International Journal of Geo-Information

MDPI

Article

In-Path Oracles for Road Networks

Debajyoti Ghosh [1,*], Jagan Sankaranarayanan [2], Kiran Khatter [1] and Hanan Samet [3]

[1] School of Engineering and Technology, BML Munjal University, Haryana 122413, India; kiran.khatter@bmu.edu.in
[2] Google Inc., Sunnyvale, CA 94089, USA; jsan@google.com
[3] Department of Computer Science, University of Maryland, College Park, MD 20742, USA; hjs@umd.edu
* Correspondence: 4u.debajyoti@gmail.com

Abstract: Many spatial applications benefit from the fast answering to a seemingly simple spatial query: "Is a point of interest (POI) *'in-path'* to the shortest path between a source and a destination?" In this context, an in-path POI is one that is either on the shortest path or can be reached within a bounded yet small *detour* from the shortest path. The fast answering of the in-path queries is contingent on being able to determine without having to actually compute the shortest paths during runtime. Thus, this requires a precomputation solution. The key contribution of the paper is the development of an *in-path oracle* that is based on precomputation of which pairs of sources and destinations are in-path with respect to the given POI. For a given road network with n nodes and m POIs, an $O(m \times n)$-sized oracle is envisioned based on the reduction of the well-separated pairs (WSP) decomposition of the road network. Furthermore, an oracle can be indexed in a database using a B-tree that can answer queries at very high throughput. Experimental results on the real road network POI dataset illustrate the superiority of this technique compared to a baseline algorithm. The proposed approach can answer $\approx$ 1.5 million in-path queries per second compared to a few hundred per second using a suitable baseline approach.

Keywords: spatial databases; spatial coherence; spatial indexes; spatial data structures; geographic information systems; database query processing; shortest path problem; geospatial analysis

Citation: Ghosh, D.; Sankaranarayanan, J.; Khatter, K.; Samet, H. In-Path Oracles for Road Networks. *ISPRS Int. J. Geo-Inf.* **2023**, *12*, 277. https://doi.org/10.3390/ijgi12070277

Academic Editors: Wolfgang Kainz and Sisi Zlatanova

Received: 19 May 2023
Revised: 1 July 2023
Accepted: 3 July 2023
Published: 13 July 2023

1. Introduction

A bread and butter operation on a road network is finding the shortest path from a source to a destination and a natural corollary to that is the determination of whether a POI is along the shortest path or can be reached by making a small detour. This problem is called an "in-path" query. While this problem is deceptively simple, it almost always involves computing the shortest path or at least some part of the shortest path, which makes this operation slow. Given a set of POIs, the in-path query is a determination of which of the POIs lies on the shortest path or can be reached by taking a bounded detour from the shortest path. The detour can be expressed either by a fixed value (e.g., 1 km) or a percentage fraction (e.g., say 5% additional detour from the shortest path).

Now, there are many applications that require answering millions of in-path queries on a road network. Consider the following real-life scenarios where the notion of in-path is of critical importance.

1. **In-path POIs** A coupon company wants to place digital coupons on route planner apps. The coupons of participating businesses that are in-path to the user's shortest paths are placed on the maps app. Since routing apps are quite popular, there may be millions of drivers requiring quick lookups of the coupons that are in-path to their shortest paths. Related use-cases include ride-sharing [1–4], location-based POI recommendation [5–11], package delivery [12], location-based advertisement [13–16], or scenic spots [17–20].

2. **Analysis or Simulation Queries** A city wants to choose a COVID vaccination camp and wants to choose one among hundreds of possible locations. One of the considerations among others is that these camps should be in-path to many commuters from a target group whose home and work locations are known. The city takes the daily commute routes of their target group and chooses the one location for the camp that is in-path to the most number of commuters. Related use-cases here include evacuation or shelter planning [21–26], location planning for new businesses [27], choosing new sites for playgrounds, etc.

Both these queries require computing millions of in-path queries per second and at high throughput. When a user is interacting with a routing app, the window of opportunity for displaying coupons or gas stations along the way is very short. Answering in-path queries requires finding the shortest path or at least part of it. Our challenge in this paper is doing so without having to actually compute the shortest path during runtime.

We introduce an in-path oracle that is able to answer millions of in-path queries per second. An oracle [28–31] is a data structure that is able to answer a query not by computing but by looking up a precomputed result quickly. In our case, the in-path oracle is able to make the determination for any arbitrary source and destination if a POI is along the shortest path or not. The key innovation here is that to make this determination, we do not have to compute the shortest path between the sources and the destinations. The construction of the oracle involves precomputation and storing all the pertinent information needed to answer the in-path queries. The precomputation is done in an offline process which aids in our stated goal of building a spatial platform that can support in-path queries at high throughput. To the best of our knowledge, the high-throughput computation of "in-path" queries is not yet explored on a real road network. Our work builds on a body of road network oracles like *distance oracles* [29], and *path oracles* [30] that have been pioneered by the authors and others over the last decade. This work adds to the arsenal of high throughput operations one can perform on road networks. We list out the following important contributions in the paper.

Throughput: The in-path oracle relies on precomputing using an offline process so that queries can be answered at a high throughput rate. Once precomputation is done, our in-path oracle can answer 1.5M in-path queries per second on a single desktop machine. We compared our approach to the *dual* Dijkstra approach [12] and show that our approach is many orders of magnitude faster since it can answer in-path queries without computing the shortest path.

Size: The next consideration is the size of the oracles. We show both theoretically and experimentally that the size of our oracle is $O(m \times n)$ where n is the number of nodes in the road network and m is the number of POIs. In-path queries can be answered in $O(\log n + F)$ time where F is the number of POIs in the result. The complexity bound is achieved by appealing to the similarity to WSP decomposition [32–36] which provides us with the linear bound on the number of nodes on the road network. Note that the linear bound means that each POI requires just a few MB of storage and our approach can be scaled to a large number of POIs.

Computation Time: The precomputation takes about tens of minutes on a single machine for city-sized road networks. Note that, although not explored in the paper, since the computation is data parallel, there are straightforward techniques to speed up the precomputation by employing more machines.

Interoperability: The in-path oracle can be embedded, employed, and used in multiple ways. One of the methods described in this paper is embedding in a relational database system [37] and querying using SQL. This means that the in-path oracle can be used in complex querying scenarios.

Related work is discussed in Section 2. Next, Section 3 provides the background and problem statement. A dual Dijkstra based baseline algorithm for finding in-path POIs is provided in Section 4. We describe the in-path oracle in Section 5, while experimental results are provided in Section 6. Finally, concluding remarks are drawn in Section 7.

2. Related Work

In-path queries are closely related to shortest path findings and the prior work in this area can be categorised into two groups—oracle techniques [29–31] and node-importance based methods [38–41]. In the following, we provide an overview of a body of work in this area and broadly distinguish our in-path oracle from these approaches.

2.1. Path and Distance Oracles

Thorup and Zwick [31] introduced the theoretical frameworks for distance oracle and since then many approximate distance oracle approaches have been proposed that trade-off between preprocessing time, space requirements, and query time. Below we briefly describe some of the oracles built on road networks while noting that our in-path oracle extends the idea of spatial coherence in road networks.

For a road network with n nodes, Sankaranarayanan et al. [30] introduced a compact construct called the path distance oracle that takes $O(n)$ storage to encode the n^2 shortest paths between each pair of nodes. The path oracle's is predicated on the observed coherence between the spatial locations of the source and destination nodes and the shortest paths between them, which makes it easier to organise source and destination nodes into groups that have common nodes or edges on the shortest paths. Precomputing the distances between all possible sources and destinations forms the foundation of shortest path retrieval in a path oracle. Sankaranarayanan and Samet [29] extend the idea to network distances and propose approximate distance oracles for road networks based on spatial coherence. The approximate shortest distances between all possible nodes in the spatial network are precomputed and stored using a concise amount of space.

Peng and Samet [42] introduced City Distance Oracles (CDO) for local area road networks spreading over several cities to achieve fast *spatial analytic queries* on a city road network. Peng et al. [43] also propose SPark and Distance Oracles (SPDO), which is an approximate distance oracle for spatial analysis queries on continental road networks on Apache Spark which results in high throughput. Finally, Peng et al. [44] recent development called Distance Oracle System (DOS) can quickly process a huge number of spatial analytic queries on large road networks.

SILC [45,46] is a method to take advantage of the spatial coherence property between a single to multiple destination vertices. The shortest paths and distances are precomputed and stored using efficient SILC encoding. The Path-Coherent Pairs Decomposition (PCPD) [30] decomposes a road network based on WSP decomposition and approximately estimates the road network distance and the shortest path between any pair of nodes in a spatial road network.

2.2. Node-Importance Based Methods

Node-importance-based methods [38–40,47–49] focus on computing the network distances in the road network. The oracle approaches use the path coherence of road networks and in contrast these approaches use the graph nature of road networks. The node-importance approaches use the observation that some nodes in a spatial road network are more important than other nodes and every shortest path passes through at least one of these nodes. Node-importance-based methods [41,50–54] rank the nodes in a road network according to their importance and pre-calculate the shortest path distances among the important nodes to speed-up query processing. In contrast to these approaches, in-path oracles completely precompute away the graph information of a road network, relying on spatial information for answering in-path queries.

