathcal{P}|$ is large, doubling it doubles query times. Still, running times are only around 100 ms for 262144 POIs.

Finally, Figure 4(b) shows the impact of indexing on the performance of CRP by considering various hybrid setups. We again fix $|\mathcal{P}| = 16384$, but vary how many levels we index. We test reverse and bulk indexing, and both types of queries. Indexing the lowest level takes 1.5 seconds, but helps both types of queries. Bulk indexing then spends little time on the higher levels, but closest POI queries hardly benefit from indexing additional levels. One-to-many queries benefit more, but require the costlier reverse indexing.

To summarize, for closest POI queries, CRP without an index is fast enough for practical applications if k is small, but we do need an index for large values of k.

6.2 Double-Source Queries

We now consider k-best via queries. Table 2 shows the main results for computing the $k = 1, 4, 16$ best via points for several methods, always with $|\mathcal{P}| = 16384$. Note that we here only use our level-based indexing technique for CRP, since it turns out to be superior to the POI-based approach, as further experiments will show. Moreover, to make non-indexed CRP faster, we use a sixth (lowest) level with maximum cell size $U = 2^4$; this triples the amount of metric-dependent data, with little effect on customization times.

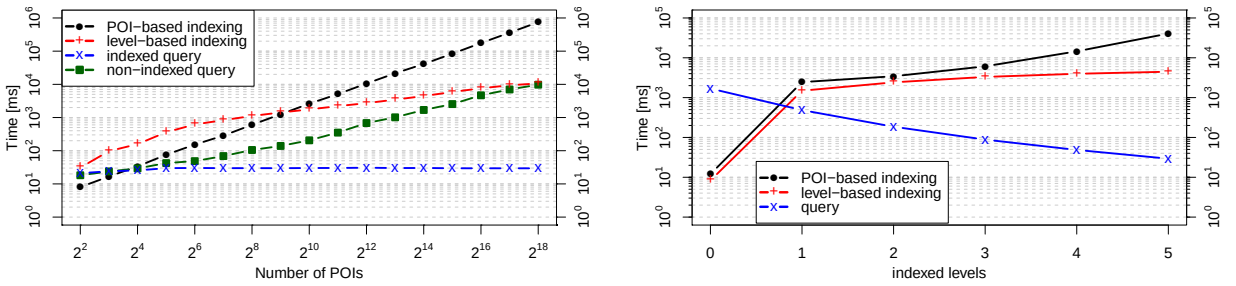
Unlike for closest POI queries, Dijkstra's algorithm or CRP without an index are too slow for interactive applications, since they lack a good stopping criterion. RPHAST and HL-based methods do allow fast queries, but are only applicable when fast metric updates are not needed. CRP queries are fast enough if a full index is available; building such an index takes roughly the same time as metric update (around 4 seconds sequentially). This is fast enough to support real-time traffic updates and personalized functions, particularly with multithreaded implementations. Indexing the lowest 3 levels only, however, does not pay off—indexing times decrease only by about 25% (the upper levels are small anyway), but queries become 3 to 4 times slower (we need to evaluate many more buckets).

We now consider the impact of the number of POIs on the performance of CRP. Figure 5(a) reports selection (POI- and level-based indexing) and query (indexed and non-indexed) times, always with $k = 4$. While doubling $|\mathcal{P}|$ doubles the time needed for POI-based indexing, level-based indexing scales much better (it depends more on the number of cells processed than the number of POIs). Interestingly, indexed queries take around 30 ms regardless of $|\mathcal{P}|$, but non-indexed queries are only practical for very small $|\mathcal{P}|$.

Figure 5(b) shows that indexing more levels (in a hybrid setup) helps via queries. Queries are too slow without an index, but get faster as more levels are indexed. For POI indexing, however, a lot of time is spent on indexing the lowest level, whereas the higher levels are rather cheap. This confirms our conclusions from analyzing Table 2: indexing all levels is the best option.

Table 2: Main results for best via node with $|\mathcal{P}| = 16384$.

