The shortest path to happiness: Efficiently finding beer-paths using Oracles

Bachelor Thesis

Course of Studies Informatik

University Konstanz

Jakob Sanowski

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

Abstract

This thesis analyzes the concept of the in-path oracle introduced 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

In graph theory and computer science, the problem of identifying paths that require visiting specific vertices, referred to as 'beer vertices,' presents a unique challenge. 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 vertices which are in-path for any two vertices.

This thesis 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 thesis focuses on the theoretical analysis of the oracle outlining the problems and shortcomings it has.

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 thesis 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

The *beer-path* problem is closely related to shortest path finding on road networks. Prior work in this area can be broadly categorized into oracle-based techniques, which use precomputation to answer queries quickly, node-importance-based methods, which leverage the graphical structure of the network, and on-the-way search approaches that find POIs with minimal deviation. This section provides an overview of these approaches, drawing from the survey in .

2.1 Path and Distance Oracles

The concept of a distance oracle was formally introduced by [2], which established a theoretical framework for trading off between preprocessing time, space, and query time. Many subsequent approaches have been developed specifically for road networks, leveraging the principle of **spatial coherence**—the observation that spatially adjacent nodes share similar path characteristics.

For example, [3] introduced a path oracle that encodes all n^2 shortest paths in $O(n)$ storage. This was extended by [4] to create approximate distance oracles, also based on spatial coherence. Other notable developments include City Distance Oracles (CDO), SPark and Distance Oracles (SPDO), and the Distance Oracle System (DOS), which are designed for high-throughput queries on large-scale road networks. The *in-path oracle* builds upon these foundations by applying the concept of spatial coherence to determine if groups of paths include a given POI.

2.2 Node-Importance-Based Methods

In contrast to oracle-based techniques that rely on spatial properties, node-importance-based methods focus on the graph structure of the road network . These approaches are based on the observation that certain nodes (e.g., major highway intersections) are more "important" and that most shortest paths will pass through at least one of them. The general strategy is to rank nodes by a measure of importance and then pre-calculate the shortest path distances between these important nodes. This precomputed information is then used to accelerate query processing at runtime. The *in-path oracle* differs significantly from these methods, as it aims to completely precompute away the graph information, relying solely on spatial data to answer queries.

2.3 Detour and On-the-Way POI Queries

Several approaches directly address finding POIs along a route with a minimal detour. [5] introduced the "in-route nearest neighbor query," which finds the POI with the smallest deviation from a given shortest path. Similarly, [6] explored path nearest neighbor queries for dynamic road networks.

Other related problems include obstructed detour queries, where obstacles prevent straight-line navigation [7], best point detour queries, which seek the detour with the lowest cost [8], and finding routes that must include a stopover of a certain type [9].

The work by [1] is considered more fundamental because its oracle succinctly encodes the in-path status of a POI for every possible source-destination pair, allowing these more complex detour queries to be processed efficiently on top of it.

3 Preliminaries

In this section we will establish some preliminary concepts. We begin with the foundational concepts of graph theory and network paths, the describe detours and in-path POIs. Lastly, we will introduce well-separated pairs (WSPs) and well-separated pair decomposition (WSPD).

3.1 Graphs

Graphs represent relationships (*edges*) between objects (*vertices*). They are the foundational structure for the algorithms discussed in this thesis. In our case, graphs will be used to model road networks.

Definition 1

Graph

An (undirected) graph is a tuple $G = (V, E)$, where

- $V \stackrel{\text{def}}{=} \{v_1, ..., v_n\}$ are the vertices.
- $E \subseteq \{(u, v) \mid u, v \in V\}$ are the edges.

We denote $n \stackrel{\text{def}}{=} |V|$ and $m \stackrel{\text{def}}{=} |E|$.

For an edge $e = (u, v)$, we call u and v its endpoints and say they are incident to e (and vice versa). u and v are neighbours in the graph. The set of all neighbours of a vertex is called its neighbourhood and, for a vertex v, we denote it with $\mathcal{N}(v)$.

Definition 2

Network Paths

Given a graph $G = (V, E)$, the sequence of vertices $\Pi_G(s,t) = \langle s = v_i, v_2, ..., t = v_k \rangle$ is called an *s-t-network-path* of length $k-1$ if it connects two vertices s and t and

- For all $v_i \in \Pi_G(s,t), v_i \in V$.
- For all $i \in \{2, ..., k\}, (v_{i-1}, v_i) \in E$. We call $\langle (v_1, v_2), ..., (v_{k-1}, v_k) \rangle$ its *edge sequence*.

Given a cost function, we call $c(\Pi_G)$ its cost (see).

A path is said to be a *shortest s-t*-network-path if its cost is minimal among all *s-t*-paths. We then write Π_G^*.

