Project Report

Approximate Distance Oracles: A Randomized Approach

Submitted by:

Kartik Anant Kulkarni (210493)
Siddharth Garg (211031)
Goural Dureja (210393)

Under the Guidance of:

Prof. Surender Baswana

CS648: Randomized Algorithms
Indian Institute of Technology, Kanpur

11th April, 2025

Acknowledgment

We are extremely grateful to our professor, Prof. Surender Baswana, for his unwavering support, expert guidance, and invaluable suggestions throughout the project. His insights played a key role in refining our approach and in the development of our algorithms. In particular, the hints provided during discussions about 3-approximate distance oracles and advice on extending these ideas to a 5-approximate variant were crucial in enabling us to generalize the method to a $(2k-1)$-approximate oracle. We would also like to emphasize that, aside from some initial background reading on implementation techniques, no external assistance was used in the development of this project. Overall, this experience has been highly educational, offering us a practical insight into the challenges and problem-solving required in algorithm design and evaluation—mirroring the real-world complexities faced by researchers in the field.

Contents

1 Problem Statement **3**

2 Introduction **3**

3 Initial Deterministic Attempts **3**
3.1 Method 1: Vertex-Specific Landmark Selection . . . 3
3.2 Method 2: Regular Spacing by Sorted Distances . . . 4
3.3 Trade-Off and Move to Randomization . . . 4

4 3-Approximate Distance Oracle **4**
4.1 Algorithm Overview . . . 4
4.2 Query Processing . . . 4
4.3 Proof of Correctness . . . 5
4.4 Pre-processing . . . 6
4.5 Complexity Analysis for the 3-Approximate Distance Oracle . . . 6
4.6 Space Optimized Variant of the 3-Approximate Distance Oracle . . . 7

5 5-Approximate Distance Oracle **8**
5.1 Construction of the Oracle . . . 8
5.1.1 Initial Attempt . . . 8
5.1.2 Improved Version . . . 9
5.2 Pre-processing . . . 9
5.3 Query Processing . . . 10
5.4 Proof of Correctness . . . 10

6 (2k-1)-Approximate Distance Oracle **11**
6.1 Construction of the $(2k-1)$-Approximate Distance Oracle . . . 12
6.2 Size Analysis of the $(2k-1)$-Approximate Distance Oracle . . . 12
6.3 Query Answering Procedure . . . 13
6.4 Proof of Correctness . . . 14
6.5 Expected Size of Datastructure . . . 14
6.6 Preprocessing for (2k-1)-Approximate Distance Oracles . . . 15
6.7 Final Result . . . 16

7 Implementation **16**

8 Future Work **21**

1 Problem Statement

The aim of this project is to build a space-efficient approximate distance oracle for a non-negative weighted undirected graph $G = (V, E)$ that returns a $(2k-1)$-approximation of the true distance between any two vertices in just $O(k)$ time. The algorithm developed would be practically experimentally on real-world datasets to validate the theoretical results and observe the impact of randomized algorithms in the real-world.

2 Introduction

Consider an undirected non-negative weighted graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. The stretch of a distance oracle is defined as the ratio of the approximate distance reported by the oracle to the true shortest path distance. Mathematically, for any two vertices u and v, the stretch is given by:

$$\text{Stretch}(u, v) = \frac{d_{\text{approx}}(u, v)}{\delta(u, v)},$$

where $d_{\text{approx}}(u, v)$ is the approximate distance returned by the oracle, and $\delta(u, v)$ is the true shortest path distance between u and v as reported by the All-Pairs Shortest Paths Problem (APSP). Our goal is to answer queries with a maximum stretch of $2k-1$ for any two vertices in $O(k)$ time.

Our approach constructs a distance oracle in $O\left(k\, m\, n^{1/k}\right)$ expected time. The oracle occupies $O\left(k\, n^{1+1/k}\right)$ expected space (in the word RAM model) and can answer any $(2k-1)$-approximate distance query in constant $O(k)$ time. We also construct a space efficent version of the algorithm that would further give us a worst case $O\left(k\, n^{1+1/k}\right)$ space. The key idea behind our approach is to trade exactness for efficiency by carefully selecting and storing only a subset of the distance information.

Finally, we implement the algorithm in C++ and analyse the performance on different real-world graphs for various conditions.

3 Initial Deterministic Attempts

Before exploring the randomized approach for selecting landmark vertices, we considered if we can solve the challenge deterministic strategies. Both methods, however, face significant challenges in balancing the number of landmarks with the resulting ball sizes.