2.3. On the Way POI Search Approaches

Yoo and Shekhar [55] propose the *in-route nearest neighbour* query problem. Given a source and a destination, their method identifies a POI with the minimum deviation from the query path on the way to the destination. Chen et al. [56] explore the problem of path nearest neighbour query in dynamic road networks, where a user is travelling

towards a given destination from his/her current location and wants to avail of a service along the way. So, the path nearest neighbour query differs from the conventional nearest neighbour query in the sense that the neighbour is a detour along the shortest path. Each path nearest neighbour query builds a shortest path between the user's current location and the destination and then searches for the POI (i.e., with the smallest detour distance) that is closest to the shortest path relative to the entire journey path and his/her current query location. Our dual Dijkstra is a good stand-in for these approaches and experimental results show that our approach outperforms these techniques by many orders of magnitude.

2.4. Detour Queries

Obstructed detour queries were introduced by Saha et al. [57]. In their problem formulation, obstacles such as private property or a fence, and obstructed detours make straight line navigation impossible. In this formulation, one can define the notion of nearest POI which is similar in flavour to the POI approaches discussed earlier.

Shang et al. [58] propose optimization methods to quickly process the *best point* detour queries on road networks. From a source to a destination location on the road network, a user preferred path P is given. The best point detour query seeks quick identification of the best detour POI, i.e. detour with the lowest detour cost on the path to be travelling along P. Nutanong et al. [59] explore the problem of determining the shortest route between two locations that contain a stopover of a specific type. They are interested in minimising the overall travel distance between two locations while covering a specific kind of stopover. Note that our work with in-path oracle is more fundamental in the sense that it succinctly encodes the notion of what POI is in-path to every possible source and destination. This means that one can formulate these detour problems on top of our oracle and be able to process them efficiently.

3. Background

In this section, we first develop the preliminary concepts before describing the problem setup.

3.1. Road Network

In our formulation, the road network is modeled as a directed, and weighted graph, $G = (V, E, W)$, such that V corresponds to nodes which are road intersections, while E corresponds to edges which are directed road segments connecting two nodes. Let n, and m be the number of nodes and edges respectively in the road network. Edge weights represent either the travel distance or the travel times between two adjacent nodes v_i and v_j that have an edge between them. The edge weight $w(v_i, v_j)$ is strictly greater than zero. In other words, there are no negative or zero edge weights.

3.2. Shortest Distance

Given source s and destination t nodes, the shortest distance $d_N(s, t)$ is obtained by summing up edges along the shortest path between s and t. Note that since $w(.)$ denotes either distance or trip time, $d_N(s, t)$ denotes the shortest path using an appropriate unit. Note that G is fully connected such that given any two nodes u and v, there is a path from u to v and vice-versa, although they may not have the same distance due to the directed nature of the road network.

3.3. Detour

Given source s and destination t nodes, let $\pi(s, t)$ denote a simple path that is not necessarily the shortest. The detour of such a path is the difference in the network distance along $\pi(s, t)$ compared to $d_N(s, t)$. Furthermore, it is fairly trivial to see that the detour of any path is greater or equal to zero.

3.4. Detour Bound

We are interested in finding detours that are bounded by a fraction ϵ such that the detours are bounded by $\epsilon \times d_N(s,t)$. For instance, if $\epsilon = 0.1$, we allow up to 10% increase in the path distances to accommodate the detour, so, all paths between s and t of distance $\leq (1+\epsilon) \times d_N(s,t)$ are acceptable.

3.5. POI

A point of interest (POI) is an arbitrary node on the road network through which we want to route the shortest paths. There are m such POIs in the road network and these can be dynamically added or deleted as the algorithm proceeds.

3.6. In-Path POI

A traveller starts from a source node s and is driving to the destination node t. A POI p is said to be in-path if there is a path that passes through p that is a detour bounded by ϵ.

Now that we have provided the necessary details to define the problem setup, we can provide the problem statement that we solve in this paper.

3.7. Problem Definition

We are given a road network G, set P of m POIs, and a detour bound ϵ. A driver travels from source s and destination t, we want to find the set of POIs in P that are "in-path" under the conditions specified.

Now that we have provided our problem statement we can also discuss the limitations of our approach. First, we are interested in retrieving all the POIs that satisfy the detour constraints. Our focus here is maximising the *throughput* where one can answer millions of in-path queries a second using a single machine. Our solution is not geared towards a driver that wants to visit multiple POIs yet stay within the detour bound. In our model, the expectation is that the user is presented with the POI choices and may choose one of the in-path POI to visit. Such examples include coffee shops, restaurants, gas stations, vaccination clinics, etc. The driver is unlikely to visit another POI of the same kind. Our work is in the context of the placement of relevant POIs on a map as an opportunistic service where speed is of the essence, so, the composition of complex trips that include visiting multiple POIs is not the focus of this work.

4. Dijkstra-Based In-Path Algorithm

Over the years, there have been many variants, modifications, extensions, and case-specific (negative weight edges, small and fixed integer weight edges) alternatives proposed to Dijkstra's algorithm. Prominent among them are A* [60], Bellman–Ford algorithm [61,62], Moore's algorithm [63], Dial's algorithm [64], etc. that improved the original algorithm or applied it the algorithm to different use-cases. For instance, Moore's algorithm uses a queue (instead of a priority queue) that can handle negative edge weights. In our problem formulation, we require a way of not only finding the shortest paths but also visiting the detour paths in an incremental fashion.

In this paper, we focus on developing the dual Dijkstra algorithm [12], a Dijkstra based variant for finding POIs within the specified detour tolerance limit ϵ. This approach serves as a baseline algorithm for our oracle-based approach as well as a useful mechanism to highlight a few properties that we will use later. Here we are given a source s, a destination t, detour limit ϵ, and a set of POIs P. The algorithm fetches the subset of POIs in P that are in-path with respect to the given source and destination.

Algorithm 1 sets up the dual Dijkstra's implementation for determining the in-path POIs for a given source s and destination t. Our algorithm makes use of a priority queue Q of objects O, which are made up of triples, $v(O)$, $d(O)$, and $l(O)$, namely, the current node, road network distance, and the traversal label. To aid efficient storage representation, these handy functions are defined to access the fields from the object O. Note that $l(O)$ can either be a backward or forward scan or a POI. The current state of the algorithm is stored in two

data structures T and VISITED. T stores the forward and backward distances of a node from s and t. For instance, $T(v, \mathbb{B})$ and $T(v, \mathbb{F})$ provide the backward and forward distances of a node v, respectively. Another data structure VISITED$(.)$ helps avoid revisiting already visited nodes and is also initially empty. Since we have two simultaneous traversals, this data structure needs to keep track of which traversal (i.e., $\mathbb{B}$ and/or $\mathbb{F}$) has visited a node. For s and t, VISITED is marked TRUE, i.e., VISITED is initialised. We initialise the forward scans from s and the backward scans from t. Updation of the priority queue Q takes place with the forward and backward traversals from s and t respectively. The incoming and outgoing nodes are added into Q with their appropriate labels. Finally, d_N is the estimate of the shortest path distance computed so far, and *result* stores the POIs found thus far.

Algorithm 1 Setup for determining in-path POIs

1: Object $O \Leftarrow$ triple $\{v, d, l\}$ of node v, network distance d and label l indicating forward or backward traversal
2: $v(O)$, $d(O)$ and $l(O)$ are helper functions to access elements of O
3: $Q \Leftarrow$ min-priority queue of O ordered by distance from s. Q is initially empty.
4: $T(v_i, l = \{\mathbb{F}|\mathbb{B}\})$ is a table of forward or backward distances of v_i from s or t.
5: VISITED$(v_i, \{\mathbb{F}\|\mathbb{B}\})$ returns true or false if v_i has been visited by forward or backward scan.
6: VISITED$(s, \mathbb{F})$ = TRUE
7: **for** each outgoing edges $(s, v_i) \in E$ **do**
8: $\quad Q$.Insert$(O = \{v_i, w(s, v_i), l = \mathbb{F}\})$
9: **end for**
10: VISITED$(t, \mathbb{B})$ = TRUE
11: **for** each incoming edges $(v_i, t) \in E$ **do**
12: $\quad Q$.Insert$(O = \{v_i, w(v_i, t), l = \mathbb{B}\})$
13: **end for**
14: $d_N \Leftarrow \infty$ ▷ Holds estimate of $d_N(s, t)$
15: *result* $\Leftarrow \{\}$ ▷ Output of the algorithm

The primary approach is presented in Algorithm 2, which is based on the aforementioned setup and initialisation. Line 1 illustrates the termination condition of the algorithm. When the Q is empty, the algorithm stops. In other words, when the whole graph has been traversed or if the element at the front of the queue is at a distance more than d_N, the algorithm stops. We provide correctness proof for the algorithm in Lemma 1.

