DHBW
Duale Hochschule
Baden-Württemberg

Analysis of In-Path Oracles for Road Networks

Bachelor Project

Course of Studies Informatik

University Konstanz

Jakob Sanowski

Submitted on: 01.01.1980
Student ID, Course: 1095786, Bachelorprojekt
Supervisor at Uni KN: Prof. Dr. Sabine Storandt

Abstract

This report examines the *in-path oracle* proposed in the paper "In-Path Oracles for Road Networks" [1] for identifying Points of Interest (POIs) within a bounded detour from the shortest path between a source and destination in a road network. It defines essential concepts like shortest distance, detour, and in-path POIs. The study compares three algorithms: double Dijkstra, parallel dual Dijkstra, and an in-path oracle method that uses precomputed results to improve query times.

The double Dijkstra algorithm runs two separate Dijkstra instances from the source and destination to find detours through POIs. The parallel dual Dijkstra runs these two Dijkstra instances simultaneously. The in-path oracle method leverages spatial coherence in road networks to precompute results, significantly reducing query times.

Experiments were conducted on datasets from OpenStreetMap, specifically Konstanz and San Francisco, with varying detour limits and POI sampling rates. Results show that the in-path oracle method achieves higher throughput compared to the baseline dual Dijkstra, confirming its efficiency for large-scale applications. However, the oracle size was larger than expected, indicating a need for further optimization and proof refinement.

Contents

1 Introduction 1

2 Related Work 1
2.1 Path and Distance Oracles . . . 2
2.2 Node-Importance based Approaches . . . 2

3 Preliminaries 2
3.1 Problem Definition . . . 3

4 Algorithms & Implementation 3
4.1 Double Dijkstra . . . 3
4.2 Parallel Dual Dijkstra . . . 3
4.3 Beer-Path Oracle . . . 4
4.3.1 In-Path Property . . . 5
4.3.2 R*-Tree . . . 10

5 Main development 11
5.1 Baseline Analysis . . . 12
5.1.1 Theoretical Analysis . . . 12
5.1.2 Practical Worst Cases . . . 15
5.2 Improvements . . . 18
5.2.1 Merge . . . 18
5.2.2 Ceter Representant . . . 18

6 Experimental Evaluation 18
6.1 Dataset . . . 19
6.1.1 Comparative Experiments . . . 19
6.1.2 Baseline Approach . . . 19
6.2 In-Path Oracle . . . 19
6.2.1 Varying Detour Limits . . . 19
6.3 Throughput Experiment . . . 20

7 Conclusions and Future Work 21

Bibliography 22

1 Introduction

In graph theory and computer science, the beer-path problem presents a unique challenge that extends traditional shortest path queries by introducing the necessity to traverse specific vertices, known as “beer vertices.” This problem is particularly relevant in scenarios where paths must include certain checkpoints or resources, analogous to visiting a “beer store” in a network of roads. The beer-path oracle [1] is a specialized data structure designed to efficiently answer queries related to beer paths, providing all beer verticies which in-path for any two vertices.

This report delves into the performance of a beer-path oracle, exploring its efficiency, scalability, and practical applications. We begin by outlining the theoretical foundations of the beer-path problem and discussing some problems which arose during analysis. The core of this report focuses on the implementation details of the beer-path oracle, including the algorithms and data structures employed to achieve optimal query times.

We present a comprehensive performance analysis, evaluating the oracle’s response time and memory usage across different types of graphs. Through empirical testing, we analyse the oracle’s ability to handle graphs of different sizes and discuss the trade-offs between oracle size and query time. Furthermore, we compare our beer-path oracle with a double dijkstra approach, underscoring its advantages and potential areas for improvement.

The findings of this report contribute to the ongoing research in graph algorithms and data structures, offering insights into the development of efficient pathfinding techniques under constrained conditions.

2 Related Work

- Smallest detour queries

2.1 Path and Distance Oracles

2.2 Node-Importance based Approaches

3 Preliminaries

In this section we will establish some preliminary concepts and describe the problem itself. Most of the definitions are taken from D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1].

Definition 1

Shortest Distance

Given source s and destination t nodes, $d_N(s,t)$ denotes the shortest distance between s and t. $d_N(s,t)$ is obtained by summing over the edge weights along the shortest path between s and t.

Definition 2

Detour

Given source s and destination t nodes, let $\pi(s,t)$ denote a simple path that is not necessarily the shortest. The detour d_D 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.

Definition 3

Detour Bound

The detours are bounded by a fraction ε such that their total distance does not exceed $\varepsilon * d_N(s,t)$. For example, if $\varepsilon = 0.1$ a detour can be up 10% longer than the shortest path.

Definition 4

In-Path POI

A POI is said to be *in-path* if there exists a detour bounded by ε which passes through said POI.

3.1 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.

4 Algorithms & Implementation

In this section we will look at the algorithms we want to compare in this report. The first algorithm is a double Dijkstra exploring from the start and target towards the POIs. The second algorithm is a parallel version of the dual Dijkstra [1]. The last algorithm uses an in-path oracle [1] for faster query times.

4.1 Double Dijkstra

The double Dijkstra is a Dijkstra variant for finding detours passing through one $p \in P$. We use two separate instances of Dijkstra starting from the start s and end t node respectively. The input for both instances are all POIs from P and t for the instance starting from s. We combine the result of both instances by adding the costs from both instances for every $p \in P$ together. It is important to note for the instance starting from t we traverse the edges backwards.

4.2 Parallel Dual Dijkstra

D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1] proposed the dual Dijkstra algorithm for finding POIs within a specified detour tolerance limit ε which we developed a parallel version of. In order to parallelize the algorithm we run two Dijkstra at the same time starting from the source s and destination t similar to the double Dijkstra.

Algorithm 1 describes the algorithm of both instances. Each instance uses its own a priority queue Q over the distance to its respective start node. Every node n additionally holds the distance to the start and a label which can be accessed with the functions $d(n)$ and $l(n)$.