Note that even though the underlying graph is undirected, paths do specify a direction, i.e., s and t are not necessarily interchangeable. Dijkstra's algorithm [10] is the most commonly used algorithm to find shortest network paths in graphs. We will not discuss its details here, but refer to [11]. When using Fibonacci Heaps [12] as a priority queue, Dijkstra's algorithm computes the shortest path in $\mathcal{O}(n \log n + m)$ time.

The most intuitive and ubiquitous cost function is the Euclidean cost function:

Definition 3

Euclidean Cost

$$c_{\text{euclid}}(p,q) = |\overline{pq}| = \sqrt{(q_x - p_x)^2 + (q_y - p_y)^2}$$

Definition 4

Shortest Network Distance

Given source s and destination t nodes, $d_N(s,t)$ denotes the shortest network distance between s and t, i.e., $c(\Pi_G^*)$.

We define a *detour* as the difference between a path and the shortest path:

Definition 5

Detour

Given source s and destination t nodes, let $\Pi_{G(s,t)}$ 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 $\Pi_G(s,t)$ 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.

We need to define a bounded *detour* as follows:

Definition 6

Detour Bound

A detour is 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 bounded detour can be up 10% longer than the shortest path.

Most crucially we define a POI to be in path when:

Definition 7

In-Path POI

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

3.2 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.

Definition 8

Well-Separated Pair

Two sets of points are considered *well-separated* if the *minimum distance* between A and B is at least $s \cdot r$, where $s > 0$. 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).

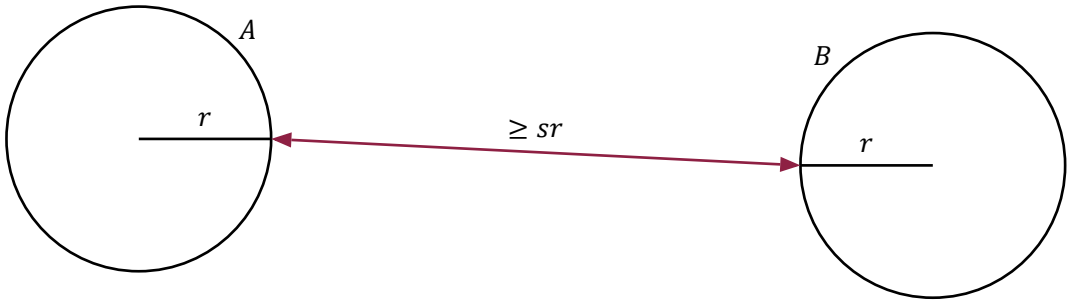

Figure 1 — A and B are well-separated if the distance between them is larger than sr.

Definition 9

Well-Separated Pair Decomposition

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 $O(s^d n)$ [13].

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.

3.3 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 The In-Path Oracle

In a recent paper [1], the authors introduce a method for computing a *in-path* oracle. They adapt the distance oracle proposed by [4] to the *in-path* problem by introducing an *in-path* and *not-in-path* property for block pairs. They claim the precomputation for city-sized road networks is in the tens of minutes with linear memory consumption.

In this section we will look at their methodology and rationale behind it. We will explain their algorithms and try to fill crucial gaps in their description. First, we will outline a baseline method for solving the *in-path* problem. Then, we give some background information necessary to understand the *in-path* oracle. Last, we will discuss the effectiveness of their method and give some possible improvements.

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 10 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

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

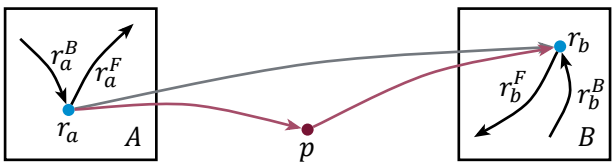

Figure 2 — Whether p is in-path with respect to all sources in A to destinations in B.

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 2). 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 describes the algorithm proposed by D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1] for computing the *in-path* oracle.