We start by looking into object O, which has been taken out of the Q front. When a POI is found at the front of the queue, the POI is in-path and is added to the result. This is shown in lines 2–5. In lines 6–10 it checks whether the current extracted object O from Q is already marked, i.e., we look at VISITED to check if the node has been already traversed. If so, we ignore it and continue with the traversal, otherwise, the node is marked as visited. To ensure that every object's label is correctly reflected in the VISITED data structure, we label every object whether the traversal is forward or backward. After that, in line 11, we check the corresponding entries of T, i.e. whether an update requires the current object's node $v(O)$ and the label $l(O)$. If the distance has already been computed, we stop trying to improve it. This is because of the best-first nature of the backward and forward traversals on G. At this point, we furthermore check whether both the labels (i.e., backward and forward) of node $v(O)$ are computed. When both the entries (i.e., backward and forward distance) entries are computed, then in line 14 we compute the current shortest path distance and compare it with the existing shortest path distance. In case, the current shortest path distance is less than the existing shortest path distance, we replace the existing shortest path distance with the current shortest path distance. Then in line 19, we add an object triple into Q which stores the fully resolved node's current shortest path distance and the POI node. When we say fully resolved, we mean that both the forward and backward traversals have reached the POI after which it is added back to Q.

Algorithm 2 Determine if p is in-path to s, t for a given ϵ

1: **while** !$Q.empty()$ && $O \Leftarrow Q.front()$ && $d(O) \leq d_N$ **do**
2: **if** $l(O) == \mathbb{POI}$ **then**
3: $result.add(v(O))$ ▷ POI is found
4: continue
5: **end if**
6: **if** VISITED$(v(O), l(O))$ == TRUE **then** ▷ Indicates cycle
7: continue
8: **else**
9: VISITED$(v(O), l(O))$ = TRUE
10: **end if**
11: **if** Not EXISTS $T(v(O), l(O))$ **then**
12: Update $T(v(O), l(O))$ with $d(O)$
13: **if** Both $(T(v(O), F)$ and $(T(v(O), B)$ EXISTS **then**
14: $d' = T(v(O), F) + T(v(O), B)$
15: **if** $d_N > d' \times (1 + \epsilon)$ **then** ▷ Updated once.
16: $d_N = d' \times (1 + \epsilon)$ ▷ d_N holds detour limit
17: **end if**
18: **if** $v(O) \in P$ **then**
19: Q.Insert$(v(O), d', \mathbb{POI})$ ▷ Add fully resolved POI to priority queue
20: **end if**
21: **end if**
22: **end if**
23: **if** $l(O) == F$ **then** ▷ Forward scan
24: **for** outgoing nodes v_i of $v(O)$ s.t. !VISITED$(v_i, l(O))$ **do**
25: $Q.Insert(v_i, d(O) + w(v(O), v_i)$,F)
26: **end for**
27: **else** ▷ Backward Scan
28: **for** incoming nodes v_i of $v(O)$ s.t. !VISITED$(v_i, l(O))$ **do**
29: $Q.Insert(v_i, d(O) + w(v_i, v(O))$,B)
30: **end for**
31: **end if**
32: **end while**
33: **return** *result*

Finally, lines 23–32 are based on the kind of the label, i.e., depending on $l(O)$, all the incoming nodes of $v(O)$ are added into Q in case of backward traversal. All the outgoing nodes are added into Q in the case of forward traversal. The algorithm continues until either the queue is empty or a decision for an in-path POI is reached. At this point, if *result* is empty, then it means that the algorithm failed to find an in-path POI (line 33).

Lemma 1. *The above algorithm correctly retrieves all the POIs in P that are in-path to the shortest path from s to t.*

Proof. Suppose $p \in P$ is a POI that may be in-path with respect to the shortest path between s and t. Here we have to be cognizant of two cases and handle them appropriately.

- Case 1: p lies on the way from s to t, so to reach p, no detour is required. In other words, p is part of the shortest path.
- Case 2: p does not lie on the way from s to t, so to reach p, a detour is required. The complexity, in this case, is that the determination of whether p is within the detour limit can only be made after the shortest distance between s and t has been determined. Therefore, the algorithm should seamlessly work for an arbitrary value of ϵ.

For Case 1, the algorithm can only stop when the object at the front of the queue is $d(Q.front()) > d_N$. Note that $d_N \geq d_N(s, t)$, so we are guaranteed that all the nodes in the shortest path are visited by the traversal before the algorithm fetches the POIs corresponding to Case 2.

Now for Case 2, similarly, the algorithm cannot stop before all nodes that have a distance of $d_N(s,t) \times (1+\epsilon)$ have been visited by both the forward and backward scans. Note that this means all the POIs qualifying for the detour requirement are retrieved by the algorithm. □

5. In-Path Oracles

The objective of this paper is to design an in-path oracle to efficiently answer the problem defined in section 3 whether a given POI is in-path with respect to a given source and destination provided during query time. The goal of an in-path oracle is to answer this question quickly using precomputed results. In this paper, we define a new kind of oracle but the concept of oracles in itself is not new. For instance, a distance oracle [29] provides an ϵ-approximate shortest distance, while a path oracle [30] provides both an approximate distance as well as the shortest path. In contrast, the in-path oracle proposed here retrieves the POIs that are in-path to the shortest path of the driver travelling from s to a destination t, subject to a detour limit ϵ.

The key idea behind in-path oracles is that it exploits the *spatial coherence* [46] property in road networks which observes that spatially adjacent nodes in a road network share common characteristics. The shortest paths and distances between nodes and their spatial locations are coherent with one another which forms the key principle behind the path and distance oracles [29,30].

The in-path oracle expands on the spatial coherence property in road networks. We know from path coherence that a set of source nodes A and destination nodes B may share the same shortest paths if A and B are sufficiently far apart from one another but the nodes in A and B themselves are close to one another. Since they share common paths it means that we can now determine if a given POI is in-path with respect to this group of nodes as opposed to single pairs of shortest paths. This is a powerful idea since it allows us to define the "in-path" property on groups of nodes as opposed to pairs of nodes resulting in dramatic space savings and efficient access.

In the rest of this section, in Section 5.1, we establish the sufficient conditions by which we can determine if a POI p is in-path to a set of sources A and destinations B. Next, we provide the algorithm for computing the in-path oracle in Section 5.2. We discuss how the oracle is stored and indexed in Section 5.3. Finally, discuss how the oracle is queried using a few examples in Section 5.4.

5.1. Identifying In-Path Property

In this section, we define the in-path property for a set of source nodes A and a set of destination nodes B. The sources A and destinations B are represented by a bounding box (later we restrict the box to be a quadtree [65–72] block) that contains all the nodes in A (and B). Let s be a randomly chosen representative source node in A while t is a representative destination node in B. Let p be the POI to which we want to make a determination if all the shortest path from all the sources in A to all the destinations in B are in-path to p. In other words, we want to determine if block-pair (A, B) is in-path to p. Note that we make this determination for a single POI and then repeat the process for all the m POIs.

Let r_a^F be the forward radius from s denoting the farthest distance of a node in A from s. Similarly, r_a^B is the backward radius to s denoting the farthest distance from a node in A to s. We can compute a similar radius for t in B. We define r_b^F and r_b^B denoting the forward and backward radius of B, which is depicted in Figure 1. For every pair of source nodes in A to every destination node in B, one can define the shortest path. Among these shortest paths, one can bound the shortest and longest shortest paths. These are captured by the following lemmas.

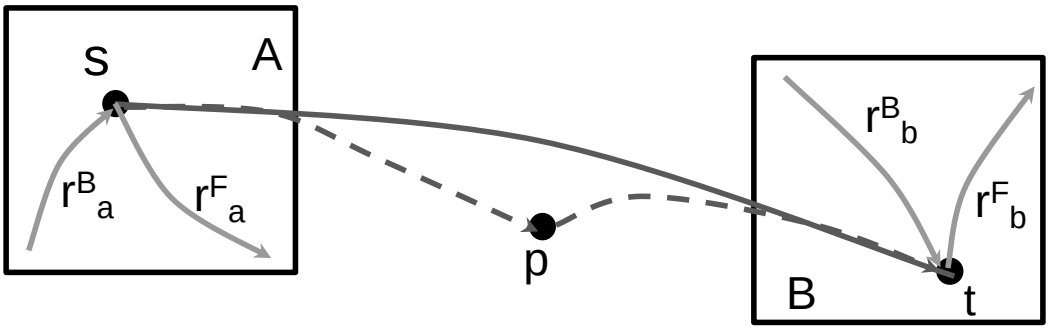


Figure 1. The mechanics of determining if p is in-path with respect to the shortest paths from multiple sources in A to destinations in B. Here (A,B) denotes a block pair and we develop rules to determine if p is in-path to (A,B).

Lemma 2. *The length of any shortest path between A and B is greater than or equal to* $d_N(s,t) - (r_a^F + r_b^B)$.

Proof. This can be trivially shown by counter example. If the path is smaller than that quantity, then one can go from s to t using a shorter distance than $d_N(s,t)$. Let us choose an arbitrary s' and t' to be an arbitrary source and destination and, furthermore, the network distance between them is smaller than the unit derived above. Then one can take a shortest path $s \longrightarrow s' \longrightarrow t' \longrightarrow t$. Note that $s \longrightarrow s'$ is bounded by r_a^F and $t' \longrightarrow t$ is bounded by r_b^B. Hence, $d_N(s',t')$ has to be greater than or equal to $d_N(s,t) - (r_a^F + r_b^B)$ else $d_N(s,t)$ is not the shortest distance between s,t, which would be a contradiction. □

Lemma 3. *The length of any shortest path between A and B is at most* $d_N(s,t) + (r_a^B + r_b^F)$.

