Customizable Point-of-Interest Queries in Road Networks

Daniel Delling
Microsoft Research Silicon Valley
Mountain View, CA, USA
dadellin@microsoft.com

Renato F. Werneck
Microsoft Research Silicon Valley
Mountain View, CA, USA
renatow@microsoft.com

ABSTRACT

We present a unified framework for exact point-of-interest (POI) queries in dynamic road networks. We show how to extend partition-based shortest-path algorithms to answer queries such as finding the closest restaurant or the best post office to stop on the way home, ranking POIs according to a user-defined cost function. We provide various trade-offs between indexing effort and query time. Our most flexible variant allows the road network to change frequently (to account for traffic or personalized cost functions) and the set of POIs to be specified at query time, and is still fast enough for interactive applications on continental networks.

Categories and Subject Descriptors

G.2.1 [Discrete Mathematics]: Combinatorics—Combinatorial algorithms

General Terms

Algorithms, Experimentation

Keywords

location services, points-of-interest, large road networks

1. INTRODUCTION

Modern spatial systems support a wide range of location services that deal with points-of-interest (POIs) with certain properties (such as open times, category, or personal preferences). Examples are “find the closest restaurant to me” or “find the best post office on my way home”. In general, we must rank POIs based on travel times to or from given locations. Since applying Dijkstra’s algorithm [16] takes too long, map services need more sophisticated solutions.

One approach is to augment hierarchical speedup techniques for point-to-point queries, such as contraction hierarchies [11] or hub labels [2]. They run an offline preprocessing phase to compute some auxiliary data, which is then used during an online phase to compute exact shortest paths. POI queries [1, 6, 10] use the fact that all shortest paths from a source (including those to POIs) go through a small set of “important” nodes (or hubs). An indexing step associates POI information with these hubs, leading to fast queries. Unfortunately, hierarchical methods have high preprocessing effort and space requirements, and are too sensitive to small changes to the cost function [5, 9].

We propose a unified framework for dealing with POI-related queries based on multilevel overlays [12, 13, 14]. Although this classical speedup technique has often been dismissed as uncompetitive [7], the recent customizable route planning (CRP) [5, 9] variant not only is fast enough for map services (such as Bing Maps), but also has other advantages, such as flexibility, space usage, predictable performance, realistic modeling, and robustness. The main idea of CRP is to move the metric-dependent portion of the preprocessing to a customization phase; it runs in seconds on continental inputs, supporting real-time traffic and personalized cost functions. Moreover, CRP supports realistic turn costs with little overhead and is robust to cost function changes.

We show how CRP can handle POI queries efficiently and robustly, with various trade-offs between indexing effort and query times. With an index, CRP finds the closest POI as fast as the best hierarchical methods, with much lower space overhead; it is still practical even without an index. We also present a CRP-based method to find the best via node on continental networks with tens of thousands of candidate POIs. CRP can update the cost function in seconds (compared to hours for hierarchical methods), making ours the only practical solution for many dynamic scenarios.

For additional details, see the full version [8] of this article.

2. PRELIMINARIES

We model a road network as a directed graph $G = (V, A)$, where each vertex $v \in V$ represents an intersection and each arc $(v, w) \in A$ a road segment. A metric (or cost function) $\ell : A \rightarrow \mathcal{N}$ maps each arc to a positive length (or cost) and associates a turn table $T_v$ to each vertex $v$; $T_v[i,j]$ is the cost of turning from the $i$-th incoming to the $j$-th outgoing arc. In the point-to-point shortest path problem, we must find the length $\text{dist}(s,t)$ of the cheapest path from a source $s$ to a target $t$, considering both arc and turn costs; $s$ and $t$ are points located anywhere along the arcs. This can be solved by Dijkstra’s algorithm [16], which scans vertices in increasing order of distance from $s$ and stops when $t$ is processed.

Let $P$ be the set of all POIs available, where each POI $p \in P$ is a location along an arc. In the $k$-closest POI
A generalized MLD search consists of running Dijkstra’s algorithm on the active graph; it finds the distances from the source to all active vertices. An s-t point-to-point query is just a generalized MLD search with the nontrivial cells (i.e., those on levels > 0) that contain s and t marked as unsafe.

More generally, in the one-to-many problem we are given a source s and a set of POIs P, and must compute the distance from s to all vertices in P. To solve it, we first mark nontrivial cells containing POIs (or s) as unsafe (during a selection phase), then run generalized MLD from s. It only descends into (enters) cells that contain POIs, and uses shortcuts to skip other cells. Since both s and P are active, this automatic descent strategy is correct. On L levels and |P| cells, selection takes O(min{L|P|, |P| + |C|}) time, or a few milliseconds in practice (fast enough for online queries). For fixed |P|, queries are faster when POIs are close together, since large areas can be skipped. Moreover, queries have little overhead compared to Dijkstra.