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:      a_r, b_r ← random node from A, B, respectively
7:      values ← Compute d_N(a_r, b_r), d_N(a_r, p), d_N(p, b_r), 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
```

The algorithm takes a road network as input and R denotes the root block of a quadtree on the spatial positions of the nodes. Q is a queue holding the block pairs. We initialize Q with the block pair (R, R). The algorithm runs until there are no more block pairs in Q.

In each iteration the front block pair (A, B) is retrieved from Q. The representants a_r and b_r are randomly chosen from A and B respectively. The algorithm proceeds to compute the shortest network distance between the two representants $d_N(a_r, b_r)$ as well as the shortest network distance while passing through p, i.e., $d_N(a_r, p)$ and $d_N(p, b_r)$. It also computes the radii of A and B $r_a^F, r_a^B, r_b^F, r_b^B$. Because computing the radii for a block A requires to compute Dijkstra from a_r to every other node in A we often end up computing Dijkstra multiple times for the same pair of nodes. Using a cache greatly improves computation time for this step. Lastly, the algorithm checks if (A, B) is either *in-path* or *not-in-path*. If (A, B) is *in-path* it is added to the result, if it is *not-in-path* the algorithm just continues to the next iteration. If neither is the case A and B are split into their 4 children blocks and each child of A is paired with each child of B. The resulting block pairs are inserted into Q.

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 [4]. In order to provide some context we will give an overview over these arguments.

4.3.2 Well-Separated Pair Decomposition

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

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

Lemma 6