Proof. Let us choose an arbitrary s' and t' be an arbitrary source and destination and, furthermore, one can define a path as follows: $s' \longrightarrow s \longrightarrow t \longrightarrow t'$. This path is bounded by $d_N(s,t) + (r_a^B + r_b^F)$. Hence, the lemma. □

Lemma 4. (A,B) *is in-path to* p *if the following condition is satisfied.*

Proof. For any given node s', t' in A, B, respectively, the network distance between $d_N(s',t')$ is at least $d_N(s,t) - (r_a^F + r_b^B)$ from Lemma 2. We can estimate a path that passed $s' \longrightarrow s \longrightarrow p \longrightarrow t \longrightarrow t'$. If p is in-path to $s' \longrightarrow t'$, then any path passing via p should be at most $r_a^B + d_N(s,p) + d_N(p,t) + r_b^F$. For the shortest possible path in A, B to be in-path with p, we have the following inequality:

$$r_a^B + d_N(s,p) + d_N(p,t) + r_b^F \leq (d_N(s,t) - (r_a^F + r_b^B)) \times (1+\epsilon)$$

$$\frac{r_a^B + d_N(s,p) + d_N(p,t) + r_b^F}{d_N(s,t) - (r_a^F + r_b^B)} - 1 \leq \epsilon$$

If the above inequality is satisfied for a particular pair of (A,B), then any shortest path between A and B is in-path to p. □

Lemma 5. (A,B) *is not in-path to* p *if the following condition is satisfied.*

Proof. For any given node s', t' in A, B, respectively, the network distance between $d_N(s',t')$ is at most $d_N(s,t) + (r_a^B + r_b^F)$ from Lemma 3. A path that passes through p is $s' \longrightarrow s \longrightarrow p \longrightarrow t \longrightarrow t'$ is at least $d_N(s,p) + d_N(p,t) - (r_a^B + r_b^F)$ if s', t' are part of the shortest path between s, p and p, t, respectively. If the shortest path between s', t' is not in-path to p, then we have the following inequality:

$$d_N(s,p) + d_N(p,t) - (r_a^B + r_b^F) \geq d_N(s,t) + (r_a^B + r_b^F) \times (1+\epsilon)$$

$$\frac{d_N(s,p) + d_N(p,t) - (r_a^B + r_b^F)}{d_N(s,t) + (r_a^B + r_b^F)} - 1 \geq \epsilon$$

If the above inequality is satisfied for a particular pair of (A, B), then any shortest path between A and B is not an in-path to p. □

Lemma 6. *If both conditions in Lemmas 4 and 5 are not satisfied, then some sub-division of A, B is in-path to p.*

Proof. Subdividing A and B reduces the radius of the blocks and could satisfy the conditions in Lemmas 4 and 5. In other words, if we cannot make a determination whether A, B is in-path to p, we need to keep subdividing it until children blocks are either empty, in-path, or not in-path. □

Figure 2 shows an example of both in-path and not in-path cases in practice. Here the POI is shown using a green circle. The set of sources and destinations are shown using blue colour. The common shortest path between a randomly chosen representative source and destination, and the resulting detour path are also shown. Now that the lemmas establishing the properties have been established, an algorithm for computing the in-path oracle for a road network is described next.

(**a**) In-path example (**b**) Not in-path example

Figure 2. For the Washington, DC dataset where the POI p is shown as using a green circle, figure (**a**) in-path and (**b**) not in-path example between a set of sources (block A) and destination (block B). The red dots indicate randomly chosen sources or destinations whose shortest paths are drawn.

5.2. Computing In-Path Oracle

Algorithm 3 provides the mechanics for computing the in-path oracle. The input to the algorithm is a road network consisting of nodes and edges. There is a quadtree on the spatial positions of the nodes such that R denotes the root block. Queue Q holds the quadtree block pairs. Note that Q is initialised with the block-pair (R, R). The algorithm continues till Q is empty (line 4). At every iteration of the algorithm, a block pair A, B is retrieved from the front of the queue. We choose a source s and destination t at random from A and B, respectively. We compute the shortest distance between s and t, i.e., $d_N(s,t)$, and the distance of the shortest path that passes through p, i.e., $d_N(s,p)$ and $d_N(p,t)$. We also compute the forward and backward radiuses of the blocks, i.e., r_a^F, r_a^B, r_b^F, r_b^B.

Algorithm 3 Determine in-path oracle for a given POI p

1: $R \leftarrow$ quadtree root block on the nodes of the road network.
2: *result* is the list of block-pairs forming the output.
3: $Q \leftarrow \{R, R\}$ is a Queue of block pairs.
4: **while** $!Q.empty()$ **do**
5: $(A, B) \leftarrow Q.pop\_front()$
6: Choose s, t at random from A, B, respectively.
7: Compute $d_N(s,t)$, $d_N(s,p)$, $d_N(p,t)$, r_a^F, r_a^B, r_b^F, r_b^B
8: **if** satisfies Lemma 4 **then**
9: Add (A, B) to *result*
10: **else if** satisfies Lemma 5 **then**
11: continue ▷ A,B is known to be not in-path
12: **end if**
13: Subdivide A and B into 4 children blocks. Discard empty children blocks if any.
14: Insert all children block-pairs into Q
15: **end while**
16: **return** *result*

Armed with the necessary components, one can use Lemma 4 if (A, B) denotes an in-path block-pair. If so, we add it to the result set. Similarly, Lemma 5 informs if the block-pair cannot be in-path in which case it is discarded. Finally, if both the conditions are not satisfied, we will simply break A, B into their 4 children blocks to form the resulting children blocks. These are inserted into Q. It is fairly easy to see that the children blocks might satisfy one of the two lemmas after a few decomposition steps. If one continues running the algorithm at the end we will get pairs of blocks such that they are all in-path with respect to p.

Referring back to the example in Figure 2a–b, one can see a block pair A, B denoting both the in-path and not in-path cases. Here we can see that Figure 2a corresponds to the case in line 9 where all the paths between A, B pass through the POI under consideration. The POI is shown by the green dot in the figures. Figure 2b corresponds to the case where we can determine that the p is definitely not in-path and the pair A, B can simply be discarded. This covers in line 11 in the construction algorithm. Finally, if some of the paths between A and B pass through p but some of them do not, we have to further subdivide. This is the case in line 13 where we have to further subdivide A, B into children block pairs and insert them into the priority queue.

Now, we have established the principles of the in-path oracle decomposition, for a single POI, we can estimate the size of the resulting decomposition in Lemma 7.

Lemma 7. *The size of the in-path oracle for p is $O(\frac{1}{\epsilon^2}n)$ since it is a Well-Separated Pair (WSP) decomposition of the road network.*

Proof. The size of the in-path oracle uses the same arguments as in [29]. Since there is a bit of background information that connects road network oracles (e.g., distance, path, and now in-path) to WSP, the interested reader is referred to [29,30,45,46,73] for the necessary background information. Here we will only provide a rough sketch of how the bounds are established. □

The first intuition is that the in-path oracle decomposition in Algorithm 3 is a WSP decomposition of the nodes of the road network. This can be seen from Lemmas 4 and 5 that the block-pair under consideration is in-path or not depending on the relative magnitude of the radius of the blocks to the distance separating the blocks. For instance, if the blocks are sufficiently "far" and their radiuses are small, the block may (or may not) satisfy the in-path condition. It is important to note that the WSP decomposition has an identical condition of determining if a block-pair is sufficiently far. Once the similarity has been established, we know that the size of the in-path oracle decomposition is $O(s^d n)$, where s is

the separation factor [33] and d is the dimensionality of the embedding space which is 2 for the case of road networks. In our case, s is dependent on ϵ as we discuss next.

Now, we can observe that, as ϵ increases (i.e., relaxed), more blocks easily satisfy the in-path oracle condition. Similarly, as ϵ decreases (i.e., made more stringent), more block-pairs can be marked off as not being in-path. So, there is a sweet-spot for ϵ that controls the size of the in-path oracle; setting it high or low results in a smaller in-path oracle. Since we are only interested in the worst-case size behaviour here, we can note that the size of the in-path oracle is inversely proportional to the ϵ. This can be intuitively observed from the fact that a smaller ϵ requires that the radiuses are much smaller than the distance between the blocks. In other words, the separation factor s is proportional to $\frac{1}{\epsilon^2}$ where the factor 2 denotes the dimensionality of the embedding space.

5.3. Storing the In-Path Oracle

From Lemma 7 we know that a single POI results in an in-path oracle of size $O(\frac{1}{\epsilon^2}n)$. For each entry, we store a block-pair (A, B) corresponding to the quadtree blocks as a four-dimensional Morton code [72,74–76]. In this scheme, a pair of Morton codes of the same level can be bit-interleaved to form a 64-bit number with the level of the block stored in the most-significant bit position of the code. Here 64 bits is sufficient to represent a fairly small area of less than 100 meters on the surface of the road network. This variation is also *linear* quadtrees [72] and forms the basis of storing multidimensional quadtrees on a database system. In the following we provide the size and access time results for our in-path oracle.