At the core of this algorithm is the shared data structure VISITED. This data structure holds all nodes visited by both Dijkstra instances together with a label indicating which instance found the node and the distance to the start node s or t respectively. The key

of this algorithm is in Line 11 where we add the two distances together. If this node $n \in P$ we mark it as $\mathbb{POI}$ so it gets added to the result.

Algorithm 1: Dual Dijkstra

Data:
Result:

```
1  while !Q.empty() n ≔ Q.front() d(n) ≤ d_N do
2      if l(n) == POI then
3          result.add()
4          continue
5      end
6      if VISITED(n, l(n)) do
7          continue
8      end
9      if n_r := VISITED(n, l(n).inverse()) do
10         d' ≔ d(n) + d(n_r)
11         n.distance(d')
12         d_N ≔ min(d_N, d' * (1 + ε))
13         if n ∈ P do
14             Q.insert(n.label(POI))
15         end
16     end
17     VISITED.insert(n)
18     for neighbour v_i of n do
19         Q.insert(v_i.label(l(n)))
20     end
21 end
22 return result
```

4.3 Beer-Path Oracle

The beer-path oracle proposed by D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1] aims to reduce query times using precomputed results. It uses the *spatial*

coherence [2] property in road networks which observes similar characteristics for nodes spatially adjacent to each other. Or more precisely the coherence between the shortest paths and distances between nodes and their spatial locations [2], [3]. We know for a set of source nodes A and destination nodes B they might share the same shortest paths if A and B are sufficiently far apart and the nodes contained in A and B are close together. This enables determining if a POI is in-path with respect to this group of nodes opposed to single pairs of nodes.

The focus here is maximizing the throughput where one can answer millions of in-path queries a second using a single machine.

This approach though is not able to find multiple POIs one might want to visit without exceeding the detour bound. It is expected that the user only wants to visit one of the presented POIs. Such examples include coffee shops, restaurants, gas stations, vaccination clinics, etc.

4.3.1 In-Path Property

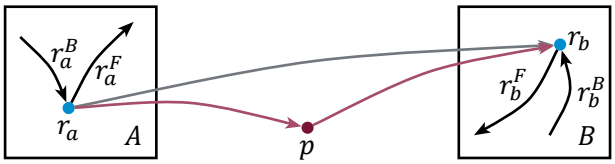

Figure 1 — Whether p is in-path with respect to all sources in A to destinations in B.

In order to define the *in-path* property for a set of source nodes A and a set of destination nodes B these sets are restricted to be inside a bounding box containing all nodes. Let a_r be a randomly chosen representative source node in A and b_r a representative destination node in B. Let p be the POI we want to determine as in-path with respect to the block-pair (A, B) if all shortest-paths from all sources in A to all destinations in B are in-path to p.

We start by defining r_a^F as the forward radius of a given block A denoting the farthest distance from a_r to any node. Similarly, r_a^B defines the backwards radius denotes the farthest distance of any node to a_r. We also define the forward and backwards radius for any block B as r_b^F and r_b^B respectively (see Figure 1). The following lemmas define bounds for the shortest and longest shortest-paths for all shortest-paths from A to B.

Lemma 1

Shortest Shortest Path

Any shortest path between A and B has a length equal to or greater than

$$d_N(a_r, b_r) - \left(r_a^F + r_b^B\right).$$

PROOF: Let s and t be an arbitrary source and destination with $d_N(s,t) < d_N(a_r, b_r)$. Now one can consider the path $a_r \to s \to t \to b_r$. Note that $a_r \to s$ is bounded by r_a^B and $t \to b_r$ is bounded by r_b^F. Following this $d_N(s,t) \geq d_N(a_r, b_r) - \left(r_a^B + r_b^F\right)$ has to hold. If $d_N(s,t) < d_N(a_r, b_r) - \left(r_a^B + r_b^F\right)$ then $d_N(a_r, b_r)$ would not be the shortest distance between a_r and b_r because $d_N(a_r, s) \leq r_a^B$ and $d_N(t, b_r) \leq r_b^F$ which leads to $d_N(a_r, b_r) < d_N(a_r, b_r) - \left(r_a^B + r_b^F\right) + \left(r_a^B + r_b^F\right) = d_N(a_r, b_r)$ which is a contradiction.

■

Lemma 2

Longest Shortest Path

Any shortest path between A and B has a length of at most

$$d_N(a_r, b_r) + \left(r_a^B + r_b^F\right)$$

PROOF: Let s and t be an arbitrary source and destination. Then one can define the following path: $s \to a_r \to b_r \to t$. This path is bound by $d_N(a_r, b_r) + \left(r_a^B + r_b^F\right)$.

■

Lemma 3

In-Path Property

A block-pair (A, B) is in-path if the following condition is satisfied and $d_N(a_r, b_r) - \left(r_a^F + r_b^B\right) > 0$:

$$\frac{r_a^B + d_N(a_r, p) + d_N(p, b_r) + r_b^F}{d_N(a_r, b_r) - \left(r_a^F + r_b^B\right)} - 1 \leq \varepsilon$$

PROOF: For any given node s, t in A, B, respectively, $d_N(s,t)$ is at least $d_N(a_r, b_r) - \left(r_a^F + r_b^B\right)$ (see Lemma 1). Considering the path $s \to a_r \to p \to b_r \to t$ it has a length of at most $r_a^B + d_N(a_r, p) + d_N(p, b_r) + r_b^F$. If p is *in-path* to $a_r \to b_r$ then we get the following inequality in order for all possible paths in A, B to be *in-path*:

$$r_a^B + d_N(a_r, p) + d_N(p, b_r) + r_b^F \leq \left(d_N(a_r, b_r) - \left(r_a^F + r_b^B\right)\right) \cdot (1 + \varepsilon)$$
$$\frac{r_a^B + d_N(a_r, p) + d_N(p, b_r) + r_b^F}{d_N(a_r, b_r) - \left(r_a^F + r_b^B\right)} - 1 \leq \varepsilon$$

■

Note that the condition $d_N(a_r, b_r) - \left(r_a^F + r_b^B\right) > 0$ is omitted by D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1] but is necessary because $d_N(a_r, b_r)$ can be 0 in which case $d_N(a_r, b_r) - \left(r_a^F + r_b^B\right) < 0$ and thus the condition would suddenly be satisfied if $d_N(a_r, b_r)$ is smaller than some specific value. Even $d_N(a_r, b_r) > 0$ would not be enough because $d_N(a_r, b_r) > \left(r_a^F + r_b^B\right)$ still isn't guaranteed.