3.1 Method 1: Vertex-Specific Landmark Selection

One idea was to assign each vertex w to a landmark $u \in L$ such that the ball $\text{Ball}(u)$ remains as small as possible (e.g., by performing a breadth-first search and selecting the next closest vertex). Although this keeps the ball sizes small, it tends to produce a large number of landmarks:

- **Advantage:** Restricting each ball to a tightly bounded size.
- **Drawback:** In the worst case, there is minimal or no sharing of landmarks among different vertices. Consequently, the landmark set L could become very large, and its size is highly graph-dependent.

3.2 Method 2: Regular Spacing by Sorted Distances

Another attempt was to fix an initial landmark vertex (e.g., a root or "0th node"), compute distances for all other vertices, sort them by distance, and then designate every $\sqrt{n}$th vertex in the sorted list as a landmark:

- **Advantage:** The number of landmarks can be predetermined, offering a fixed cap.
- **Drawback:** Estimating the ball sizes for each landmark becomes difficult. Cross-connections and variations in the underlying graph may cause excessive aggregation in some balls.

3.3 Trade-Off and Move to Randomization

Both of these deterministic techniques highlight a tension between the number of landmark points ($|L|$) and the size of the balls $\text{Ball}(u)$. Overly restrictive strategy on ball size risk exploding the landmark set, while methods ensuring a fixed landmark count can produce unmanageably large balls. This trade-off motivates the use of a *randomized* method, which, in expectation, balances the size of each ball and the total number of landmarks more effectively. Consequently, the randomized approach offers a tunable and provably efficient compromise between these competing objectives. In the following section, we describe one such instantiation: the 3-Approximate Distance Oracle, which demonstrates how approximate distances can be obtained with minimal overhead.

4 3-Approximate Distance Oracle

To improve efficiency over maintaining full APSP data, we sacrifice exactness by approximating distances. Instead of storing the exact distance between every pair of vertices, each vertex keeps distances only to a limited subset of other vertices. Moreover, for a carefully selected group of vertices—referred to as *landmark vertices*—we compute and store the distances from these landmarks to all vertices in the graph. This design enables fast query responses while ensuring that the approximated distances are within a factor of 3 of the true shortest path lengths.

4.1 Algorithm Overview

The oracle is built using a two steps:

- **Landmark Vertices:** A set of landmark vertices is chosen by sampling vertices with probability of a particular vertex being landmark point as p. For each landmark point, the shortest path distance to every other vertex is computed.
- **Selective Storage:** For each vertex, only a small subset of distances to nearby vertices (which are closer than the nearest landmark point) is stored. This significantly reduces the storage overhead compared to storing all pairwise distances.

4.2 Query Processing

When a query is made for the distance between two vertices u and v, the oracle proceeds as follows:

1. **Direct Distance Available:** If the oracle has stored the direct distance between u and v — typically when they lie within each other's local vicinity — it returns this distance immediately without further computation.

2. **Indirect Distance via Landmark:** If the direct distance is not stored, the oracle uses *l(u)* or *l(v)* which is the nearest landmark point to vertex u and v respectively. In this scenario, the approximate distance is computed as:

$$d_{\text{approx}}(u, v) = \delta(u, l) + \delta(l, v),$$

where $\delta(u, l)$ and $\delta(l, v)$ are the precomputed distances from u to l and from l to v respectively. This approximation is guaranteed to be within a factor of 3 of the true shortest path length.

4.3 Proof of Correctness

We now prove that for any two vertices u and v in G, there exists a landmark l such that the approximate distance

$$d_{\text{approx}}(u, v) = \delta(u, l) + \delta(l, v), \quad \text{satisfies} \quad d_{\text{approx}}(u, v) \leq 3\,\delta(u, v).$$

Proof: For each vertex $x \in \{u, v\}$, let $l(x)$ denote the landmark vertex closest to x; that is,

$$l(x) = \arg\min_{l \in L} \delta(x, l).$$

By definition, we then have: $\delta(x, l(x)) \leq \delta(x, y)$ for all $y \in L$.
Thus, we now consider two cases:

- **Case 1**: $l(u) = l(v)$. If both u and v share the same closest landmark, say $l = l(u) = l(v)$, then we can apply the triangle inequality (which is simply, a result of the definition of the shortest distance δ),

$$\delta(u, v) \leq \delta(u, l) + \delta(l, v).$$

Moreover, since l is the closest landmark for both u and v, it follows that

$$\delta(u, l) \leq \delta(u, v) \quad \text{and} \quad \delta(l, v) \leq \delta(u, v).$$

Thus, $d_{\text{approx}}(u, v) = \delta(u, l) + \delta(l, v) \leq \delta(u, v) + \delta(u, v) = 2\delta(u, v)$, which is clearly within the desired factor of 3.