To find k-closest POIs, we run a one-to-many query and pick the k POIs with the smallest distances. We can stop as soon as we scan the k-th POI.

Similarly, we can find k-best via POIs with two one-to-many queries (from s and into t), which allows us to compute \( \text{dist}(s, p_i) + \text{dist}(p_i, t) \) for every \( p_i \in P \). For efficiency, we run both searches at once (alternating by radius), and stop when the radii of both searches exceed the length of the k-th best path found so far. Note that this pruning technique only works well when the via paths themselves are short.

4. INDEX-BASED APPROACHES

Relative to online methods, index-based approaches provide a different trade-off: faster queries, but worse selection time and space. They precompute (at selection time), for every cell with a POI, the information the automatic descent approach would learn at query time. This information (the index) is then stored with arcs or vertices of the cell, allowing it to be skipped during queries. Such a query visits about as many vertices as a point-to-point query, for any P.

4.1 Single-Source Indexing

We first consider how single-source indexing can accelerate the k-closest POI problem. We index a cell C by associating with each entry point v of C a bucket \( B(v) \) containing the up to k POIs \( p_i \in P \) located in cell C that minimize \( \text{dist}(v, p_i) \), together with the distances themselves.

With this index in place, a k-closest POI query first marks the cells containing the source s as unsafe, then runs a modified forward generalized MLD search from s. It keeps a list \( L \) (initially empty) with the best k POIs found so far. Before scanning a vertex v, we examine each entry (\( p_i, \text{dist}(v, p_i) \)) in \( B(v) \). We add \( p_i \) to \( L \) if \( \text{dist}(s, v) + \text{dist}(v, p_i) \) is among the lowest k distances seen so far. We stop when the search radius exceeds the k-th best distance in \( L \).

We propose three approaches to compute the buckets during selection. We consider the time to process a cell C containing \( m_C \) edges, \( p_C \) POI arcs, and \( b_C \) boundary vertices.

With forward indexing, we run a forward search (restricted to C) from each entry vertex, finding the distance to all POI arcs within C and visiting \( O(b_C m_C) \) edges. (This is similar to the customization phase of CRP.) Each search stops when it visits k POIs. We process cells bottom-up, restricting the search to arcs (and buckets) on the level below.
Operation; the total time per cell is thus of entry points correspond to the appropriate buckets.

\[ u \] each incoming arc \((u, v)\)

priority given by the lowest-distance fresh entry in its label.

\[ v \] otherwise. The algorithm keeps a heap of vertices, each with priority queues are at least as large as all paths in \(K\).

It keeps a label \((v, \ell(v))\) improves, we update \(L(u)\) and \(u's\) entrance in the heap. When the heap runs dry, the labels of entry points correspond to the appropriate buckets.

This scans each vertex at \(O(k)\) times, since each scan makes a fresh entry final. Each of the resulting \(O(kmC)\) arc scans merges two sorted \(O(k)\)-sized arrays and causes a heap operation; the total time per cell is thus \(O(k^2mC \log mC)\).

4.2 Double-Source Indexing

We now discuss how double-source indexing can accelerate \(k\)-best via node queries. Here we associate a bucket \(B(v, w)\) to each arc \((v, w)\) (as opposed to vertex) in the graph. If \((v, w)\) is an original arc, \(B(v, w)\) has the POIs assigned to it; if \((v, w)\) is a shortcut for cell \(C, B(v, w)\) has the best \(k\) POIs within \(C\) for \((v, w)\). The entry for \(p\) in \(B(v, w)\) also has the length of the shortest \(v\)-\(p\)-\(w\) path restricted to \(C\).

Given such an index, consider an \(s\)-\(t\) via POI query. First, we mark the nontrivial cells containing \(s\) and \(t\) as unsafe then run two MLD searches (from \(s\) and into \(t\)), alternating by radius and keeping a set \(K\) of the best \(k\) via paths found so far. To process a vertex \(v\) in the forward direction (the backward case is similar), we examine its outgoing arcs, including shortcuts. For arc \((v, w)\), we first update \(w's\) forward label using \(\ell(v, w)\). In addition, if \(w\) is already scanned by the backward search, each POI in \(B(v, w)\) gives us a valid via path; if it is shorter that the best known path, we update \(K\) appropriately. We stop when the top values in both priority queues are at least as large as all paths in \(K\).

We now consider the selection (indexing) phase. For each shortcut \((v, w)\) in the overlay, it must build a bucket \(B(v, w)\) with the \(k\) best \(v\)-\(w\) via points within \((v, w)'s\) cell. One approach is POI-based indexing. All buckets are initially empty. We then process each POI arc \((a, b)\) by running a forward MLD search from \(b\) and a backward MLD search from \(a\), pruning both searches at the boundary of the top-level cell containing \((a, b)\). For each cell \(C\) that contains \((a, b)\), 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 \(dist_C\) is the distance restricted to \(C\).)