Lemma 4

Not In-Path Property

A block pair (A, B) is not *in-path* if the following condition is satisfied:

$$\frac{d_N(a_r, p) + d_N(p, b_r) - \left(r_a^B + r_b^F\right)}{d_N(a_r, b_r) + \left(r_a^B + r_b^F\right)} - 1 \geq \varepsilon$$

PROOF: For any given nodes s, t in A, B, respectively, $d_N(s, t)$ is at most $d_N(a_r, b_r) + \left(r_a^B + r_b^F\right)$ (see Lemma 2). Considering the path $s \to a_r \to p \to b_r \to t$ it has a length of at least $d_N(a_r, p) + d_N(p, b_r) - \left(r_a^B + r_b^F\right)$. We get the following inequality in order for all possible paths in A, B to not be *in-path* to p:

$$d_N(a_r, p) + d_N(p, b_r) - \left(r_a^B + r_b^F\right) \geq \left(d_N(a_r, b_r) + \left(r_a^B + r_b^F\right)\right) \cdot (1 + \varepsilon)$$
$$\frac{d_N(a_r, p) + d_N(p, b_r) - \left(r_a^B + r_b^F\right)}{d_N(a_r, b_r) + \left(r_a^B + r_b^F\right)} - 1 \geq \varepsilon$$

■

Lemma 5

In-Path Parent

A block pair (A, B) is *in-path* if all its children are *in-path*

PROOF: For any given nodes s, t in A, B respectively we find a child block pair (A', B') with $s \in A'$ and $t \in B'$. Because all child block pairs of (A, B) are *in-path*, s, t are *in-path* and thus (A, B) has to be *in-path*.

■

Algorithm 2 — In-Path Oracle for a given POI

```
1:  R ← root block of the road network
2:  result ← ∅
3:  Q ← {R, R}
4:  while !Q.empty() do
5:      (A, B) ← Q.pop_front()
6:      s, t ← random node from A, B, respectively
7:      values ← 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 values.in-path() then
9:          result.add((A,B))
10:     end
11:     if values.not-in-path() then
12:         continue
13:     end
14:     Subdivide A and B into 4 children blocks. Discard empty children blocks.
15:     Insert all children blocks into Q
16: end
```

D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1] claim the size of the *in-path* oracle is $O\left(\frac{1}{\varepsilon^2}n\right)$. Their proof references the arguments in [3]. In order to provide some context we will give a overview over these arguments.

Distance Distortion

The distance distortion is the ratio of the network distance to the spatial distance between two vertices. One can define a minimum and maximum distortion γ_L, γ_H for a spatial network such that

$$\gamma_L \leq \frac{d_G(u,v)}{d_S(u,v)} \leq \gamma_H; \gamma_L.\gamma_H > 0.$$

Well-Separated Pair Decomposition

Given a point set A then r denotes the radius of the hypersphere containing all points in A. The *minimum distance* of two point sets A and B is the distance between the hyperspheres containing them. Two sets of points are considered *well-separated* if the *minimum distance* between A and B is at least $s \cdot r$ with $s > 0$. (see Figure 2) s is the *separation factor* and r is the larger radius of the two sets. Such a pair is termed a *well-separated pair* (WSP).

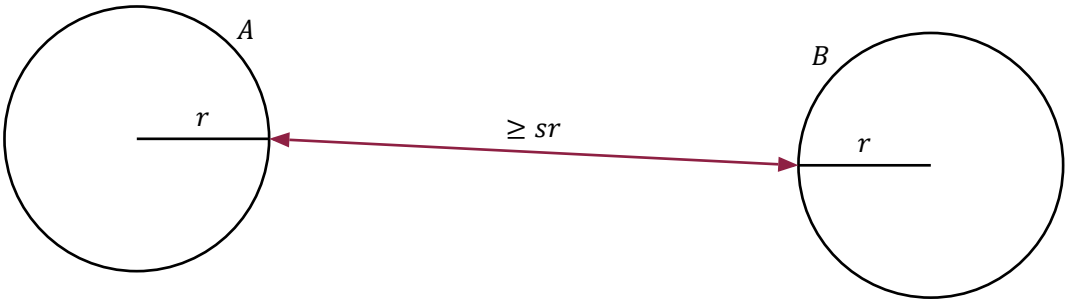

Figure 2 — A and B are well-separated if the distance between them is larger than sr.

A *well-separated pair decomposition* (WSPD) of a point set S is a set of WSPs such that $\forall u, v \in S, u \neq v$, there is exactly one WSP (A, B) with $u \in A, v \in B$. One possible WSPD would be pairs of singleton element subsets $(u, v) \forall u, v \in S, u \neq v$ containing $n \cdot (n - 1)$ pairs. It has been proven one can always construct a WSPD of size $O(s^d n)$ [4].

Such a WSPD of S can be constructed by first constructing a PR quadtree T on S. The decomposition of S into WSPs using T is called a *realization* on T, i.e., the subsets A_i, B_i of S forming a WSP (A_i, B_i) correspond to nodes of T. Starting with the pair (T, T) corresponding to the root of T we check for each pair (A, B) if it is separated with respect to s. If so it is reported as WSP. Otherwise we pair each child of A with each child of B in T and repeat the process until all leafs of T are covered.