- **Case 2**: $l(u) \neq l(v)$. In this case, we choose one of the landmarks; without loss of generality, let $l = l(u)$. By the triangle inequality,

$$\delta(l(u), v) \leq \delta(l(u), u) + \delta(u, v) = \delta(u, l(u)) + \delta(u, v).$$

Hence, the approximate distance when using $l(u)$ is given by,

$$d_{\text{approx}}(u, v) = \delta(u, l(u)) + \delta(l(u), v) \leq \delta(u, l(u)) + \big(\delta(u, l(u)) + \delta(u, v)\big) = 2\delta(u, l(u)) + \delta(u, v).$$

Since $l(u)$ is the closest landmark to u, we have $\delta(u, l(u)) \leq \delta(u, v)$.
Substituting this bound yields $d_{\text{approx}}(u, v) \leq 2\delta(u, v) + \delta(u, v) = 3\delta(u, v)$.

In both cases, we have shown that $d_{\text{approx}}(u, v) \leq 3\delta(u, v)$. Thus, the distance oracle indeed provides a 3-approximation of the true shortest path distance, completing the proof of correctness. $\square$

4.4 Pre-processing

During pre-processing we need to build necessary data structures for efficient query answering while reducing both computational and storage costs. This phase is divided into three key steps:

1. **Landmark Selection:** A subset $L \subseteq V$ of vertices is chosen to serve as landmark vertices. This selection is performed in a randomized manner: each vertex is independently chosen as a landmark with probability $p = \frac{1}{\sqrt{n}}$. This randomized selection ensures that, in expectation, the number of landmarks is $O(\sqrt{n})$, providing good coverage of the graph while keeping the landmark set small.

2. **Ball Construction, Local Distance Computation, and Hash Table Setup:** For every vertex $u \in V$, let $l(u)$ denote the landmark vertex closest to u. We then define the *ball* of u as
$$B(u) = \{v \in V \mid \delta(u, v) < \delta(u, l(u))\}.$$
This set comprises all vertices that are closer to u than its nearest landmark. Within this ball, the exact distance $d(u, v)$ for all $v \in B(u)$ are computed and stored. To achieve *O(1)* time distance lookups, for each vertex u we will create a hash table to store *Ball(u)*. The size of each hash table is proportional to the size of its respective ball.

3. **Distance Computation for Landmarks:** To compute the shortest distance for each landmark point to all vertices, we use multi-source Dijkstra, which is equivalent to adding a dummy node, connected to all landmarks by a 0 weighted edge, and this dummy node is chosen as the source. These global distances from each landmark allow the oracle to efficiently approximate the distance between vertices that are not within each other's balls.

Note: Here, we are missing a method of efficiently computing a ball for every vertex as the naive Dijkstra algorithm will not achieve the required time complexity. The modification in Dijkstra is described later to achieve the required bound.

4.5 Complexity Analysis for the 3-Approximate Distance Oracle

Space Complexity: To analyze the space required by the 3-approximate distance oracle, we consider the storage needed for each vertex $u \in V$. Let the vertices of $V \setminus \{u\}$ be ordered as $\{v_1, v_2, \ldots, v_{n-1}\}$ in non-decreasing order of their distance from u. Define the set $Ball(u)$ as those vertices in $V \setminus \{u\}$ that are closer to u than the nearest vertex in a randomly sampled set L. In this scheme, each vertex is chosen independently to be part of L with probability p, and here we take $p = 1/\sqrt{n}$.

A vertex v_i will belong to $Ball(u)$ if none of the vertices $v_1, v_2, \ldots, v_{i-1}$ have been selected into L. Thus, the probability that v_i is in $S_u = (1-p)^{i-1}$. By linearity of expectation, the expected size of $Ball(u)$ is
$$\mathbb{E}[|S_u|] = \sum_{i=1}^{n-1} (1-p)^{i-1} \leq \frac{1}{p}.$$

In other words, on average, each vertex u stores distances to no more than $1/p$ vertices in its set S_u.