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 $(2s+1)r$ centred at A, which contains a maximum of $O\left(s^d\right)$ blocks.

PROOF: It is trivial to see that the total number of blocks contained within a hypersphere of radius $(2s+1)r$ is upper bound by $((2s+1)r)^d$. Because r is a constant we get $O\left(s^d\right)$.

■

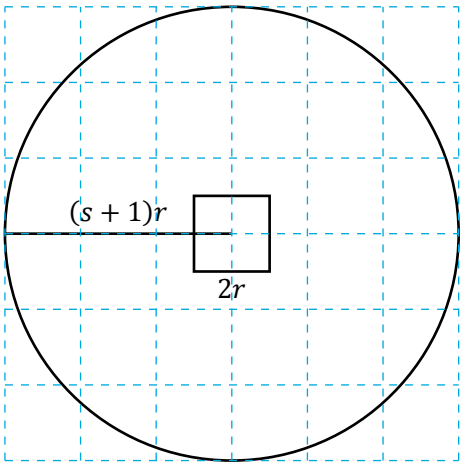

Figure 3 — Visualization of Lemma 6

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

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 $O\big((s)^d n\big)$ and because γ_H is a constant independent of n.

4.3.3 R*-Tree

In order to get fast query times we used an *R*-Tree* [15] for storing the oracle. The *R*-Tree* is a variant of the *R-Tree* [16] 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.

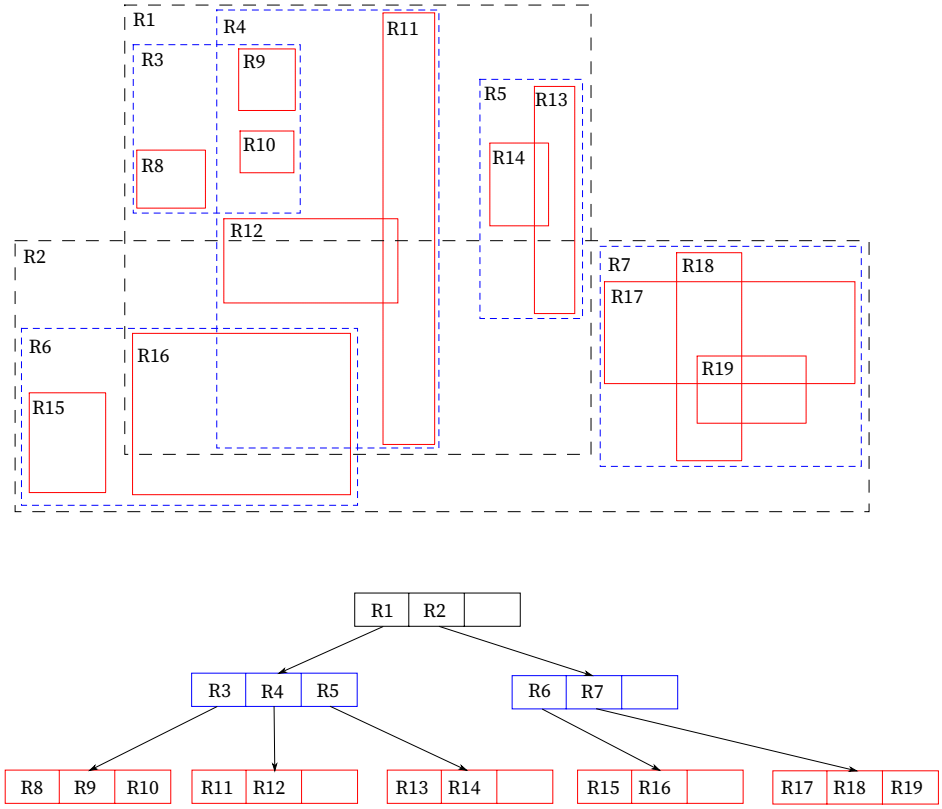

Figure 4 — *R-Tree* for 2D rectangles with a page size of 3

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.

4.4 Limitations

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.

4.4.1 Theoretical Shortcomings

We found the proof for the space complexity to be insufficient for proofing a size of the oracle of $\mathcal{O}\left(\left(\frac{1}{\varepsilon}\right)^2 n\right)$. In the following we will show why this proof does not work.

Definition 10

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 7

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:

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

PROOF: Using Lemma 3 gives us:

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

■

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

Lemma 8

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 7 and Lemma 8 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 9