Packing Lemma

Considering an arbitrary point set A then a block with side length $2r$ encloses all points in A. The total number of blocks with side length $2r$ which are not *well-separated* from A is bounded by the number of blocks contained within a hypersphere of radius $(2s + 1)r$ centered at A, which contains a maximum of $O(s^d)$ blocks.

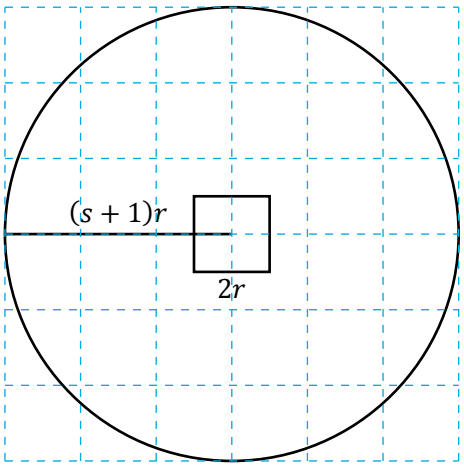

Figure 3 —

WSPD Size

Using the packing lemma we get a size of $O(s^d n)$ for a WSPD since a PR quadtree has $O(n)$ inner nodes and each inner node can produce a maximum of $O(s^d)$ WSPs.

Network Distance WSPD

For a WSPD build using the network distance we can bound r' by

$$r' \leq \gamma_H r.$$

The effective separation factor s' is $s\gamma_H$. Therefore, the size of the WSPD is $O((s)^d n)$ and because γ_H is a constant independent of n.

4.3.2 R*-Tree

In order to get fast query times we used an *R*-Tree* [5] for storing the oracle. The *R*-Tree* is a variant of the *R-Tree* [6] which tries to minimize overlap.

The idea behind *R-Trees* is to group nearby objects into rectangles and in turn store them in a tree similar to a *B-Tree* (see Figure 4). Also like in a *B-Tree* the data is organized into pages of a fixed size. This enables search similarly to a *B-Tree* recursively searching through all nodes which bounding boxes are overlapping with the search area.

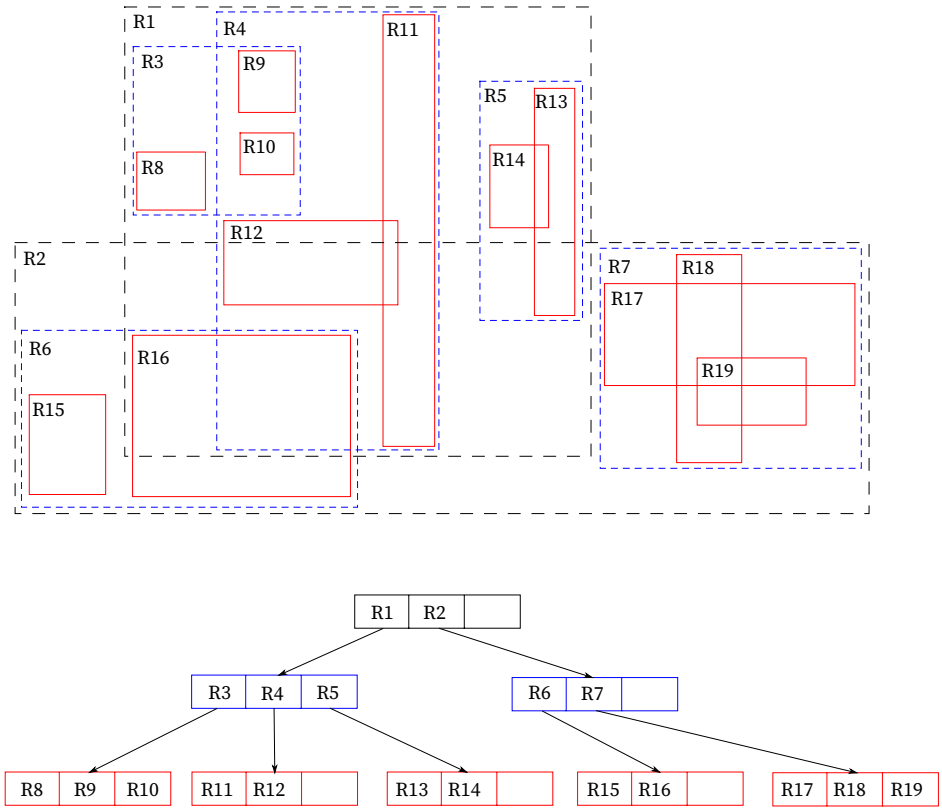

Figure 4 — *R-Tree* for 2D rectangles with a page size of 3

The performance of an *R-Tree* greatly depends on the overlap of the bounding boxes in the tree. Generally less overlap leads to better performance. For this reason the insertion strategy is crucial for achieving good performance. *R*-Trees* try to minimize the overlap by employing insertion strategies which take this into account. This improves pruning performance, allowing exclusion of whole pages form search more often. The key for achieving this is based on the observation that *R-Trees* are highly susceptible to the order in which their entries are inserted. For this reason the *R*-Tree* performs reinsertion of entries to “find” a better suited place in the tree.

In the case of a node overflowing a portion of its entries are removed and reinserted into tree. To avoid infinite reinsertion, this may only be performed once per level of the tree.

5 Main development

5.1 Baseline Analysis

The approach presented in Section 4.3 has some shortcomings especially in its space consumption. In this section we will look at some possible reasons for these shortcomings.

The biggest shortcoming of the *in-path* oracle is the space consumption. We found the oracle to be very large even on relatively small instances. Furthermore it was not possible to test instances of similar size to the instances used by [1]. This bakes the question for the cause of the large size of the oracle.

5.1.1 Theoretical Analysis

Definition 5

Radius

Let r be the average of $r_a^F, r_a^B, r_b^F, r_b^B$ such that $4r = r_a^F + r_a^B + r_b^F + r_b^B$.