In addition to the S_u sets, the landmark vertices are used to store distances to all other vertices, and this contributes an extra $O(n^2p)$ term to the space. Thus, the total expected space required by the oracle is given by: $O\left(n^2p + \frac{n}{p}\right)$. Choosing $p = \frac{1}{\sqrt{n}}$ minimizes the total storage, yielding an overall expected space complexity of $O\left(n^{3/2}\right)$.

Time Complexity Analysis for the 3-Approximate Distance Oracle The overall preprocessing time can be divided into two main components: processing the local distance information for each vertex and computing global distances from the landmark vertices. In our scheme, every vertex is independently selected as a landmark with probability $p = \frac{1}{\sqrt{n}}$.

For each vertex u, the local distance processing involves building a set $Ball(u)$ consisting of vertices that are closer to u than its nearest landmark. The expected size of $Ball(u)$ turns out to be approximately $1/p = \sqrt{n}$, leading to an overall local processing time of $O(n^{3/2})$ when summed over all vertices.

In addition, since the expected number of landmarks is $\sqrt{n}$, running a single-source shortest path algorithm from each landmark incurs an aggregate cost of $O\Big(\sqrt{n}(m + n\log n)\Big)$.

Thus, combining both components, the total expected preprocessing time is

$$O\Big(n^{3/2} + \sqrt{n}(m + n\log n)\Big).$$

A detailed proof of these bounds, along with a more comprehensive analysis for the general k-approximate distance oracle, is provided in a later section.

Overall Preprocessing Time: Combining the work done in local processing and the landmark computations, the total expected preprocessing time is

$$O\Big(n^{3/2} + \sqrt{n}(m + n\log n)\Big).$$

This overall time complexity reflects the efficiency gain from not computing full all-pairs shortest paths, while still ensuring that the oracle can later answer queries rapidly.

Query Time: Querying the oracle involves either directly retrieving a stored local distance, if available, or combining the distances from u to a landmark and from the landmark to v. Since the hashing used allows *O(1)* lookups, the query time for a 3-approximate distance oracle is $O(1)$.

4.6 Space Optimized Variant of the 3-Approximate Distance Oracle

A simple modification can transform the randomized preprocessing for the 3-approximate distance oracle into a *space optimised* algorithm, which ensures correctness while maintaining expected polynomial running time and an upper bound on space utilized for storing landmark points and balls. The space utilization is pre-dominated by the balls.

An important point here is that the space utilization may blow up even if some ball gets too large although many balls are of small size. Thus, it is more meaningful to consider sum of ball size for all vertex instead of individual ball sizes. A check can be used to check that the sum of ball sizes is within a bound and repeat the pre-processing only if the size exceeds a threshold.

Bounding the Total Size. Consider all the "balls" stored in the data structure. Let X_u be the size of the ball associated with vertex u, and let $X = \sum_{u \in V} X_u$ denote sum of ball size for all vertex u.
By linearity of expectation:
$\mathbb{E}[X] = \sum_{u \in V} \mathbb{E}[X_u] = n\sqrt{n}$.
Markov's Inequality states that for any constant $c > 0$,

$$\Pr\big(X \geq c\,\mathbb{E}[X]\big) \leq \frac{1}{c}.$$

Hence, with probability at least$1 - \frac{1}{c}$, the total size X is at most $c\,\mathbb{E}[X]$.

Algorithm. Using this fact, we obtain a space optimized variant of preprocessing:

1. **Run the preprocessing step once**: Construct the 3-approximate distance oracle data structure in $O\Big(m \cdot n^{1/2}\Big)$. time as before.
2. **Check the data structure size**: If the total number of stored pairs $X > c\,\mathbb{E}[X]$, *discard* the structure and repeat from Step 1. Otherwise, *accept* the structure and terminate.

Analysis. By Markov's Inequality, the probability that the constructed structure is of acceptable size in any given run is at least $1 - \frac{1}{c}$. Thus, the expected number of runs until acceptance is

$$\frac{1}{1 - \frac{1}{c}} = \frac{c}{c-1}.$$

Consequently, the overall expected preprocessing time is $O\Big(\frac{c}{c-1}\Big) \times O\Big(m \cdot n^{1/2}\Big). = O\Big(m \cdot n^{1/2}\Big)$. since c is a constant. This ensures that we achieve a guaranteed bound on the size of the data structure, while preserving the same $O\Big(m \cdot n^{1/2}\Big)$. expected pre-processing time.