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 7 and Lemma 8 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 any more to define a hypersphere around a block which contains all blocks not well-separated from it, so the packing lemma does not apply any more. 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 $O\left(\left(\frac{1}{\varepsilon}\right)^d n\right)$.

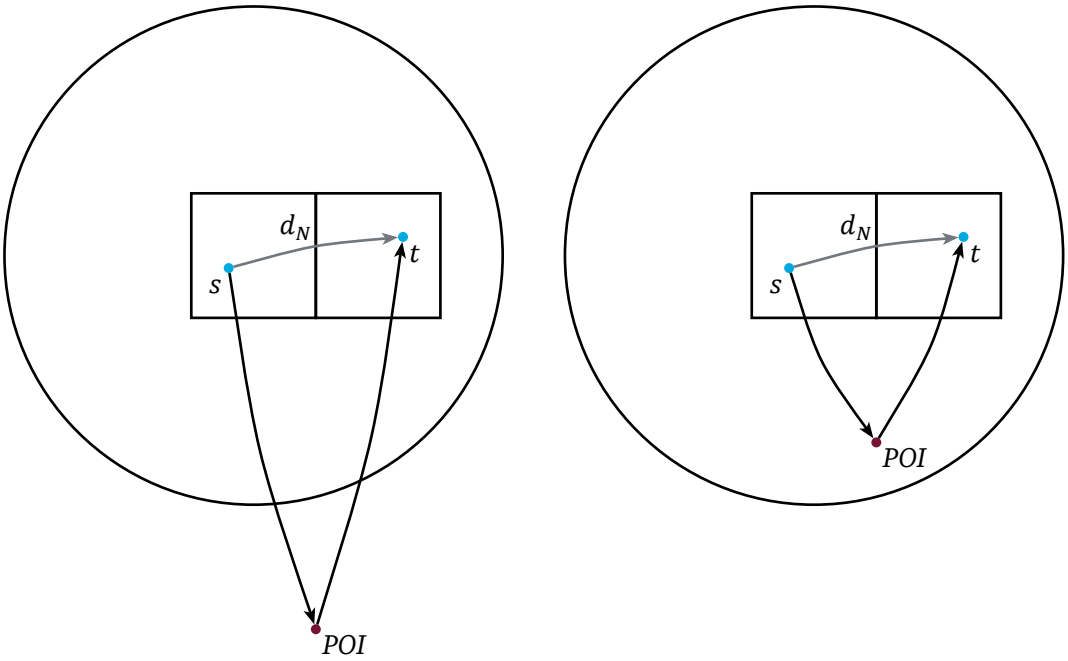

Figure 5 — Depending on the POI, d_D can vary for the same block pair and thus be a WSP or not.

4.4.2 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 unfavourably for the algorithm.

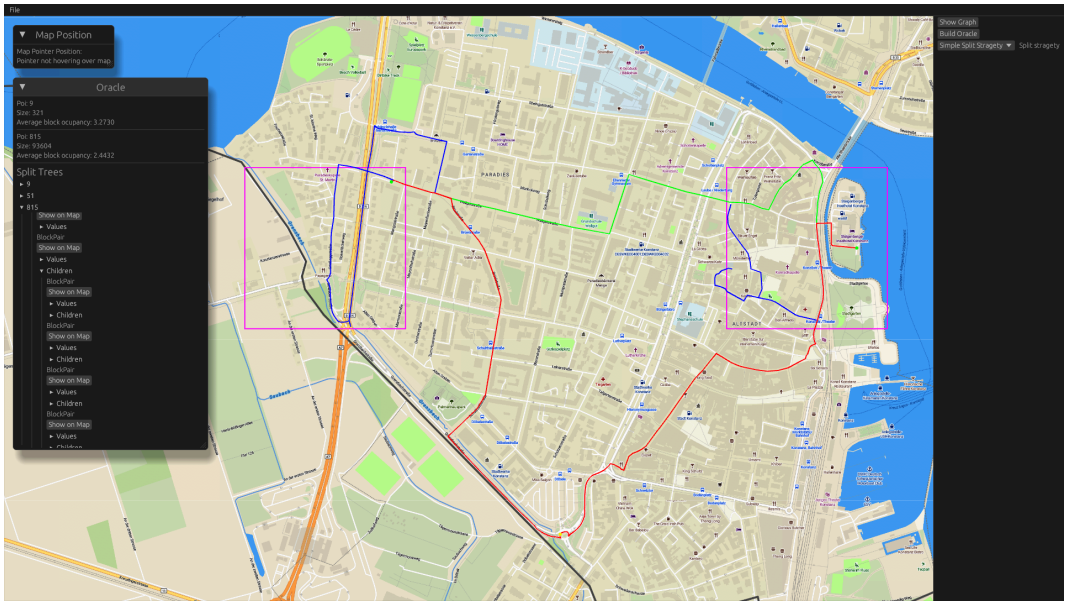

Figure 6 — A block pair is visualized in pink. The green dots show the representant of the block. The yellow dot shows the POI associated with the block pair. The shortest path is green. The detour is the red path. The blue paths are the radii of the blocks.

Road networks regularly contain nodes which are very close in Euclidean space but have a relatively high road network distance. 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.

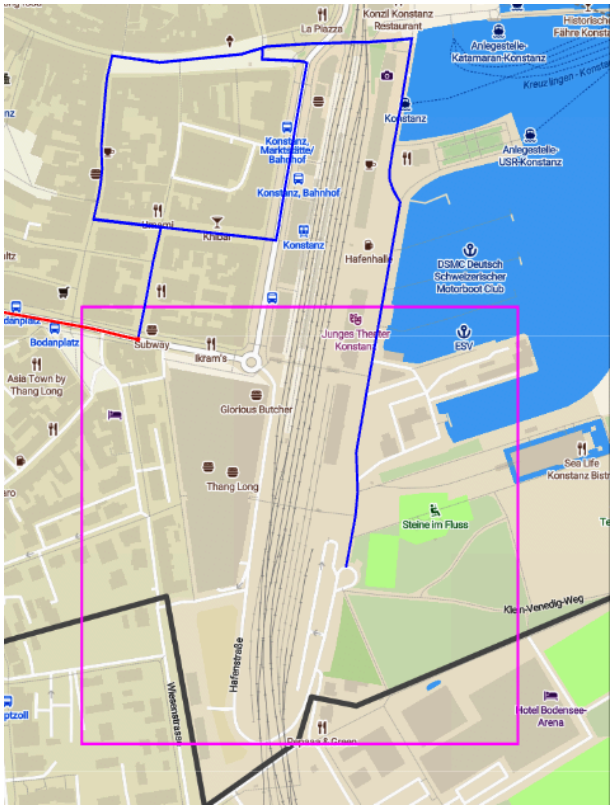

Figure 7 — In order to reach the point on the other side of the train station, a relatively large detour is taken compared to the euclidean distance.

One-Way streets 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.

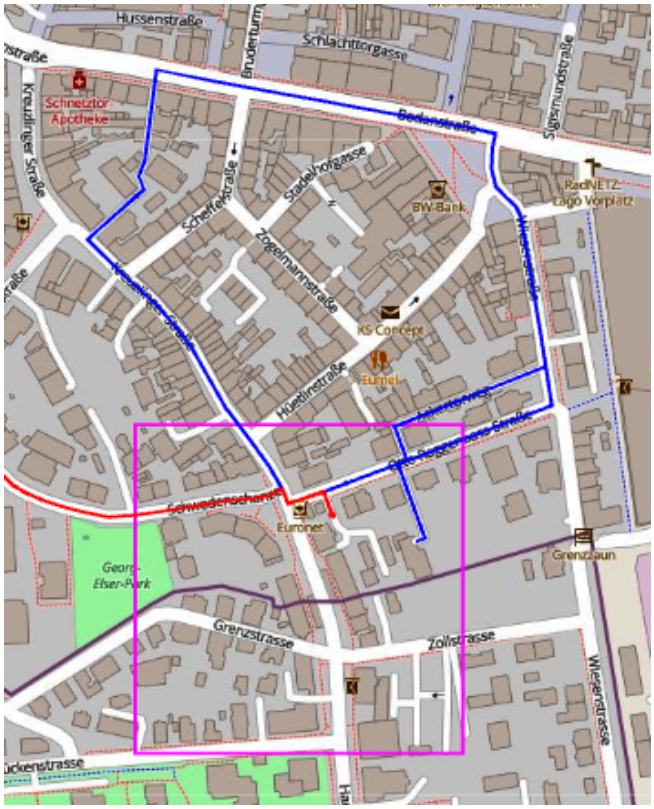

Figure 8 — One-Way streets increase the radii (blue) because having to go around

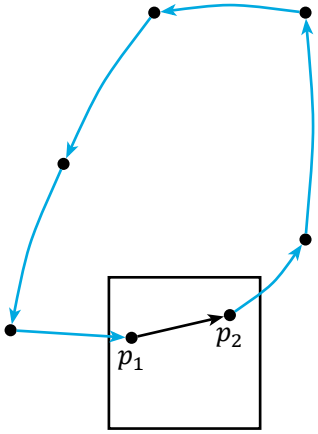

Figure 9 — To get from p_2 to p_1 one has to take the long way around

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

As established in Lemma 9, for a block pair to be well-separated is independent of $d_S(a_r, b_r)$. Figure 10 shows a block pair which is *not-in-path* and thus a WSP. Figure 11 shows the same block pair but here it is not a WSP because the detour is smaller and