	$k = 1$				$k = 4$				$k = 16$			
	SELECTION		QUERIES		SELECTION		QUERIES		SELECTION		QUERIES	
algorithm	space [MiB]	time [s]	#scan nodes	time [ms]	space [MiB]	time [s]	#scan nodes	time [ms]	space [MiB]	time [s]	#scan nodes	time [ms]
Dijkstra	—	—	36634115	19177.65	—	—	36676923	19314.13	—	—	36901721	19701.12
RPHAST	64.39	0.56	1307518	23.95	64.39	0.56	1307518	23.96	64.39	0.56	1307518	23.99
HL	—	—	—	18.10	—	—	—	18.20	—	—	—	18.23
HL index	631.97	65.22	—	1.23	933.69	66.45	—	1.37	1194.58	72.11	—	1.73
CRP no index	0.00	0.01	2502032	2019.91	0.00	0.01	2505562	2024.75	0.00	0.01	2508749	2027.56
CRP level index	77.22	4.32	7219	21.44	213.24	4.35	7517	30.21	569.24	4.54	7731	49.33
CRP level index-3	62.85	3.20	57092	70.98	155.80	3.24	57702	88.53	339.67	3.30	58580	128.74



(a) Varying the number of POIs with $k = 4$.

(b) Varying the number of indexed levels with $|\mathcal{P}| = 16384$ and $k = 4$.

Figure 5: Best via queries.

6.3 Other Inputs

Our last set of experiments now uses Bing Maps data representing North America, with 30 million vertices. It builds on Navteq data and includes turn restrictions and costs; the cost function correlates well with driving times. The POI data (also proprietary) contains the US locations of three large retail chains that operate coffee shops (about 6500 POIs), gas stations (also about 6500 POIs), and fast food restaurants (15 000 POIs). The first chain operates mostly on the East Coast; the others have national reach. For comparison, we again evaluate $|\mathcal{P}| = 16384$ random POIs.

Table 3 reports the key figures for finding the 4 closest POIs and 4 best via points with Dijkstra's algorithm and with CRP using an index or not. Without an index, we again use a sixth (lowest) level with $U = 2^4$. We observe that CRP has very similar performance on real and synthetic POI data, justifying our experimental setup. For Dijkstra's algorithm, however, closest POI queries are two orders of magnitude slower for realistic POIs, since they are not evenly spread: a search from Southern Mexico or Northern Canada will not stop before it reaches the US.

Summarizing, CRP with an index is fast enough for both query scenarios, whereas not having an index is only practical for the closest POI scenario. Building indices is always faster than customization times, and fast enough to allow personalized cost functions.

Table 3: Results for realistic inputs, Bing North America with realistic POIs ($k = 4$).

			PREPROCESS		CUSTOM		CLOSEST 4 POIS				4 BEST VIA			
			space	time	space	time	space	time	#scan	time	space	time	#scan	time
POI	algo	index	[MiB]	[s]	[MiB]	[s]	[MiB]	[s]	nodes	[ms]	[MiB]	[s]	nodes	[ms]
random	Dijkstra	—	—	—	—	—	—	—	12942	3.49	—	—	60817764	32535.30
	CRP	×	6547	941	385.3	8.12	0.00	0.01	671	0.31	0.00	0.01	2851277	2668.14
	CRP	✓	6547	941	136.5	8.12	7.01	2.50	530	0.19	289.47	7.11	7983	20.71
coffee	Dijkstra	—	—	—	—	—	—	—	1530782	563.49	—	—	61112567	33002.94
	CRP	×	6547	941	385.3	8.12	0.00	0.01	1767	0.73	0.00	0.01	970188	708.20
	CRP	✓	6547	941	136.5	8.12	2.40	0.87	924	0.36	98.55	2.63	8533	20.32
gas	Dijkstra	—	—	—	—	—	—	—	632699	255.56	—	—	61366200	33122.11
	CRP	×	6547	941	385.3	8.12	0.00	0.01	985	0.42	0.00	0.01	2739696	3094.02
	CRP	✓	6547	941	136.5	8.12	3.35	1.13	696	0.24	158.10	4.10	8030	21.07
food	Dijkstra	—	—	—	—	—	—	—	545067	206.02	—	—	60951973	32881.72
	CRP	×	6547	941	385.3	8.12	0.00	0.01	641	0.28	0.00	0.01	1337559	995.59
	CRP	✓	6547	941	136.5	8.12	6.38	2.16	582	0.20	251.16	6.01	7738	20.39

7 Conclusion

We extended the CRP routing engine to rank POIs based on arbitrary length functions from up to two locations, without losing key features of CRP, such as supporting real-time traffic updates or personalized cost functions. Our experiments on *fully realistic data sets* even showed that CRP is fast enough to enable *personalized* ranking functions of POIs, making it superior to previous approaches based on spatial information or hierarchical routing algorithms. We are optimistic that CRP can be further generalized to deal with multi-POI queries [32].

References

[1] I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. HLDB: Location-Based Services in Databases. In *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*, pp. 339–348. ACM Press, 2012.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In P. M. Pardalos and S. Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, Lecture Notes in Computer Science 6630, pp. 230–241. Springer, 2011.