We can use r to get an upper bound for the average over all the specific radii which should give us an idea how large the block pairs can be in relation to their distance.

Lemma 6

In-Path Radius Upper Bound

With d_D denoting the detour through p for any block pair (A, B) to be *in-path* the average radius is bound by:

$$r \leq \frac{d_N(s,t)\varepsilon - d_D}{4 + 2\varepsilon}$$

PROOF: Using Lemma 3 gives us:

$$\begin{aligned}
\frac{d_N(s,t) + d_D + 2r}{d_N(s,t) - 2r} &\leq 1 + \varepsilon \\
d_N(s,t) + d_D + 2r &\leq (1+\varepsilon)(d_N(s,t) - 2r) \\
4r &\leq d_N(s,t)\varepsilon - 2r\varepsilon - d_D \\
4r + 2r\varepsilon &\leq d_N(s,t)\varepsilon - d_D \\
r(4 + 2\varepsilon) &\leq d_N(s,t)\varepsilon - d_D \\
r &\leq \frac{d_N(s,t)\varepsilon - d_D}{4 + 2\varepsilon}
\end{aligned}$$

■

We can see r can be at most $\frac{1}{4}$ of $d_N(s,t)\varepsilon - d_D$ for a block pair to be *in-path*. This is especially bad for small ε because then $d_N(s,t)\varepsilon$ is small which in turn causes r to be a small fraction of $d_N(s,t)$. Moreover, d_D is subtracted from $d_N(s,t)\varepsilon$ causing r to have to be even smaller or even zero.

Lemma 7

Not In-Path Radius Upper Bound

With d_D denoting the detour through p for any block pair (A,B) to be not *in-path* the average radius is bound by:

$$r \leq \frac{d_D - d_N(s,t)\varepsilon}{4+2\varepsilon}$$

PROOF: Using Lemma 4 gives us:

$$\begin{aligned}
\frac{d_N(s,t) + d_D - 2r}{d_N(s,t) + 2r} &\geq 1+\varepsilon \\
d_N(s,t) + d_D - 2r &\geq (1+\varepsilon)(d_N(s,t) + 2r) \\
4r + 2r\varepsilon &\leq d_D - d_N(s,t)\varepsilon \\
r &\leq \frac{d_D - d_N(s,t)\varepsilon}{4+2\varepsilon}
\end{aligned}$$

■

For a block pair to be *not-in-path* r is primarily bound by d_D which makes sense because a large detour increases the difference to the detour limit and thus increases the size a block can have without containing a node which can have a detour within the limit.

Using Lemma 6 and Lemma 7 we can find a bound for $d_N(s,t)$ where it is neither *in-path* nor *not-in-path* or in other words where a block pair (A,B) is not well-separated.

Lemma 8

Not Well-Separated Block

A block pair (A,B) is not well-separated when

$$\frac{-r(4+2\varepsilon) + d_D}{\varepsilon} < d_N(s,t) < \frac{r(4+2\varepsilon) + d_D}{\varepsilon}$$

PROOF: Solving Lemma 6 and Lemma 7 for $d_N(s,t)$ gives us

$$d_N(s,t) \geq \frac{r(4+2\varepsilon) + d_D}{\varepsilon}$$

and

$$d_N(s,t) \leq \frac{-r(4+2\varepsilon)+d_D}{\varepsilon}$$

Using their negations we get:

$$\frac{-r(4+2\varepsilon)+d_D}{\varepsilon} < d_N(s,t) < \frac{r(4+2\varepsilon)+d_D}{\varepsilon}$$

■

We can see for a block pair (A,B) to be a WSP is dependent on d_D. This poses a problem because we can no longer use the spacial coherence argument like D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1] suggest. Figure 5 shows how only the relation to the *POI* is relevant for a block pair to be a WSP. It is not possible anymore to define a hypersphere around a block which contains all blocks not well-separated from it so the packing lemma does not apply anymore. We therefore can not get an upper bound for the total number of blocks which are not well-separated from any given block and thus cannot guarantee the size of the oracle to be $O\left(\left(\frac{1}{\varepsilon}\right)^d n\right)$.

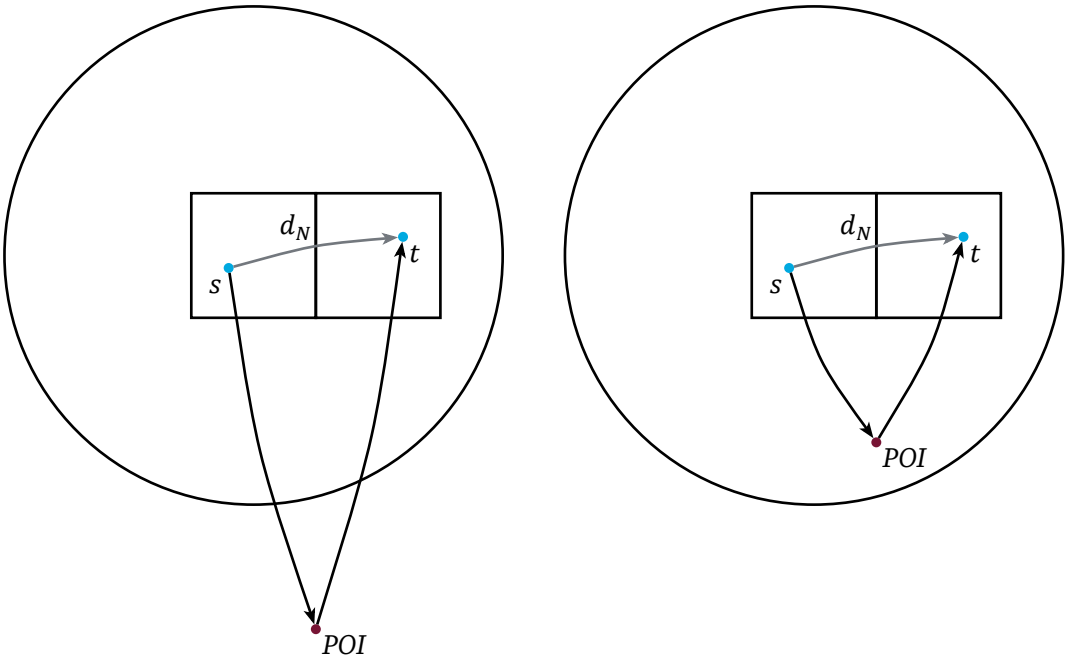

Figure 5 — Depending on the *POI*, d_D can vary for the same block pair and thus be a WSP or not.