therefore it is not possible to decide if the block pair is *in-path* or *not-in-path*. This is the reason it is not possible to use the argument of the packing lemma.

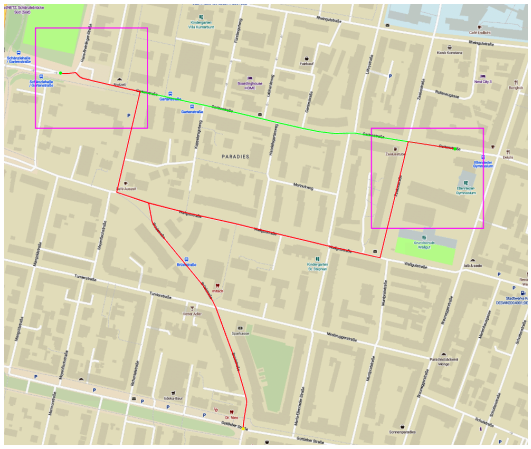

Figure 10 — Not-in-path block pair and thus a WSP.

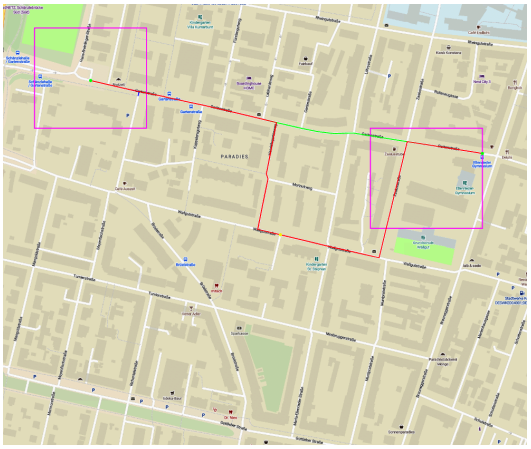

Figure 11 — The same block pair not a WSP for a different POI.

4.5 Improvements

4.5.1 Merge

On real world data we could observe for many block pair all their children either all being *in-path* or *not-in-path*. Therefore, using Lemma 5 we can mark a block pair as *in-path* if all its children are *in-path*.

Algorithm 3 describes an algorithm utilising this observation. The algorithm takes a road network as input and R denotes the root block of a quadtree on the spatial positions of the nodes. The function **process_block_pair()** lies at the heart of the algorithm. It takes a block pair as input and returns if it is *in-path* or not. Note that **process_block_pair()** returns *false* if a block pair is truly *not-in-path* or had to be split. The initial block pair is the root block paired with it self.

Similar to Algorithm 2 we select a_r and b_r at random from A and B respectively and compute the shortest distance between a_r and b_r, the distance of the shortest path passing through p as well as the radii of A and B. If we find (A, B) to be *in-path* or *not-in-path* we simply return *true* or *false* respectively. If neither is the case we split A and B into their 4 children blocks pair each child of A with each child of B. We proceed to call **process_block_pair()** on each resulting block pair.

Using Lemma 5 we can simply report the current block pair as *in-path* by returning *true* if we find all child block pairs to be *in-path* as well. Otherwise we add all child block pairs which are *in-path* to the result and return *false*.

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

