Beer-Paths

Bachelor Project

Course of Studies Informatik

University Konstanz

Jakob Sanowski

Submitted on: 01.01.1980
Student ID, Course: 7654321, TINF22B2
Supervisor at Uni KN: Prof. Dr. Sabine Storandt

Contents

1 Introduction

In the realm of 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 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, highlighting its significance in various computational contexts such as network routing and logistics. 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 demonstrate the oracle's ability to handle large-scale (lol) graphs and discuss the trade-offs between preprocessing time and query efficiency. 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. By understanding the performance characteristics of the beer-path oracle, we aim to provide a robust framework for future advancements in this field.

2 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 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.

2.1 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.

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.

3 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 algorihm is a parallel version of the dual Dijkstra [1]. The last algorithm uses a in-path oracle [1] for faster query times.

3.1 Double Dijkstra

The double Dijkstra is a Dijkstra variant for finding detours passing through one $p \in P$. We use two seperate 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.

3.2 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 parallelise 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 it's 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: Parallel dual Dijkstra

Data:

Result:

```
while !Q.empty() n := Q.front() d(n) ≤ d_N do
    if l(n) == ℙ𝕆𝕀 then
        result.add()
        continue
    end
    if Visited(n, l(n)) do
        continue
    end
    if n_r := Visited(n, l(n).inverse()) do
        d' := d(n) + d(n_r)
        n.distance(d')
        d_N := min(d_N, d' * (1 + ε))
        if n ∈ P do
            Q.insert(n.label(ℙ𝕆𝕀))
        end
    end
    Visited.insert(n)
    for neighbour v_i of n do
        Q.insert(v_i.label(l(n)))
    end
end
return result
```

3.3 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 percisely the coherence between 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.

3.3.1 In-Path Property

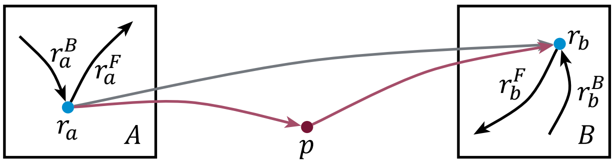

Figure 1 — Weather p is in-path with respect to all sources in A to destinations in B.

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 of 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:

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

PROOF: For any given node s, t in A, B, respectively, $d_N(s, t)$

■

3.3.2 R*-Tree

In order to get fast query times we used an *R*-Tree* [4] for storing the oracle. The *R*-Tree* is a variant of the *R-Tree* [5] 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 2). 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.

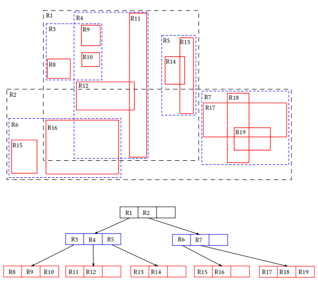

Figure 2 — *R-Tree* for 2D rectangles with a page size of 3

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 Experimental Evaluation

5 Conclusions and Future Work

0 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] 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.

[5] 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.