5.1.2 Practical Worst Cases

In order to get a better understanding of the performance of Algorithm 2 we build a tool to visualize the results produced by the algorithm. It enables us to look at the concrete values for any block pair as well as the paths leading to these values (see Figure 6). The tool also allows us to have a look at intermediate results occurring during the execution of the algorithm. We could identify multiple cases proofing to be unfavorably for the algorithm.

Figure 6 — A block pair is visualized in pink. The green dots show the representant of the block. The yellow dot shows the POI associated with the block pair. The shortest path is green. The detour is the red path. The blue paths are the radii of the blocks.

Road networks often contain nodes which are very close in euclidean space but have a relatively high road network distance. (see) This case is very common on the border between different suburbs because they are often self contained networks with only one or two access roads with no roads connecting the suburbs. Another reason can be some kind of obstacle having to go around.

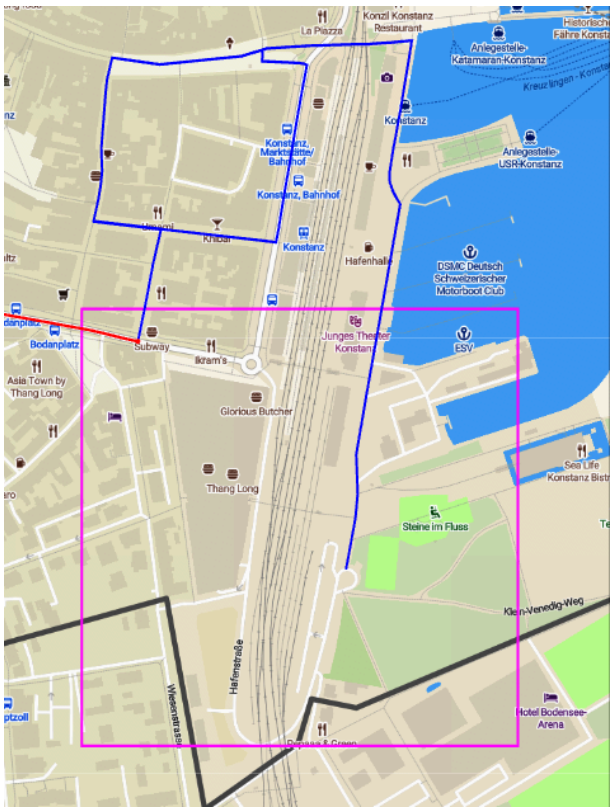

Figure 7 — In order to reach the point on the other side of the train station, a relatively large detour is taken compared to the euclidean distance.

One-Way Streets

One-Way streets tend cause larger radii and thus the blocks to be smaller. As we can see in Figure 8 to reach some nodes inside the block we have to take a significantly longer route due to one-way street. This has the effect of the radii being very long in relation to the size of the block. Furthermore it can require blocks to be split until only one node is contained in a block because we always have to take the long route to reach other nodes on the one-way street.

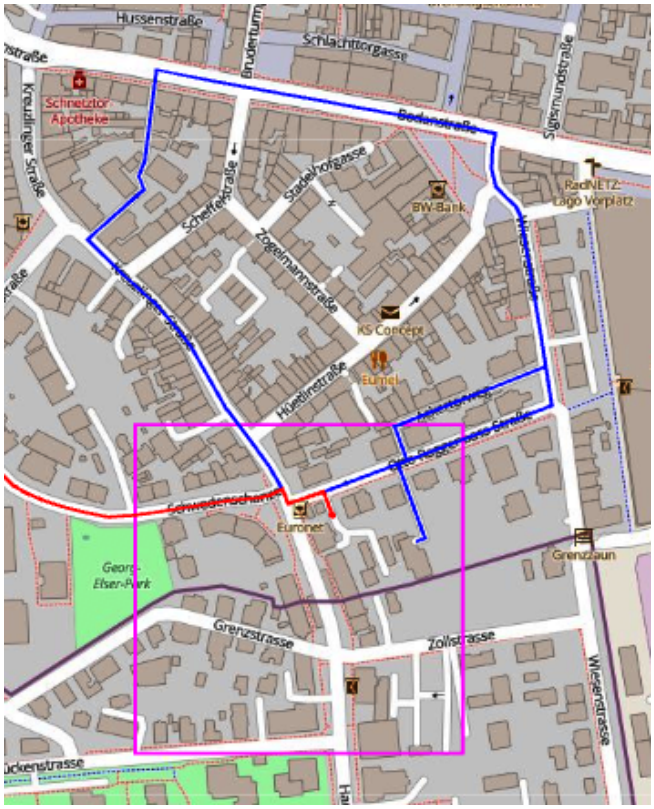

Figure 8 — One-Way streets increase the radii (blue) because having to go around

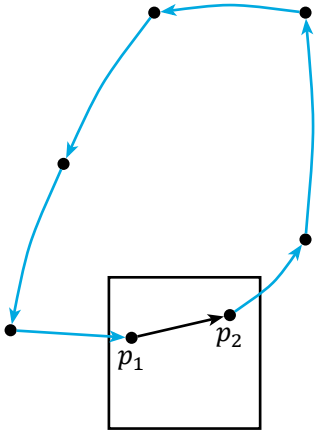

Figure 9 — To get from p_2 to p_1 one has to take the long way around

Figure 9 illustrates this problem. When p_2 is the represent for this block we have to take a really long route to reach p_1.