```
1:  R ← bounding box of the road network
2:  result ← ∅
3:  process_block_pair((R, R))
4:  function PROCESS_BLOCK_PAIR((A, B))
5:      a_r, b_r ← random node from A, B, respectively
6:      values ← Compute d_n(a_r, b_r), d_N(a_r, p), d_N(p, b_r), 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:         return false
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_pair(child)
16:     end
17:     if all children in-path then
18:         return true
19:     end
20:     for child in children do
21:         if child is in-path then
22:             result.add((A, B))
23:         end
24:     end
25:     return false
26: end
```

4.5.2 Ceter Representant

Another possible improvement would be to try and minimize r. This directly follows from Lemma 9. A lower r decreases the range for $d_N(a_r, b_r)$ and require the block pair to be split and thus increasing the possibility for it to be either *in-path* or *not-in-path*.

r can be minimized by simply choosing the node closest to all other nodes in a block. Because we need to compute the distance between each node in a block to get r^F and r^B it does not cost any extra computing time.

5 Implementation Details

This section describes some of the technical details and challenges in our implementation. Not every aspect is described in detail, but rather we highlight the most interesting aspects.

The implementation accompanying this thesis was written in Rust, a compiled general-purpose programming language. The decision to use Rust was based on its performance, combined with type and memory safety. We use its in-house tool cargo to manage dependencies on third-party libraries[1]. All of our code has been written in safe Rust to avoid any memory leaks or undefined behavior.

Geospatial Primitives

The `geo` library provides data types for geospatial primitives like points, lines, polygons or line strings. Even though we only use the point type for storing the spatial positions of vertices, the widespread use of the `geo` library by other libraries provides a good base for interoperability. Furthermore, `geo` provides basic spatial algorithms like geodesic distance calculations.

5.1 Graph Data Structure

Because we need to be able to work on the graph as well as the spatial relation of the vertices we build a custom data structure combining these two. As a base we used a compressed sparse row [17] graph representation. The positions of the vertices are stored in an *R*-Tree* provided by the `rstar` library and linked to the graph vertices. This enables fast retrieval of all vertices inside a given area.

5.2 File Handling

The road network is provided as *Geojson*. It contains geometry objects with tags describing the type of geometry (e.g. streets). To read a road network into memory

[1] These are called *crates* in the Rust ecosystem

we use the `geo-zero` library. It enables us to filter out unwanted geometry like foot-paths. After creating the graph data structure we find the biggest strongly-connected-component and delete all other components. This is necessary because our algorithms do not handle the case of two unconnected vertices.

To write the graph and oracle to external memory, and to load it for later invocation of the algorithms, we use the `serde` library. It provides extensive support to serialize and deserialize data from internal memory into a number of file formats. This makes it possible to have to perform the expensive computation of the oracle only once and load it from disk for later analysis.

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 12 the oracle size is roughly bell-curve shaped, 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. We only get similar results when applying the merging step though our results are still slightly higher than D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1]. 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 12) we found for a graph with 2248 nodes.