5 5-Approximate Distance Oracle

The aim here is to generalize the idea of 3-approximate distance oracle to get an oracle of smaller size with a trade-off of getting a higher stretch. More precisely, we aim to get a oracle of $O(m \cdot n^{1/3})$ space which returns distance between two vertices $(u,\ v)$ such that $\delta(u, v) \leq d(u, v) \leq 5 \cdot \delta(u, v)$.

5.1 Construction of the Oracle

5.1.1 Initial Attempt

An intuitive generalization of an approximate distance oracle from stretch 3 to 5 arises from attempting to extend the query procedure "step by step" through intermediate nodes. In the $k = 2$ case (stretch 3), the query path between two nodes u and v involves a single nearest landmark node (called as the focus) at level 1, leading to a path of the form:

$$d(u, f_1(u)) + d(f_1(u), v),$$

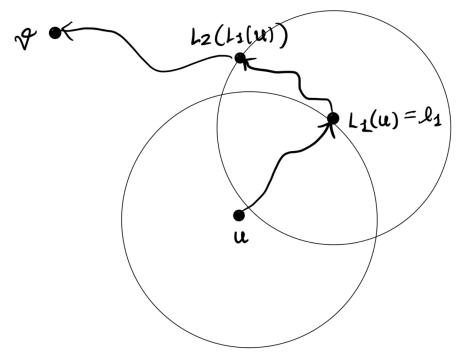


Figure 1: Wrong-Stretch while generalising to 5-Approximate Distance Oracle

with stretch at most 3 if $v \in B_1(u)$.

When extending to $k = 3$ (stretch 5), if we store $Ball(u, v, L_1)$, as the distances of all vertices in V from u upto closest $L_1(u)$, called as the *focus* of u, and $Ball(l_1, V, L_2)$, as the distances of all vertices in V from L_1 upto closest $L_2(l_1)$, then we can try to proceed (See also: Figure 1) in two steps in the worst-case (when v is not in both balls):

$$d(u, L_1(u)) + d(L_1(u), L_2(L_1(u))) + d(L_2(L_1(u)), v),$$

However, it is easy to see that, following earlier derivations of the triangle inequality, if the balls are constructed accordingly, then each segment in the path may individually incur a stretch up to 3, compounding to an overall stretch of 7. The core reason of the mistake is assuming that local triangle inequalities and greedy shortcuts preserve the global stretch bound. Losing our reference of u while querying is leading us to to overapproximation and failure to guarantee the intended stretch of $2k - 1$.

5.1.2 Improved Version

We would like to maintain the information of u and v all way along the computation of the distance between the two vertices. Thus, a better idea would be two to store 2 balls for every vertex (instead of earlier solution), storing vertices part of L_1 and L_2 respectively. This will loosen the bound on $\delta(u, l)$ as compared to $\delta(u, l) \leq \delta(u, v)$ for a far away point. Further, the last level's closest landmark vertex (called as the focus), would then store the distance to all vertices. Now the distance computation between the two vertices must take place in the form of finding closest common landmark vertex between the two, so that we wouldn't have to go to the outermost level.

5.2 Pre-processing

1. From set V of vertices, choose each vertex to be the landmark of first level L_1 with probability $p_1 = 1/n^{1/3}$ independent of other vertices.

2. From set L_1, choose each vertex to be the landmark of second level L_2 with probability $p_2 = 1/n^{1/3}$ independent of other vertices.

3. For each vertex u, compute $Ball_1(u)$ comprising vertices from V such that $\delta(u,v) \leq \delta(u, L_1(u))$ where $L_1(u)$ is the vertex nearest to u from set L_1.

4. For each vertex u, compute $Ball_2(u)$ comprising vertices from L_1 such that $\delta(u,v) \leq \delta(u, L_2(u))$ where $L_2(u)$ is the vertex nearest to u from set L_2.

5. For each vertex u from set L_2, compute distance to all vertices of graph.

Note: Again, here, we are missing a method of efficiently computing a ball for every vertex. The modification in Dijkstra is described later to achieve the required bound.

5.3 Query Processing

To find the distance between two vertices *(u, v)*, we exploit symmetry in sense that we try to find u in ball of vertex v and v in ball of vertex u. We use the following method using conditional conditions from top to bottom for query processing:

1. If $u \in Ball_1(v)$: report $\delta(u,v)$

2. If $v \in Ball_1(u)$: report $\delta(u,v)$

3. If $L_1(u) \in Ball_2(v)$: report $\delta(u, L_1(u)) + \delta(L_1(u), v)$

4. If $L_1(v) \in Ball_2(u)$: report $\delta(u, L_1(v)) + \delta(L_1(v), v)$

5. Else report $\delta(u, L_2(u)) + \delta(L_2(u), v)$

Note: On careful observation, proving case 5 requires information only about case 1 and 4. A pattern of switching between u and v is evident. Thus, the query processing algorithm can be improved further as detailed in the kth level generalization of the oracle (described in the next section), by introducing another level of balls and getting rid of global distances.