[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In L. Epstein and P. Ferragina, editors, *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*, Lecture Notes in Computer Science 7501, pp. 24–35. Springer, 2012.

[4] J. Arz, D. Luxen, and P. Sanders. Transit Node Routing Reconsidered. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, Lecture Notes in Computer Science 7933, pp. 55–66. Springer, 2013.

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

[6] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

[7] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[8] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu. Monitoring Path Nearest Neighbor in Road Networks. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*, pp. 591–602. ACM Press, 2009.

[9] H.-J. Cho and C.-W. Chung. An Efficient and Scalable Approach to CNN Queries in a Road Network. In *Proceedings of the 31st International Conference on Very Large Databases (VLDB 2005)*, pp. 865–876, 2005.

[10] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

[11] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In P. M. Pardalos and S. Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, Lecture Notes in Computer Science 6630, pp. 376–387. Springer, 2011.

[12] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pp. 1135–1146. IEEE Computer Society, 2011.

[13] D. Delling, A. V. Goldberg, and R. F. Werneck. Faster Batched Shortest Paths in Road Networks. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, OpenAccess Series in Informatics (OASIcs) 20, pp. 52–63, 2011.

[14] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. Zweig, editors, *Algorithmics of Large and Complex Networks*, Lecture Notes in Computer Science 5515, pp. 117–139. Springer, 2009.

[15] D. Delling and R. F. Werneck. Customizable Point-of-Interest Queries in Road Networks. In *Proceedings of the 21st ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'13)*. ACM Press, 2013.

[16] D. Delling and R. F. Werneck. Faster Customization of Road Networks. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, Lecture Notes in Computer Science 7933, pp. 30–42. Springer, 2013.

[17] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS Book 74. American Mathematical Society, 2009.

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

[19] L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton U. Press, 1962.

[20] R. Geisberger. *Advanced Route Planning in Transportation Networks*. PhD thesis, Karlsruhe Institute of Technology, February 2011.

[21] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.

[22] R. Geisberger and C. Vetter. Efficient Routing in Road Networks with Turn Costs. In P. M. Pardalos and S. Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, Lecture Notes in Computer Science 6630, pp. 100–111. Springer, 2011.

[23] A. V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal on Computing*, 37:1637–1655, 2008.

[24] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS Book 74, pp. 93–139. American Mathematical Society, 2009.

[25] M. Holzer, F. Schulz, and D. Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

[26] H. Hu, D. Lee, and J. Xu. Fast Nearest Neighbor Search on Road Networks. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT'06)*, Lecture Notes in Computer Science 3896, pp. 186–203. Springer Berlin Heidelberg, 2006.

[27] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Effective Graph Clustering for Path Queries in Digital Maps. In *Proceedings of the 5th International Conference on Information and Knowledge Management*, pp. 215–222. ACM Press, 1996.

[28] C. S. Jensen, J. Kolár, T. B. Pedersen, and I. Timko. Nearest Neighbor Queries in Road Networks. In *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems (GIS'03)*, pp. 1–8. ACM Press, 2003.

[29] S. Jung and S. Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, September 2002.

[30] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pp. 36–45, 2007.

[31] M. Kolahdouzan and C. Shahabi. Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB'04)*, pp. 840–851, 2004.

[32] M. Rice and V. Tsotras. Exact Graph Search Algorithms for Generalized Traveling Salesman Path Problems. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, Lecture Notes in Computer Science 7276, pp. 344–355. Springer, 2012.

[33] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[34] P. Sanders and D. Schultes. Engineering Highway Hierarchies. *ACM Journal of Experimental Algorithmics*, 17(1):1–40, 2012.

[35] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.

[36] S. Shekhar and J. S. Yoo. Processing In-Route Nearest Neighbor Queries: A Comparison of Alternative Approaches. In *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems (GIS'03)*, pp. 9–16. ACM Press, 2003.

[37] S.-H. Shin, S.-C. Lee, S.-W. Kim, J. Lee, and E. G. Lim. K-Nearest Neighbor Query Processing Methods in Road Network Space: Performance Evaluation. In *IEEE International Conference on Network Infrastructure and Digital Content*, pp. 958–962, 2009.

[38] C. Sommer. Shortest-Path Queries in Static Networks, 2012. submitted.

[39] R. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[40] H. Yanagisawa. A Multi-Source Label-Correcting Algorithm for the All-Pairs Shortest Paths Problem. In *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*, pp. 1–10. IEEE Computer Society, 2010.