No Spatial Coherence

5.2 Improvements

5.2.1 Merge

5.2.2 Ceter Representant

- Merge Algorithm

Algorithm 3 — Merged In-Path Oracle for a given POI

```
1:  R ← root block of the road network
2:  result ← ∅
3:  process_block((R, R))
4:  function PROCESS_BLOCK((A, B))
5:      s, t ← random node from A, B, respectively
6:      values ← 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
7:      if values.in-path() then
8:          return true
9:      end
10:     if values.not-in-path() then
11:         continue
12:     end
13:     children ← Subdivide A and B into 4 children blocks. Discard empty children
        blocks.
14:     for child in children do
15:         process_block(child)
16:     end
17:     if all children in-path then
18:         Set this block as in-path
19:     end
20:     for child in children do
21:         result.add((A, B))
22:     end
23:     return false
24: end
```

6 Experimental Evaluation

The experiments were performed on an AMD Ryzen 5 5600X with 6 cores and 12 threads at 4.651 GHz and 16 GB of RAM.

6.1 Dataset

The road networks used for evaluation were obtained from OpenStreetMap and sanitized of foot-paths to only include one edge per street. We used two datasets in our evaluation, Konstanz with 2282 nodes and 4377 edges and San Francisco with 95092 nodes and 172256 edges. The weight of each directed edge denotes the travel distance between two nodes. Note that *chains* (or *ways*) are not simplified.

6.1.1 Comparative Experiments

We used the dual Dijkstra as a baseline for comparison similar to D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1]. We also compared against a simple parallel version of the dual Dijkstra. Each data point is sampled at random meaning a source and destination node is chosen randomly. Each query is run 100 times for all approaches and averaged across all runs. Furthermore, we increase the number of queries in order to measure the throughput of the algorithms. The set of POIs is uniformly sampled from the nodes in the road network with a rate. The rate is multiplied with the total number of nodes in order to get the number of sampled nodes.

6.1.2 Baseline Approach

The dual Dijkstra serves as a baseline for the *in-path* oracle. As a query we used the sampled data points consisting of source and destination pairs.

6.2 In-Path Oracle

To measure the performance we examine the size of the oracle with varying the detour limits and road network size as well as the throughput. Unfortunately we could not compute an *in-path oracle* for the San Francisco dataset in reasonable amount of time.

6.2.1 Varying Detour Limits

To measure the impact of the detour limit on the oracle size we varied the detour limit from 0.05 to 5. The test were performed on the Konstanz data set consisting of 2282

nodes and 4377 edges. As we can see in Figure 10 the oracle size is roughly shaped like a bell which makes sense when looking at Lemma 3 and Lemma 4. When ε is very small Lemma 4 is more easily satisfied. Similarly, when ε is very big Lemma 3 is satisfied for bigger blocks. It is important to note D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1] report much smaller sizes for a graph of this size. For a graph with 5000 nodes they report an oracle size of a bit more than 100,000 compared to the 3,010,095 (see Figure 10) we found for a graph with 2248 nodes.

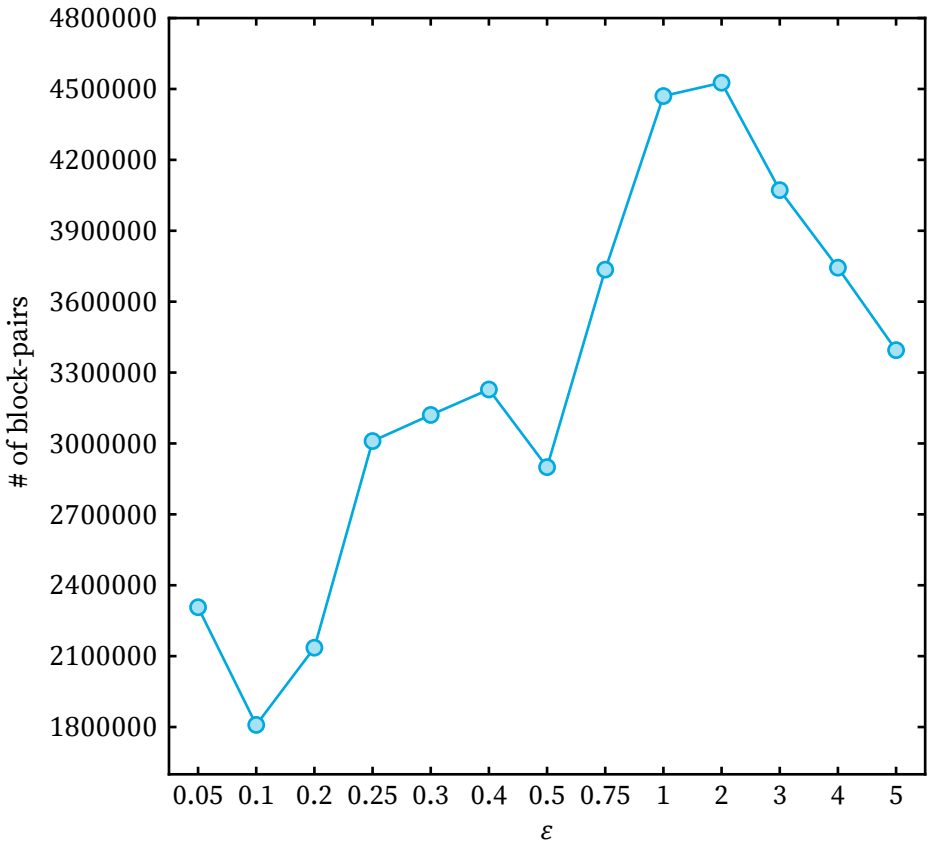

Figure 10 — Size of the oracle for different ε.