5.4 Proof of Correctness

Refer Figure 2. Case 1 and 2 of query processing reports the exact distance, which is within the bound. Note that, here we assumed that the landmark vertices of higher levels are farther, but this can be ensured by hierarchical sampling as described in the construction of the generalised oracle.

- **Case 3:** $\delta(u, L_1(u)) \leq \delta(u,v) + \delta(v, L_1(u))$ [Triangle Inequality]
  $\delta(v, L_1(u)) \leq \delta(u,v)$ {case ③ $\Rightarrow$ case ② does not hold}
  $Thus, \delta(u, L_1(v)) \leq 2\delta(u,v)$
  and $\delta(u, L_1(v)) + \delta(L_1(v), v) \leq 3\delta(u,v)$
  Similar analysis hold for Case 4.

- **Case 5:**
  - Case 1 does not hold : $\delta(u,v) \geq \delta(u, L_1(u))$
  - Case 2 does not hold : $\delta(u,v) \geq \delta(v, L_1(v))$
  - Case 3 does not hold : $\delta(v, L_1(u)) \geq \delta(v, L_2(v))$
  - Case 4 does not hold : $\delta(u, L_1(v)) \geq \delta(u, L_2(u))$

Combining,

$$\begin{aligned}
\delta(u, L_2(u)) + \delta(L_2(u), v) &\leq \delta(u, L_1(v)) + \delta(L_2(u), u) + \delta(u, v) \\
&\leq 2\delta(u, L_1(v)) + \delta(u, v) \\
&\leq 2(\delta(u, v) + \delta(v, L_1(v))) + \delta(u, v) \\
&\leq 2(2\delta(u, v)) + \delta(u, v) \\
&\leq 5\delta(u, v)
\end{aligned}$$

Thus, the maximum stretch in this case is 5.

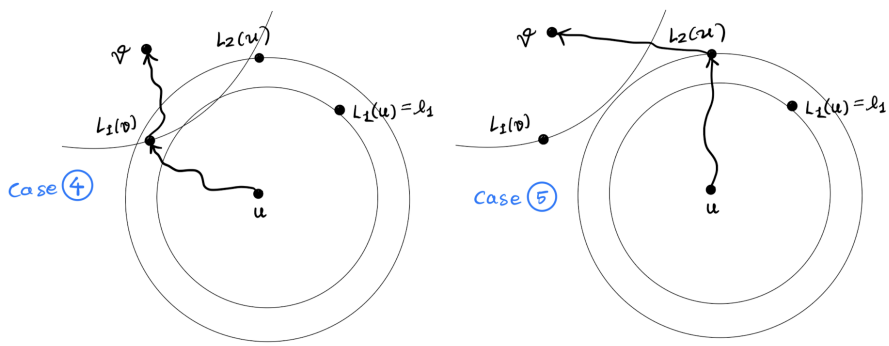


Figure 2: Cases while generalising to 5-Approximate Distance Oracle

6 (2k-1)-Approximate Distance Oracle

Building on our analysis of the 5-approximate distance oracle, we extend these ideas to construct a $(2k-1)$-approximate distance oracle for any positive integer k.

At each level, vertices are sampled with a specific probability, and local distance information is stored similarly to the 3-approximate case. The hierarchical structure allows us to bridge longer distances at higher levels, ensuring the overall approximation factor does not exceed $(2k-1)$.

This multi-level design preserves theoretical guarantees while improving both storage efficiency and query time for large graphs. In the following sections, we detail the construction of the oracle, explain the hierarchical sampling process, and analyze the algorithms for preprocessing, query answering, and their respective complexities.