Lemma 8. *The in-path oracle for m POIs takes up $O(\frac{1}{\epsilon^2}nm)$ space and has an access latency of $O(\log n + F)$, where F is the number of POIs in the result.*

Proof. Given m POIs, we compute the in-path oracle for each POI. We store the identity of the POI along with each block-pair. The table containing these blocks is encoded as a four-dimensional Morton code, sorted and indexed by a B-tree [73,77–81]. The size of this representation is $O(\frac{1}{\epsilon^2}nm)$. Given a source s and destination t, we can first compute its four-dimensional Morton code, i.e., $Z_4(s, t)$ in $O(1)$. Next, we can use that key to look for matching blocks using the B-tree index. The time complexity of this operation is $O(\log n + F)$, where F is the number of POIs that match the results. This leads us to the storage and access complexity of the in-path oracles. □

5.4. Querying the In-Path Oracle

To illustrate how one can use the in-path oracle effectively, we consider a relational database [37,73,82] containing a few sample tables using which we can create a few scenarios. The oracle is stored as a table `ORACLE` (Z_{AB}, poi_id). Here Z_{AB} represents the block-pair that is known to be in-path with respect to poi_id. Here the attribute poi_id is a foreign key to another table `POI` $(poi\_id, Type, Name, \ldots)$ containing additional dimensional information about the POIs. Finally, there is a very large table of `ACTIVE_TRIPS`$(trip\_id, Z4\_st)$ where the starting and ending locations are encoded $Z4\_st$ as 4-dimensional Morton codes. Note that the `ORACLE` table is indexed by (Z_{AB}, poi_id) which means that one can binary search for a particular Z_4 pair in $O(\log n + \log m)$ time, where n is the number of nodes and m is the number of POIs. Once we have the following setup, we can apply some interesting queries to the in-path oracles.

Find all POIs that are in-path to a given source s and destination t: Here we are given a source s and a destination t and we want to find all the POIs that are in-path to the given source and destination. In this case, the query is a lookup on the `ORACLE` table using the primary index to fetch the in-path poi_id and joining it with the `POI` table to provide the dimensional fields of the POIs. This query is expected to be very quick since it is a simple lookup using a primary index. One can do millions of such queries per second using a single machine hosting the database which means that one can trivially scale up

this setup for use in the critical path of a popular navigation app to add an in-path layer during navigation.

```
SELECT poi_id, Type, Name
FROM POI JOIN ORACLE using (poi_id)
WHERE ORACLE.Z_AB = Z4_st;
```

In this setup, one can add new POIs by adding a row to the `POI` table and corresponding entries to the `ORACLE` table. Similarly, if one had to delete a POI, one just needs to delete the entry from the `POI` table and the delete operation would cascade to the `ORACLE` table as well.

Find the POIs that are in-path to most active trips: Here we use the `ACTIVE_TRIPS` table since we want to find the most common set of POIs to a potentially large table of trips. Such a query denotes an analysis OLAP style of the query. Here we join the `ACTIVE_TRIPS` table to obtain the list of POIs and then join it with the `POI` table to obtain the list of POIs and only retain those POIs that are of type "VACCINATION-CLINIC."

```
SELECT poi_id, count(*) as num_in_path
FROM POI JOIN ORACLE using (poi_id)
WHERE Z_AB IN
(SELECT Z4_st FROM ACTIVE_TRIPS)
AND POI.Type = ''VACCINATION-CLINIC''
GROUP BY poi_id
ORDER BY 2 DESC;
```

As one can see, a fairly complex query can be reduced to a few lines of SQL using the in-path oracle which speaks to the real power of this abstraction.

6. Experimental Results

The experiments were conducted on a GPU consisting of an AMD Ryzen Threadripper 3960 CPU, which has multi-processing power. It has 48 threads and 24 cores, having 2.2 GHz clock speed, 128 GB RAM, 4 TB HDD, and 512 KB cache memory. The implementation of proposed algorithms is done using Python programming language, version 3.10, running in 64-bit Ubuntu Linux operating system, version 20.04.3, Kernel version 5.4.0-125.

Dataset: The road network dataset was obtained from the 9th DIMACS Implementation Challenge website [83]. We used two datasets in our evaluation, Washington, D.C. road network with 9559 nodes and 12,304 edges and San Francisco dataset which included 321,270 nodes and 800,172 edges. Each directed edge's weight represents the road network travel distances between two nodes. The spatial position of each node was also available which was normalised into a [0, 1) grid box.

Comparison Techniques: Our primary evaluation consists of dual Dijkstra baseline algorithm and comparing it against the proposed in-path oracle. In particular, we choose a source and destination node at random and ask both dual Dijkstra algorithm and in-path oracle to determine which of the POIs are in-path to the shortest path. The parameters that we change to examine how the algorithms perform under diverse settings are listed in Table 1. Each data point was repeated 100 times by randomly chosen sources and destinations for each time and then averaged across all execution runs. The POIs used in the experiments were randomly chosen by uniformly sampling the nodes in the road network. The sampling rate of the POI refers to the rate of choosing POIs among the nodes of the road network. For instance, 0.001 means that we roughly choose 1 node from a set of 1000 nodes. Recall that ϵ refers to the detour tolerance limit which is varied as a fraction over the shortest distance between the source and destination. We vary the epsilon value from 0.1 (i.e., 10%) to 5 (500%).

Table 1. Parameter settings used in experimental evaluation.

Parameters	Values
POIs sampling rate	0.1 to 0.001
ϵ	0.1 to 5

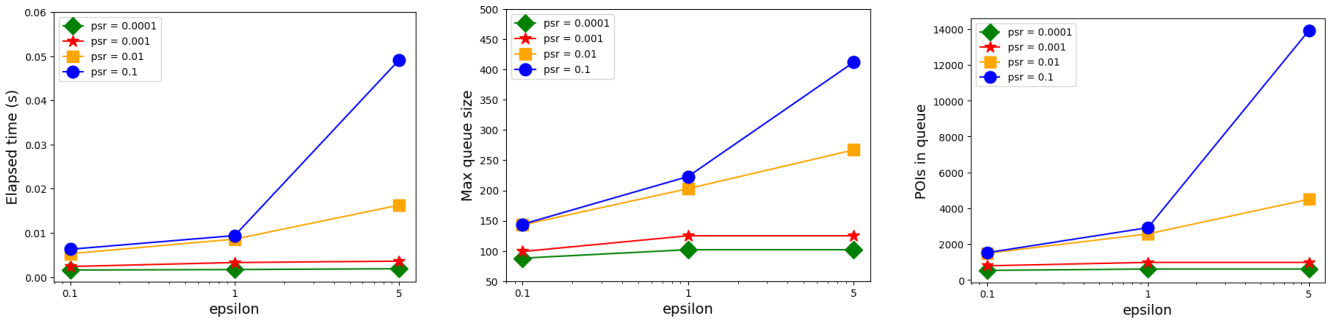
Methodology: We collected the following metrics for each experiment: (1) the elapsed time spent waiting for the program to deliver its outputs (measured in seconds), and (2) the throughput of the algorithm using a single machine. For the dual Dijkstra approach, we additionally report the (3) largest size of the priority queue and (4) the number of POIs added to the priority queue as a way of quantifying how much of the road network had to be traversed by the algorithm. Note that among these experiments, the throughput approach provides the best indicator of the relative performance of the experiments. At the end of the day when these algorithms are deployed in production, the key question is how much machine resources are needed to support the incoming workload. Here, as one can see later, the in-path oracle is a much superior solution compared to the dual Dijkstra baseline approach.

6.1. Baseline Approach

In this section, we examine the performance of a general in-path query variant where the driver is travelling between a source and a destination. The experimental results are on the dual Dijkstra algorithm that we developed earlier, which serves as a credible variant for the in-path oracle approach.

6.1.1. Varying Detour Limit

The plot shown in Figure 3a demonstrates that the running time increases with the detour tolerance limit. It can be seen from the graphs that, as we increase the detour tolerance limit, the running time also increases. Similarly, the size of the priority queue increases when the detour tolerance limit increases, shown in Figure 3b. Finally, Figure 3c shows that the number of POIs added to the priority queue increases when the detour tolerance limit increases. Note that this result again underscores the observation from earlier that the algorithm is sensitive to detour tolerance limit. The larger the detour tolerance limit we allow, the larger is the search space and consequently slower is the query response time.