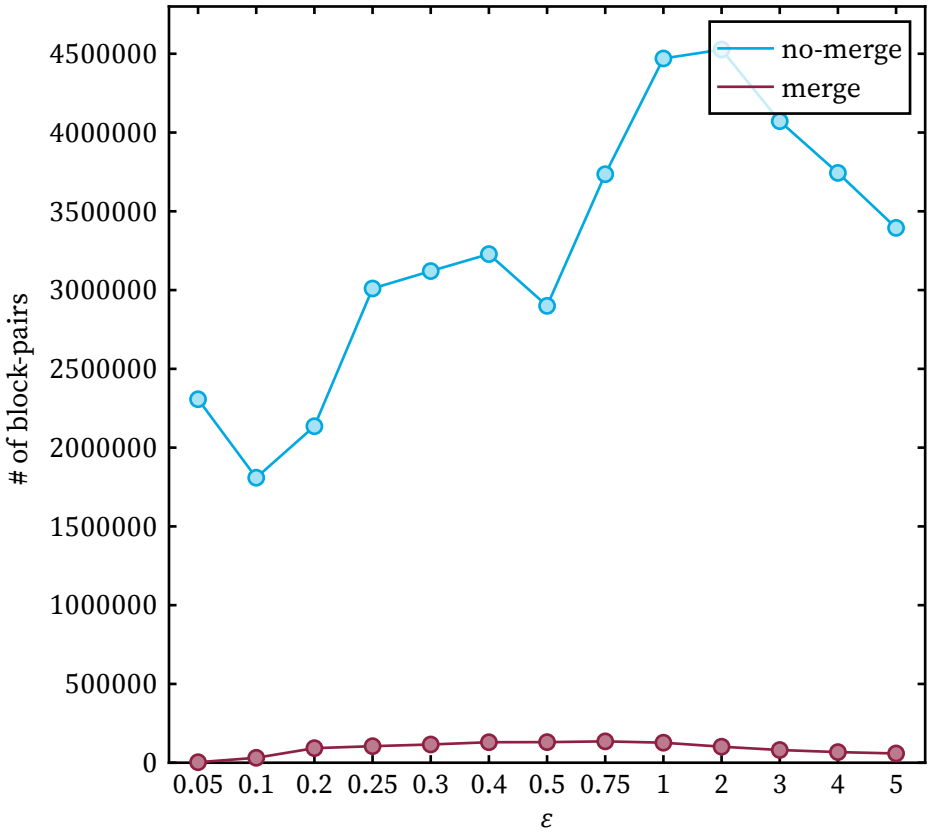

Figure 12 — Size of the oracle for different ε.

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.

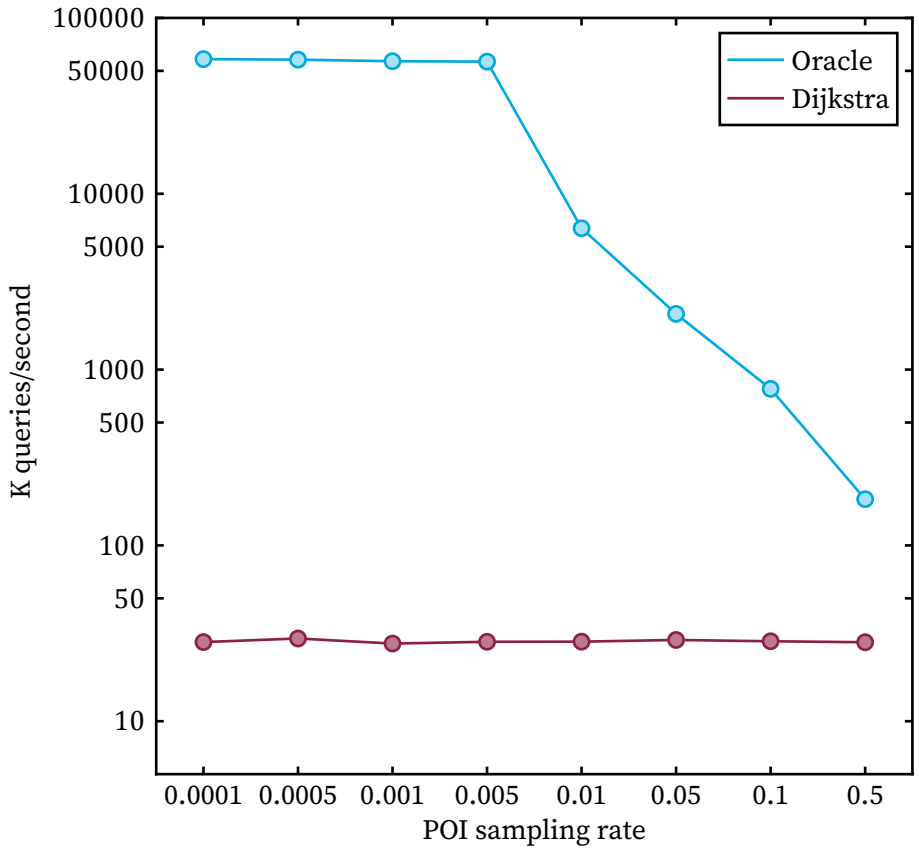

Figure 13 — Throughput of the dual Dijkstra and Oracle for different sampling rates.

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 13 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

In this work we examined the *beer-path* problem and its intricacies and difficulties associated with building an oracle for it. We devoted particularly attention to a method building on WSPDs and its use for distance oracles [4] proposed by D. Ghosh, J. Sankaranarayanan, K. Khatter, and H. Samet [1].

Although the idea of using a similar approach as distance oracles the practicality leaves more to be desired.

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 13). We were able to show, that the proof provided for the size of the oracle is not sufficient. 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.

Another contribution of ours is an improvement to the algorithm creating the oracle. We identified the oracle to be more fine grain than necessary. On many occasions a block pair got split even though all children turned out to be either all *in-path* or *not-in-path*. In order to reduce the required space for storing the oracle we only save the parent block pair. This suggests the *in-path* property can be possibly improved to prevent this split.

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.

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 and find some proofs insufficient.

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] M. Thorup and U. Zwick, “Approximate distance oracles,” *Journal of the ACM*, vol. 52, no. 1, pp. 1–24, Jan. 2005, doi: 10.1145/1044731.1044732.

[3] J. Sankaranarayanan, H. Samet, and H. Alborzi, “Path oracles for spatial networks,” *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1210–1221, Aug. 2009, doi: 10.14778/1687627.1687763.

[4] 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.

[5] J. S. Yoo and S. Shekhar, "In-Route Nearest Neighbor Queries," *GeoInformatica*, vol. 9, no. 2, pp. 117–137, Apr. 2005, doi: 10.1007/s10707-005-6671-1.

[6] Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu, "Monitoring path nearest neighbor in road networks," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, in SIGMOD/PODS '09. ACM, Jun. 2009, pp. 591–602. doi: 10.1145/1559845.1559907.

[7] R. R. Saha, T. Hashem, T. Shahriar, and L. Kulik, "Continuous Obstructed Detour Queries," Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi: 10.4230/LIPICS.GISCIENCE.2018.14.

[8] S. Shang, K. Deng, and K. Xie, "Best point detour query in road networks," in *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, in GIS '10. ACM, Nov. 2010, pp. 71–80. doi: 10.1145/1869790.1869804.

[9] S. Nutanong, E. Tanin, J. Shao, R. Zhang, and R. Kotagiri, "Continuous Detour Queries in Spatial Networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 7, pp. 1201–1215, Jul. 2012, doi: 10.1109/tkde.2011.52.

[10] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," in *Edsger Wybe Dijkstra*, ACM, 2022, pp. 287–290. doi: 10.1145/3544585.3544600.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[12] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987, doi: 10.1145/28869.28874.

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

[14] 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.

[15] 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.

[16] 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.

[17] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, in SPAA 09. ACM, Aug. 2009, pp. 233–244. doi: 10.1145/1583991.1584053.