6.1 Construction of the $(2k-1)$-Approximate Distance Oracle

The key insight is to always be able to jump from u. Further, we can exploit the symmetry of the distance calculation and conclude that the jump from the destination vertex also must be single. So, we would have to maintain balls of vertices from Landmarks of all levels for every vertex. Employing a hierarchical sampling scheme, we effectively will try to find an intermediate vertex in each of the balls to see if u or its focus belonging to the next level are in a common ball. We thus maintain localized distance information in a series of hash tables, which enables constant-time query answers. The construction proceeds as follows:

1. **Hierarchical Sampling:** We maintain k "levels" of landmark vertice "sets" $\{\text{landmark}^0, \text{landmark}^1, \ldots, \text{landmark}^k\}$, starting with $\text{landmark}^0 = V$. For each $1 \leq i \leq k$, every vertex in landmark^{i-1} is included in landmark^i independently with probability $n^{-1/k}$. Allowing vertices having a max level (called as the *rank*, henceforth) in all lower levels enables us to handle cases ensuring that a focus of higher level is always the same or farther than the lower level.

2. **Local Distance Storage:** We define the notion of locality using closest landmark at any level i, called as the *focus*. It is important to note that, by the nature of sampling as described earlier, the same vertex can be focus for a given vertex at different levels. Using this, we define the locality of vertex u as *ball* of u at level i as

$$\text{Ball}_i(u) := \left\{ v \,\middle|\, v \in \text{landmark}^i \text{ and } \delta(v,u) \leq \text{focal_distance}_i(u) \right\},$$

   where $\text{focal_distance}_i(u)$ denotes the distance from u to its closest landmark in $\text{landmark}^{i+1} = focus_i(u)$. These balls store accurate local distances needed for efficient query resolution.

3. **Global Distance Storage:** From the landmarks at level k, store distance to all vertices u to handle queries of far-away point. To handle this efficiently, add all the vertices in $landmark^k$ in balls of all vertices. This will also help in efficient retrieval using hashing of the ball and no explicit storage will be required.

The complete data structure for the oracle is the collection

$$\{\text{Ball}_i(u) : u \in V,\ 0 \leq ilk\}.$$

6.2 Size Analysis of the $(2k-1)$-Approximate Distance Oracle

We now analyze the expected size of the $(2k-1)$-approximate distance oracle. In our hierarchical sampling, each vertex from the preceding level is chosen for the next level independently with probability $p = \frac{1}{\sqrt[k]{n}}$.

For a fixed vertex u and a given level i, suppose we order the remaining vertices in $V \setminus \{u\}$ by increasing distance from u. A vertex appears in $\text{Ball}_i(u)$ if none of the vertices preceding it in this order have been sampled into $landmark^{(i+1)}$. Consequently, the expected number of vertices in $\text{Ball}_i(u)$ is bounded by

$$\mathbb{E}[|\text{Ball}_i(u)|] \leq \sum_{j=1}^{n-1} (1-p)^{j-1} = O\Big(\frac{1}{p}\Big) = O\Big(n^{1/k}\Big).$$

Since there are k levels, the total expected storage for vertex u is $O\Big(k \cdot n^{1/k}\Big)$. Summing over all n vertices, the overall expected size of the oracle becomes $O\Big(n \cdot k \cdot n^{1/k}\Big) = O\Big(k\, n^{1+1/k}\Big)$.

Thus, the $(2k-1)$-approximate distance oracle requires an expected space of $O(k\, n^{1+1/k})$, confirming its efficiency in terms of storage.

6.3 Query Answering Procedure

To determine an approximate distance between any two vertices u and v using our k-level oracle, we exploit the hierarchical data structure built during preprocessing and symmetry as in case of 5-Approximate Distance Oracle. However, we leverage the powerful intuition mentioned earlier of finding a point common to both the balls of level i. We start with u, and iteratively jump back and forth between u and v by checking if this vertex belongs to the other's ball, or otherwise upgrade myself to the higher level focus of the other vertex (moving closer, but risking loss of access to the older vertex). Note that in the 5 distance oracle procedure developed previously, it was equivalent to double the comparisons we make here, because we were moving two iterators as we had 1 ball lesser for each vertex. By sampling method, each $focus_i(u)$ gets stored in the corresponding hash table for $\text{Ball}_i(u)$, the distance $\delta(u, focus_i(u))$ is readily available. If the algorithm finds no such common vertex, it returns w.

The query answering mechanism for u, v operates as follows:

Algorithm 1 Approximate Distance Query

```
function QUERY(u, v)
    if u < 0 or v < 0 or u ≥ n or v ≥ n then
        return ∞
    end if
    if u = v then                                   ▷ We don't store ourselves in the ball
        return 0
    end if
    i ← 0
    w ← u
    while w ∉ Ball_i(v) do
        i ← i + 1
        if i ≥ k then                               ▷ In Case the Graph is disconnected
            break
        end if
        swap u ↔ v
        w ← focus_{i−1}(u)
        if w < 0 then                               ▷ Error-handling
            break
        end if
    end while
     return δ(u, w) + δ(v, w)
end function
```

Since the search in each level involves a constant-time hash table lookup and the number of levels is bounded by k, the query answering process requires at most $k+1$ steps. Consequently, the overall query time is $O(k)$ in practice.