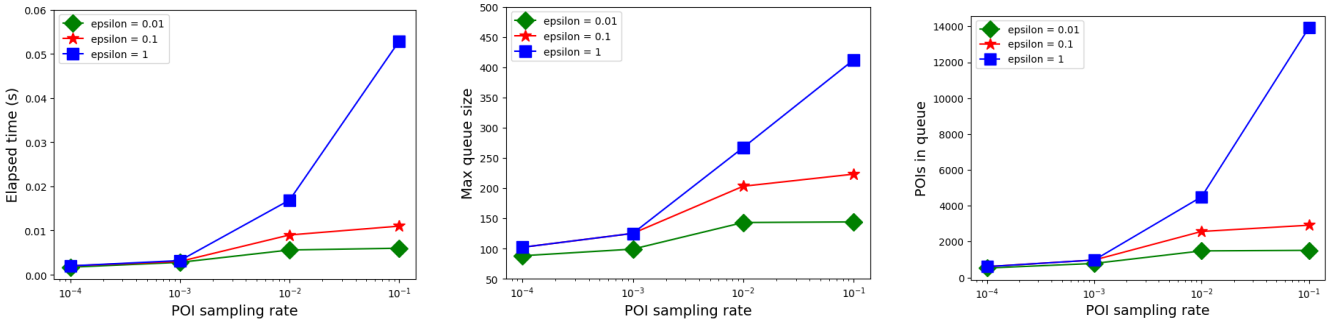
(**a**) ϵ vs. elapsed time (**b**) ϵ vs. maximum priority queue size (**c**) ϵ vs. POIs added to priority queue

Figure 3. The figure shows the effect of varying detour tolerance limits. The figure shows (**a**) elapsed time, (**b**) the maximum size of the priority queue, and (**c**) the number of POIs added into the priority queue as the detour tolerance limit is increased from 0.1 to 5. PSR in the figure legend denotes the POI sampling rate.

6.1.2. Varying POIs Sampling Rates

We vary the POIs sampling rate from 0.0001 to 0.1 while keeping the detour tolerance limit ϵ is fixed at 0.01. As mentioned earlier, we conducted each experiment 100 times,

averaging the results. We track latency, the maximum size of the priority queue, and the number of POIs added to the priority queue during the course of the search process. The plot shown in Figure 4a shows that the running time increases with the POIs sampling rate. It can be seen from the graphs that as we increase the POIs sampling rate, running time also increases. Similarly, the priority queue size increases when the POIs sampling rate increases shown in Figure 4b. Finally, Figure 4c shows that the number of POIs added to the priority queue increases when the POIs sampling rate increases.



(**a**) POIs sampling rate vs. elapsed time

(**b**) POIs sampling rate vs. maximum priority queue size

(**c**) POIs sampling rate vs. number of POIs added into the priority queue

Figure 4. The figure shows the effect of varying POI sampling rates on (**a**) elapsed time, (**b**) the largest size of the priority queue, and (**c**) the number of POIs added into the priority queue.

The key insight here is that as the detour tolerance limit increases, the search space of the algorithm increases and consequently the query latency. Although the sensitivity of the algorithm to the POI sampling rate does not seem very high. This is expected since the query is expected to return all the POIs found, so, there is no opportunity to stop the search process when a few POIs have been found.

6.2. In-Path Oracle

In this section, we examine the performance of oracle size with varying parameter values like detour limits and road network size for a fixed POI.

6.2.1. Varying Road Network Size

The road network used in the experiment was the San Francisco Bay area dataset, consisting of 321,270 road nodes and 800,172 road edges. For different sizes (number of nodes) of the road network, we recorded the size of the oracle. We considered the size of the road network between 1000 and 100,000 and kept the detour tolerance limit fixed at 0.25. Figure 5 illustrates the different sizes of oracles with varying numbers of nodes. One can see that the size of the oracle is linear in the number of nodes which confirms the earlier linear complexity result. This means that, for a road network of 100,000 nodes, the size of the in-path oracle is about 8 MB which is quite modest by today's computing standards. Note that computing the oracle took less than 30 min for all these cases using a single machine running a single processor thread. One can substantially decrease the query execution by using multiple threads or employing multiple machines. Since the algorithm is data parallel, this denotes a fairly trivial extension to our work.

6.2.2. Varying Detour Limit

For this experiment, we use the Washington D.C. road network dataset and vary the detour limits from 0.05 to 5. Figure 6 illustrates the size of the in-path oracle and one can see that the size of the oracle is bell-shaped. In particular, when ϵ is greater than 2 the size of the oracle sharply decreases since at that point many larger block-pairs would start becoming in-path thus there is no need to subdivide these blocks deeper. For instance, at $\epsilon = 5.0$, most of the road network would be in-path, so, the size of the oracle would be

quite small. In the case of the baseline approach, the latency increases with ϵ but in the case of in-path oracles the size of the oracle decreases with ϵ which is a desirable property.

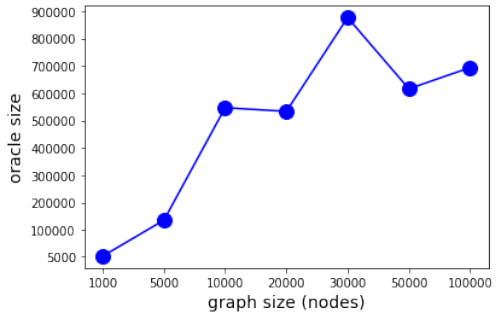


Figure 5. The figure shows the effect of varying the size of the road network with the oracle size for the San Francisco dataset.

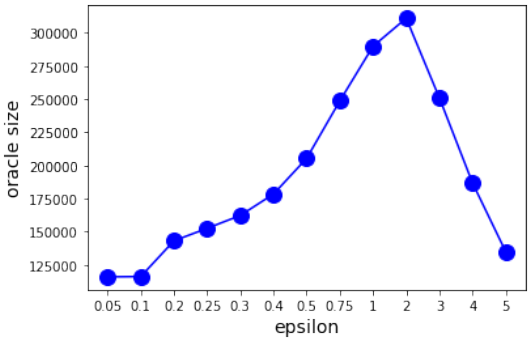


Figure 6. The figure shows the effect of varying detour limits with the oracle size for the Washington, DC dataset.

6.3. In-Path Query Throughput Experiment

In this set of experiments, the throughput of the in-path queries is measured by the varying sampling rates and compared against baseline dual Dijkstra's algorithm which was provided in Section 4. The experiment used 100,000 nodes belonging to the San Francisco dataset. Experiments were done using a single machine with 20 parallel threads corresponding to the number of cores in the machines.

The POIs were randomly chosen and the corresponding in-path oracles were computed. They were inserted into a disk-resident B-tree. The experiments randomly generated source and destination node pairs and looked up the disk-based in-path oracle to find which of the POIs (if any) were in-path to the chosen node pairs. Internally, these queries looked up a disk-resident B-tree with a fixed 3 GB cache. The results shown here correspond to the warm-cache case where the experiments have been running for some period of time. At this point, the in-memory buffers have cached pages of the B-tree in memory. Running the experiment on a cold cache results in high variation in the throughput results, but this case is not interesting from a production standpoint. In our use-case, it was envisioned that the in-path oracle service would be running continuously, and hence the cache is seldom cold.

The number of POIs is chosen similarly to the previous experiments and measured in terms of the sampling rate. The average number of blocks per in-path oracles was about 1 M, which is roughly about 8 MB in memory. Note that the disk size of these oracles are considerably smaller since B-trees encode and compress the data and the resulting representation is an order of magnitude smaller on disk.

We can observe many things from this experimental setup. First of all, the throughput from the dual Dijkstra approach was about 100–200 in-path queries/second for most of the POI sampling rates. In this case, we ran 20 parallel versions of the setup in Figure 3. In this case, there is no opportunity for work-sharing among the threads, so, we just run them in

separate threads that do not share anything other than the graph data structure. In the case of the in-path oracle experiments, they share the cache and the underlying B-trees. Note that the dual Dijkstra baseline algorithm searches the graph representation at runtime while in-path oracle looks up the precomputed representation, so, is expected to be much faster.

In particular, Figure 7 shows the throughput results as a function of the POI sampling rate. One can see that we get more than one million in-path queries per second. In other words, we can answer queries four orders of magnitude faster than the existing baseline algorithm.

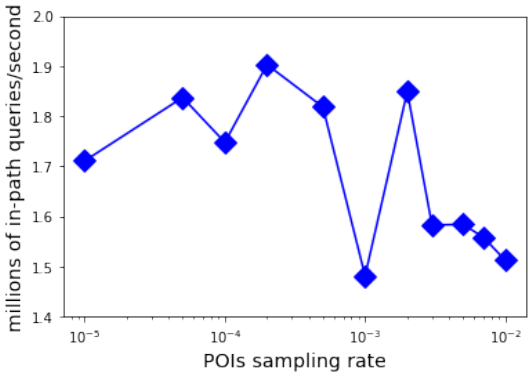


Figure 7. The figure shows the effect of varying the sampling rate on the throughput obtained from in-path oracles. Note that the dual Dijkstra variant has a throughput of ≈200 queries/s and is not shown in the graph.

This is a very powerful result that shows that we can use just a few machines to answer in-path queries for very large map services. This means that our in-path oracle technique can also be used in extensive simulations where in-path considerations may be important. This exceptional improvement that we report in this paper is due to the sensible use of precomputation in solving fundamental and basic operations in road networks.

7. Conclusions

We aimed to solve a seemingly simple problem of which POI is in-path to a given source and destination. We demonstrated a new technique that is simple yet effective. We proposed an in-path oracle that is able to answer in-path queries without having to compute the shortest path between the source and destination. Our experimental results showed that one can perform millions of such queries per second on a road network, which are many orders of magnitude faster than an existing strawman approach. Furthermore, our approach can be embedded in a database system, and thus can be used in conjunction with complex queries. A drawback with any oracle technique is that major changes to road networks such as road closures may affect the precomputed results. However, it is possible to incorporate such dynamic changes by determining which block pairs in the in-path oracle have been invalidated by such road network updates. Future work will look at the dynamism of road networks and the applicability of in-path oracles for such use-cases.