A faster approach is level-based indexing. At level zero, we initialize \(B(v, w)\) with all POIs on arc \((v, w)\). We then process non-trivial cells bottom-up. Let \(C\) be a cell at level \(i \geq 1\), let \(N\) and \(X\) be its sets of entry and exit points, let \(R\) be the set of all relevant level-\((i - 1)\) arcs in \(C\) (those with nonempty buckets), and let \(T\) and \(H\) be the sets of tail and head vertices of \(R\). We create (by running Bellman-Ford or Dijkstra on the level below) two temporary distance tables, from \(N\) to \(T\) and from \(T\) to \(H\), then fill the \(|N| \times |X|\) buckets as follows. To fill \(B(v, w)\), we go over all arcs \((a, b) \in R\) and use \(B(a, b)\) to compute the lengths of the appropriate via paths (taking \(dist_C(v, a)\) and \(dist_C(b, w)\) from the distance tables), adding to \(B(v, w)\) those that are among the best \(k\).

Double-source indexing has faster queries than automatic descent, but worse selection time and space usage. A hybrid approach (which also applies to single-source indexing) is to index the lowest \(q\) levels and use automatic descent above.

5. RELATED WORK AND DISCUSSION

Popular approaches to the \(k\)-closest POI problem use spatial data structures to filter relevant POIs [3, 4], but this often relies on approximations based on Euclidean distances, which restricts our choice of cost functions. The fastest exact solutions are instead based on hierarchical speedup techniques for point-to-point queries, which are pruned versions of bidirectional Dijkstra’s algorithm that visit only a few hundred vertices on continental inputs. The search spaces can be found at query time, as in contraction hierarchies (CH) [11], or precomputed as labels, as in hub labels (HL) [2].

A bucket-based technique leverages hierarchical methods to solve the one-to-many [15] and closest POI [10] problems. Its selection stage adds each POI \(p\) to buckets associated with the hubs (vertices) in \(p\)'s search space. Queries examine the buckets linked to the hubs of the source \(s\); each bucket entry gives a candidate path to a nearby POI. The index we introduced in Section 4.1 works similarly but, because CRP uses nested partitions, we only need to add a POI \(p\) to the buckets associated with the boundary vertices of the cells that contain \(p\), or less than 10% of the full search space.

Double-hub indexing [1] generalizes the bucket-based approach to the best via node problem by storing buckets with pairs of vertices (the cross product the backward and forward search spaces of each POI). We showed in Section 4.2 how to extend this to nested partitions. Since a POI must only be indexed by the cells that contain it (and within each level), we have much fewer bucket entries.

Another approach is to use RPHAST [6], a CH-based one-to-many algorithm, as a building block for finding \(k\)-closest
6. EXPERIMENTS

Our code is written in C++ and compiled with Microsoft Visual C++ 2012. Our 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. All runs are single-threaded. We use a benchmark instance (made available by PTV AG) representing the European road network (with 18 M vertices and 42 M arcs); its cost function reflects travel times with 100-second U-turn costs [5]. We generate $|P| = 16384$ POIs by picking arcs at random.

We report the effort (space and time) spent on (metric-independent) preprocessing, customization, and selection (indexing). We average over 1000 queries, with sources (and targets) picked uniformly at random. We compare against the fastest previous techniques [6]: RPHAST, HL, and BHL (point-to-point queries to all POIs). To handle turns, these three algorithms operate on an expanded graph (with 42 M vertices and 115 M edges) [9]. CRP works and only practical for $k$-best via POIs (although it cannot be pruned for local queries). Finally, one can run point-to-point queries from $s$ (and $t$) to each POI, then pick the $k$ best POIs among those. With a fast algorithm (such as HL), this can be competitive. Neither HL nor RPHAST supports fast customization.

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$k = 1$</th>
<th>$k = 4$</th>
<th>$k = 16$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>$36,634,115$</td>
<td>$19,177,65$</td>
<td>—</td>
</tr>
<tr>
<td>RPHAST</td>
<td>64.39</td>
<td>0.56</td>
<td>1.307,518</td>
</tr>
<tr>
<td>HL</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>BHL</td>
<td>63.97</td>
<td>65.22</td>
<td>1.23</td>
</tr>
<tr>
<td>CRP no index</td>
<td>0.00</td>
<td>0.01</td>
<td>2,502,032</td>
</tr>
<tr>
<td>CRP level index</td>
<td>77.22</td>
<td>4.32</td>
<td>7,219</td>
</tr>
<tr>
<td>CRP level index</td>
<td>213.24</td>
<td>4.35</td>
<td>7,517</td>
</tr>
<tr>
<td>CRP level index</td>
<td>569.24</td>
<td>4.54</td>
<td>7,751</td>
</tr>
</tbody>
</table>

7. REFERENCES