6.3 Throughput Experiment

We tested the throughput of *in-path* queries on both the baseline dual Dijkstra and the *in-path* oracle. The experiments were performed on the Konstanz dataset. POIs were randomly sampled with a sampling rate from the dataset which was varied throughout the experiment. We computed the *in-path* oracle for each POI and inserted it into an R*-Tree. Each query was performed on the dual Dijkstra, the parallel dual Dijkstra and the *in-path* oracle. We will ignore the results of the parallel dual Dijkstra moving forward because it always performed worse than the normal dual Dijkstra.

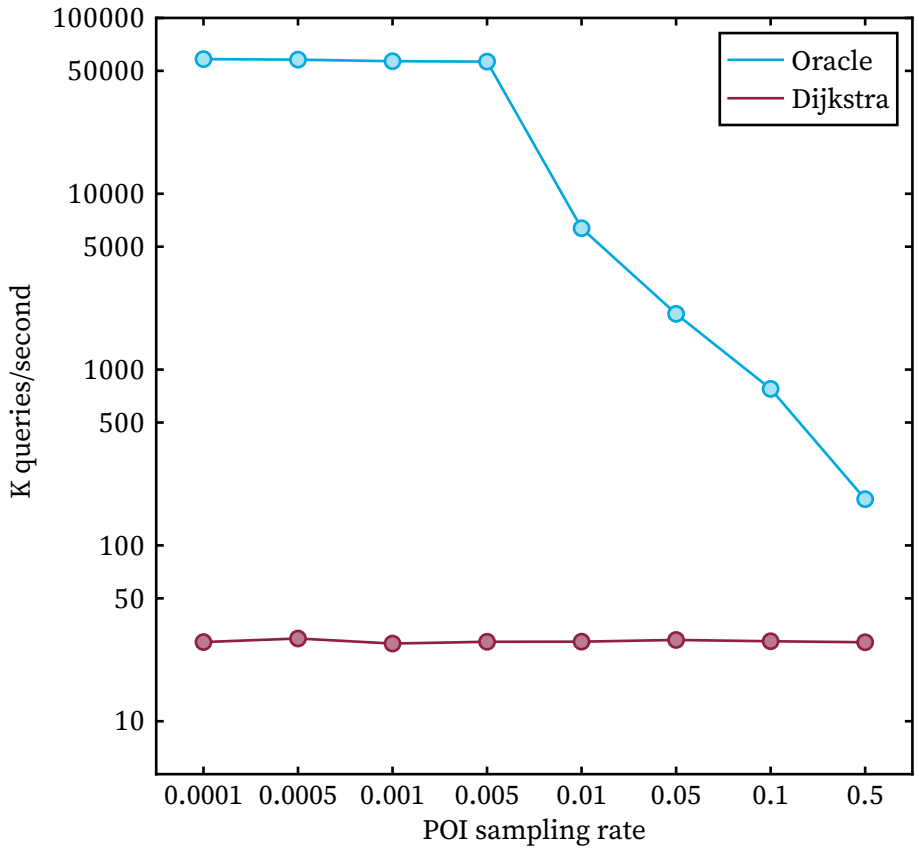

Figure 11 — Throughput of the dual Dijkstra and Oracle for different sampling rates.

We observe a constant throughput of about 28,000 *in-path* queries/second for the dual Dijkstra on most POI sampling rates running on only one single thread. This is due to the search space being dependent on ε and thus not changing for different sampling rates. As expected the *in-path* oracle has a much higher throughput than the dual Dijkstra. Figure 11 clearly shows we get more than 100,000 *in-path* queries per second for all sampling rates. This confirms the findings of D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1].

7 Conclusions and Future Work

We looked at the solution to the *beer-path* problem proposed by D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1] and implemented it in Rust. We could somewhat verify the results with regard to the throughput on small instances. On bigger instances the time to compute the oracle is to long to be practically feasible which stays in contrast to the 30 minutes claimed by D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1]. The oracle size though we find to be bigger by a factor of more than 10 and also exceeds the upper bound they presented which could be why

the compute time is so high. This obviously has an impact on the throughput because of the massive increase in search space (see Figure 11). Because the size of the oracle exceeded the bound presented by D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1] further work should be conducted to provide a concrete proof. Furthermore, we find Lemma 3 to be insufficient. Precisely the term $d_N(a_r, b_r) - \left(r_a^F + r_b^B\right)$ can be less than 0 because $d_N(a_r, b_r) > \left(r_a^F + r_b^B\right)$ is not guaranteed which is why the isolation of ε is not possible. It remains to be seen what the impact of this insufficiency is.

Looking at the findings of this work we can see the potential of the *in-path oracle* [1] though it lacks details to be easily reproducible. Especially with regard to the scalability we could not confirm the claims they made nor find their proofs sufficient.

Bibliography

[1] D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet, "In-Path Oracles for Road Networks," *ISPRS International Journal of Geo-Information*, vol. 12, no. 7, p. 277, Jul. 2023, doi: 10.3390/ijgi12070277.

[2] J. Sankaranarayanan, H. Alborzi, and H. Samet, "Efficient query processing on spatial networks," in *Proceedings of the 13th Annual ACM International Workshop on Geographic Information Systems*, in GIS '05. Bremen, Germany: Association for Computing Machinery, 2005, pp. 200–209. doi: 10.1145/1097064.1097093.

[3] J. Sankaranarayanan and H. Samet, "Distance Oracles for Spatial Networks," in *2009 IEEE 25th International Conference on Data Engineering*, IEEE, Mar. 2009, pp. 652–663. doi: 10.1109/icde.2009.53.

[4] P. B. Callahan, *Dealing with higher dimensions: the well-separated pair decomposition and its applications*. The Johns Hopkins University, 1995.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, in SIGMOD 90. ACM, May 1990. doi: 10.1145/93597.98741.

[6] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84*, in SIGMOD '84. ACM Press, 1984, p. 47. doi: 10.1145/602259.602266.