Author Contributions: Conceptualization, Hanan Samet; Methodology, Debajyoti Ghosh and Jagan Sankaranarayanan; Software, Debajyoti Ghosh and Jagan Sankaranarayanan; Validation, Hanan Samet; Writing—original draft, Debajyoti Ghosh, Jagan Sankaranarayanan and Kiran Khatter; Writing—review & editing, Debajyoti Ghosh, Jagan Sankaranarayanan, Kiran Khatter, and Hanan Samet; Visualization, Debajyoti Ghosh and Jagan Sankaranarayanan; Supervision, Kiran Khatter and Hanan Samet; Funding acquisition, Hanan Samet & Kiran Khatter. All authors have read and agreed to the published version of the manuscript.

Funding: Hanan Samet's work in this paper was sponsored in part by the NSF under Grants IIS-18-16889, IIS-20-41415, and IIS-21-1445.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest

References

1. Cao, B.; Alarabi, L.; Mokbel, M.F.; Basalamah, A. SHAREK: A scalable dynamic ride sharing system. In Proceedings of the 16th IEEE International Conference on Mobile Data Management, Pittsburgh, PA, USA, 15–18 June 2015; Volume 1, pp. 4–13.
2. Geisberger, R.; Luxen, D.; Neubauer, S.; Sanders, P.; Volker, L. Fast Detour Computation for Ride Sharing. In Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, Liverpool, UK, 9 September 2010; pp. 88–99.
3. Mahin, M.T.; Hashem, T. Activity-aware Ridesharing Group Trip Planning Queries for Flexible POIs. *ACM Trans. Spat. Algorithms Syst.* **2019**, *5*, 1–41. [CrossRef]
4. Yuen, C.F.; Singh, A.P.; Goyal, S.; Ranu, S.; Bagchi, A. Beyond Shortest Paths: Route Recommendations for Ride-sharing. In Proceedings of the The World Wide Web, San Francisco, CA, USA, 13–17 May 2019; pp. 2258–2269.
5. Bao, J.; Zheng, Y.; Mokbel, M.F. Location-based and preference-aware recommendation using sparse geo-social networking data. In Proceedings of the 20th International Conference on Advances in Geographic Information Systems, Redondo Beach, CA, USA, 7–9 November 2012; pp. 199–208.
6. Bao, J.; Zheng, Y. Location-Based Recommendation Systems. In *Encyclopedia of GIS*; Springer: New York, NY, USA, 2017.
7. Bao, J.; Zheng, Y.; Wilkie, D.; Mokbel, M.F. Recommendations in location-based social networks: A survey. *GeoInformatica* **2015**, *19*, 525–565.
8. Huang, H.; Gartner, G.; Krisp, J.; Raubal, M.; de Weghe, N.V. Location based services: Ongoing evolution and research agenda. *J. Locat. Based Serv.* **2018**, *12*, 63–93. [CrossRef]
9. Levandoski, J.J.; Sarwat, M.; Eldawy, A.; Mokbel, M.F. LARS: A Location-Aware Recommender System. In Proceedings of the IEEE International Conference on Data Engineering, Arlington, VA, USA , 1–5 April 2012; pp. 450–461.
10. Sanchez, P.; Bellogin, A. Point-of-Interest Recommender Systems based on Location-Based Social Networks: A Survey from an Experimental Perspective. *ACM Comput. Surv.* **2022**, *1*, 1–36. [CrossRef]
11. Ye, M.; Yin, P.; Lee, W.C. Location recommendation for location-based social networks. In Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, San Jose, CA, USA, 2–5 November 2010; pp. 458–461.
12. Ghosh, D.; Sankaranarayanan, J.; Khatter, K.; Samet, H. Opportunistic Package Delivery as a Service on Road Networks. *Geoinformatica* **2023**. [CrossRef]
13. Ferraro, R.; Aktihanoglu, M. *Location Aware Applications*; Manning Publishers: Shelter Island, NY, USA, 2011.
14. Ilarri, S.; Mena, E.; Illarramendi, A. Location-dependent query processing: Where we are and where we are heading. *ACM Comput. Surv.* **2010**, *42*, 1–67. [CrossRef]
15. Kupper, A. *Location-Based Services Fundamentals and Operation*; Wiley: Hoboken, NJ, USA, 2005.
16. Schiller, J.; Voisard, A. *Location Based Services*, 1st ed.; Elsevier: Amsterdam, The Netherlands, 2004.
17. Johnson, I.; Henderson, J.; Perry, C.; Schoning, J.; Hecht, B.J. Beautiful... but at What Cost?: An Examination of Externalities in Geographic Vehicle Routing. In Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, Maui, HI, USA, 11–15 September 2017; Volume 1, pp. 1–21.
18. Quercia, D.; Schifanella, R.; Aiello, L.M. The shortest path to happiness: Recommending beautiful, quiet, and happy routes in the city. In Proceedings of the 25th ACM Conference on Hypertext and Social Media, Santiago, Chile, 1–4 September 2014; pp. 116–125.
19. Sacharidis, D.; Bouros, P.; Chondrogiannis, T. Finding The Most Preferred Path. In Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Redondo Beach, CA, USA, 7–10 November 2017; pp. 1–10.
20. Zheng, Y.T.; Yan, S.; Zha, Z.J.; Li, S.Y.; Zhou, X.; Chua, T.S.; Jain, R. GPSView: A scenic driving route planner. *ACM Trans. Multimed. Comput. Commun. Appl.* **2013**, *9*, 1–18. [CrossRef]
21. Bayram, V.; Yaman, H. Shelter Location and Evacuation Route Assignment Under Uncertainty: A Benders Decomposition Approach. *Transp. Sci.* **2017**, *52*, 416–436. [CrossRef]
22. Purba, D.S.D.; Kontou, E.; Vogiatzis, C. Evacuation route planning for alternative fuel vehicles. *Transp. Res. Part C Emerg. Technol.* **2022**, *143*, 1038–1082. [CrossRef]
23. Kim, J.; Park, J.; Kim, K.; Kim, M. RnR-SMART: Resilient smart city evacuation plan based on road network reconfiguration in outbreak response. *Sustain. Cities Soc.* **2021**, *75*, 1033–1086. [CrossRef]
24. Liang, B.; Yang, D.; Qin, X.; Tinta, T. A Risk-Averse Shelter Location and Evacuation Routing Assignment Problem in an Uncertain Environment. *Int. J. Environ. Res. Public Health* **2019**, *16*, 4007. [CrossRef]
25. Russo, F.; Rindone, C. Planning in road evacuation: Classification of exogenous activities. In Proceedings of the 17th International Conference on Urban Transport and the Environment, Nanjing, China, 14–17 August 2011; Volume 116, pp. 639–651.
26. Yin, Y.; Zhao, X.; Lv, W. Emergency shelter allocation planning technology for large-scale evacuation based on quantum genetic algorithm. *Transp. Res. Part C Emerg. Technol.* **2022**, *10*, 1098–1124. [CrossRef]
27. Wagistina, S.; Syafitri, D.R.; Lestari, J.S.; Amanatinismi, K.H.; Setiawan, D.; Ramadhani, S. Service Area Network Analysis for Location Planning of Microbusiness and Local Franchise in Urban Area: A Case Study in Malang City, East Java Provence, Indonesia. *Economies* **2022**, *10*, 103. [CrossRef]
28. Chechik, S. Approximate distance oracles with constant query time. In Proceedings of the 46th Annual ACM Symposium on Theory of Computing, New York, NY, USA, 31 May–3 June 2014; pp. 654–663.

29. Sankaranarayanan, J.; Samet, H. Distance oracles for spatial networks. In Proceedings of the 25th IEEE International Conference on Data Engineering, Shanghai, China, 29 March–2 April 2009; pp. 652–663.
30. Sankaranarayanan, J.; Samet, H.; Alborzi, H. Path oracles for spatial networks. *Proc. Very Large Data Bases* **2009**, *2*, 1210–1221. [CrossRef]
31. Thorup, M.; Zwick, U. Approximate Distance Oracles. *J. Assoc. Comput. Mach.* **2005**, *52*, 1–24. [CrossRef]
32. Callahan, P.B. Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications. Ph.D. Thesis, The Johns Hopkins University, Baltimore, MD, USA, 1995.
33. Callahan, P.B.; Kosaraju, S.R. Faster algorithms for some geometric graph problems in higher dimensions. In Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, Austin, TX, USA, 25–27 January 1993; pp. 291–300.
34. Callahan, P.B.; Kosaraju, S.R. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM* **1995**, *42*, 67–90. [CrossRef]
35. Fischer, J.; Peled, S.H. Dynamic well-separated pair decomposition made easy. In Proceedings of the 17th Canadian Conference on Computational Geometry, Windsor, ON, Canada, 10–12 August 2005; pp. 235–238.
36. Park, E.; Mount, D.M. Output-Sensitive Well-Separated Pair Decompositions for Dynamic Point Sets. In Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Orlando, FL, USA, 5–8 November 2013; pp. 354–363.
37. Elmasri, R.; Navathe, S.B. *Fundamentals of Database Systems*, 7th ed.; Pearson: London, UK, 2021.
38. Bast, H.; Delling, D.; Goldberg, A.; Hannemann, M.M.; Pajor, T.; Sanders, P.; Wagner, D.; Werneck, R.F. Route Planning in Transportation Networks. In *Algorithm Engineering: Selected Results and Surveys*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 19–80.
39. Delling, D.; Goldberg, A.V.; Werneck, R.F. Shortest Paths in Road Networks: From Practice to Theory and Back. *Inf. Technol.* **2011**, *53*, 294–301. [CrossRef]
40. Schultes, D. Route Planning in Road Networks. Ph.D. Thesis, Institut fur Theoretische Informatik, Karlsruhe, Germany, 2008.
41. Wu, L.; Xiao, X.; Deng, D.; Cong, G.; Zhu, A.D.; Zhou, S. Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. In Proceedings of the VLDB Endowment, Istanbul, Turkey, 27–31 August 2012; Volume 5, pp. 406–417.
42. Peng, S.; Samet, H. CDO: Extremely High-Throughput Road Distance Computations on City Road Networks. In Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Burlingame, CA, USA, 31 October–3 November 2016; pp. 1–4.
43. Peng, S.; Sankaranarayanan, J.; Samet, H. SPDO: High-throughput road distance computations on Spark using distance oracles. In Proceedings of the 32nd IEEE International Conference on Data Engineering, Helsinki, Finland, 16–20 May 2016; pp. 1239–1250.
44. Peng, S.; Sankaranarayanan, J.; Samet, H. DOS: A Spatial System Offering Extremely High-Throughput Road Distance Computations. In Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, 6–9 November 2018; pp. 199–208.
45. Samet, H.; Sankaranarayanan, J.; Alborzi, H. Scalable network distance browsing in spatial databases. In Proceedings of the ACM SIGMOD Conference, Vancouver, BC, Canada, 10–12 June 2008; pp. 43–54.
46. Sankaranarayanan, J.; Alborzi, H.; Samet, H. Efficient query processing on spatial networks. In Proceedings of the 13th ACM International Symposium on Advances in Geographic Information Systems, Bremen, Germany, 4–5 November 2005; pp. 200–209.
47. Abraham, I.; Delling, D.; Goldberg, A.; Werneck, R.F. A hub-based labeling algorithm for shortest paths in road networks. In Proceedings of the Experimental Algorithms, Crete, Greece, 5–7 May 2011; pp. 230–241.
48. Delling, D.; Sanders, P.; Schultes, D.; Wagner, D. Engineering Route Planning Algorithms. In Proceedings of the Algorithmics of Large and Complex Networks, Design, Analysis, and Simulation, Beijing, China, 17–19 August 2009; pp. 117–139.
49. Geisberger, R.; Sanders, P.; Schultes, D.; Delling, D. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Proceedings of the 7th International Workshop on Experimental Algorithms, Provincetown, MA, USA, 30 May–1 June 2008; pp. 319–333.
50. Abraham, I.; Fiat, A.; Goldberg, A.V.; Werneck, R.F. Highway dimension, shortest paths, and provably efficient algorithms. In Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms, Austin, TX, USA, 17–19 January 2010; pp. 782–793.
51. Arz, J.; Luxen, D.; Sanders, P. Transit Node Routing Reconsidered. In Proceedings of the International Symposium on Experimental Algorithms, Rome, Italy, 5–7 June 2013; pp. 55–66.
52. Bast, H.; Funke, S.; Matijevic, D.; Sanders, P. In Transit to Constant Time Shortest-Path Queries in Road Networks. In Proceedings of the 9th Workshop on Algorithm Engineering and Experiments, Alexandria, VA, USA, 9–10 January 2007; pp. 46–59.
53. Cohen, E.; Halperin, E.; Kaplan, H.; Zwick, U. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* **2003**, *32*, 1338–1355. [CrossRef]
54. Geisberger, R.; Sanders, P.; Schultes, D.; Vetter, C. Exact routing in large road networks using contraction hierarchies. *Transp. Sci.* **2012**, *46*, 388–404. [CrossRef]
55. Yoo, J.S.; Shekhar, S. In-route nearest neighbor queries. *GeoInformatica* **2005**, *9*, 117–137. [CrossRef]
56. Chen, Z.; Shen, H.T.; Zhou, X.; Yu, J.X. Monitoring path nearest neighbor in road networks. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2009; pp. 591–602.
57. Saha, R.; Hashem, T.; Shahriar, T.; Kulik, L. Continuous Obstructed Detour Queries. In Proceedings of the 10th International Conference on Geographic Information Science, Melbourne, Australia, 28–31 August 2018; Volume 114, pp. 1–14.

58. Shang, S.; Deng, K.; Xie, K. Best Point Detour Query in Road Networks. In Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, San Jose, CA, USA, 2–5 November 2010; pp. 71–80.
59. Nutanong, S.; Tanin, E.; Shao, J.; Zhang, R.; Ramamohanarao, K. Continuous Detour Queries in Spatial Networks. *IEEE Trans. Knowl. Data Eng.* **2012**, *24*, 1201–1215. [CrossRef]
60. Hart, P.; Nilsson, N.; Raphad, B. A Formal Basis for the Heuristic determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybernat.* **1968**, *4*, 100–107. [CrossRef]
61. Bellman, R. On a routing problem. *Q. Appl. Math.* **1958**, *16*, 87–90. [CrossRef]
62. Ford, L.R. *Network Flow Theory*; RAND Corporation: Santa Monica, CA, USA, 1956.
63. Moore, E.F. The shortest path through a maze. In Proceedings of the International Symposium on the Theory of Switching, Cambridge, MA, USA, 2–5 April 1959; pp. 285–292.
64. Dial, R.B. Algorithm 360: Shortest-path forest with topological ordering [H]. *Commun. ACM* **1969**, *12*, 632–633. [CrossRef]
65. Beckley, D.A.; Evens, M.W.; Raman, V.K. Multikey retrieval from k-d trees and quad-trees. *Proc. Int. Conf. Manag. Data* **1985**, *14*, 291–301.
66. Berg, M.D.; Kreveld, M.V.; Overmars, M.; Schwarzkopf, O. *Computational Geometry Algorithms and Applications*, 2nd ed.; Springer: Berlin/Heidelberg, Germany, 2000.
67. Finkel, R.; Bentley, J. Quad trees a data structure for retrieval on composite keys. *Acta Inform.* **1974**, *4*, 1–9. [CrossRef]
68. Peled, S.H. *Geometric Approximation Algorithms*; American Mathematical Society: Providence, RI, USA, 2011.
69. Samet, H. The quadtree and related hierarchical data structures. *ACM Comput. Surv.* **1984**, *16*, 187–260. [CrossRef]
70. Samet, H.; Webber, R.E. Storing a Collection of Polygons Using Quadtrees. *ACM Trans. Graph.* **1985**, *4*, 182–222. [CrossRef]
71. Samet, H. An overview of quadtrees, octrees, and related hierarchical data structures. In *Theoretical Foundations of Computer Graphics and CAD*; Earnshaw, R.A., Ed.; Springer: Berlin/Heidelberg, Germany, 1988; Volume 40, pp. 51–68.
72. Samet, H. *Foundations of Multidimensional and Metric Data Structures*; Morgan Kaufmann Series in Data Management Systems; Academic Press: Cambridge, MA, USA, 2006.
73. Sankaranarayanan, J.; Samet, H. Roads belong in databases. *IEEE Data Eng. Bull.* **2010**, *33*, 4–11.
74. Gargantini, I. An effective way to represent quadtrees. *Commun. ACM* **1982**, *25*, 905–910. [CrossRef]
75. Morton, G.M. *A Computer Oriented Geodetic Database and a New Technique in File Sequencing*; Technical Report; IBM Ltd.: Armonk, NY, USA, 1966.
76. Perdacher, M.; Plant, C.; Bohm, C. Improved Data Locality Using Morton-order Curve on the Example of LU Decomposition. In Proceedings of the IEEE International Conference on Big Data, Virtual, 10–13 December 2020; pp. 351–360.
77. Bayer, R.; McCreight, E. Organization and Maintenance of Large Ordered Indices. In Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Houston, TX, USA, 15–16 November 1970; pp. 107–141.
78. Bayer, R. Binary B-Trees for Virtual Memory. In Proceedings of the 1971 ACM-SIGFIDET (Now SIGMOD) Workshop on Data Description, San Diego, CA, USA, 11–12 November 1971; pp. 219–235.
79. Bayer, R. The universal b-tree for multidimensional indexing: General concepts. In Proceedings of the International Conference on Worldwide Computing and Its Applications, Orlando, FL, USA, 11–13 December 1996; pp. 198–209.
80. Comer, D. The Ubiquitous B-Tree. *ACM Comput. Surv.* **1979**, *11*, 123–137. [CrossRef]
81. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*, 3rd ed.; MIT Press: Cambridge, MA, USA, 2009.
82. Peng, S.; Samet, H. Analytical queries on road networks: An experimental evaluation of two system architectures. In Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, 3–6 November 2015; Volume 1, pp. 1–10.
83. 9th DIMACS Implementation Challenge—Shortest Paths. 2006. Available online: http://users.diag.uniroma1.it/challenge9/download.shtml (accessed on 17 May 2023).