6.4 Proof of Correctness

We prove that the query algorithm returns a $(2k-1)$-approximation of the true distance $\delta(u,v)$ for any $u, v \in V$.

At each level i, the algorithm either terminates with a common node w in the ball of one of the vertices, or continues to the next level, swapping the roles of u and v. Let w_i be the focus node selected at level i, and let the current roles of the vertices be u_i, v_i (swapped at each step).

We prove by induction that after i iterations:

$$\delta(u, w_i) \leq i \cdot \delta(u, v), \quad \delta(v, w_i) \leq (i+1) \cdot \delta(u, v)$$

Base Case ($i = 0$): Initially, $w_0 = u$. Hence $\delta(u, w_0) = 0$, and by the triangle inequality,

$$\delta(v, w_0) \leq \delta(u, v)$$

Inductive Step: Assume after j iterations,

$$\delta(u, w_j) \leq j \cdot \delta(u, v)$$

Now, the algorithm checks whether $w_j \in \text{Ball}_j(v)$. If not, it proceeds to level $j+1$, swapping $u \leftrightarrow v$, and sets $w_{j+1} = focus_{j+1}(u_{j+1})$.

Using the triangle inequality,

$$\delta(v, w_{j+1}) \leq \delta(v, w_j) \leq \delta(u, v) + \delta(u, w_j) \leq (j+1) \cdot \delta(u, v)$$

When the algorithm finds such a w_l in the ball of the other vertex, it returns:

$$\delta(u, w_l) + \delta(v, w_l) \leq (l + (l+1)) \cdot \delta(u, v) = (2l+1)\delta(u, v)$$

Final Bound: When the algorithm terminates at some iteration l (with $l < k$ as the level goes to a max value of $k-1$ after which, if the vertices are connected, a common highest level vertex must be available). The final approximation is:

$$\delta_{\text{approx}}(u, v) \leq (2k-1) \cdot \delta(u, v)$$

Remark. The algorithm may terminate earlier (i.e., $l < k$), in which case the stretch is strictly better than $(2k-1)$.

6.5 Expected Size of Datastructure

Exactly same analysis (imagining sorted order of increasing weights) for 3-approximate distance oracles follow for computation of expected size of $\text{Ball}_i(v)$ for any vertex $v \in V$ at level i giving us the bound $O(n^{1/k})$. This is crucial for ensuring that both the space and query time remain efficient in the approximate distance oracle.

Summing over all the balls and levels, we find that the expected size of the data structure is $O\left(n \cdot k \cdot n^{1/k}\right) = O\left(k\, n^{1+1/k}\right)$. To guarantee a worst-case size of $O\left(k\, n^{1+1/k}\right)$ for the data structure, we rerun the entire preprocessing algorithm. Since the expected number of repetitions required to achieve this bound remains constant, the overall asymptotic performance is not adversely affected. This is a significant improvement over the naive approach of storing all-pairs distances, which would require $O(n^2)$ space.

6.6 Preprocessing for (2k-1)-Approximate Distance Oracles

In this section, we describe a polynomial time algorithm for constructing a $(2k-1)$-approximate distance oracle. The approach extends the earlier 3-approximate method via a hierarchical sampling strategy. After forming the sequence of sampled vertex sets, the remaining work consists of computing, for each vertex u and for each level $i \leq k$, the set $\mathrm{Ball}_i(u)$ along with the distances from u to vertices in these sets. The following subsections outline the key components of the construction and their analysis.

Focus of all vertices at all levels For every vertex $u \in V$ and each level i, we need to determine $focus_i(u)$, the closest vertex in $landmark^{(i+1)}$ to u. A standard way to solve this is to augment the graph by adding a dummy vertex s connected via zero-weight edges to every vertex and then run Dijkstra's algorithm from s. This procedure takes $O(m \log n)$ time per level, and by processing all levels, we obtain an overall cost of $O(k\, m \log n)$ for computing the representatives.

Efficient Computation of Balls

For each vertex u and for each level i with $1 \leq i < k$, we define the set

$$\mathrm{Ball}_i(u) = \{v \in landmark^{(i)} \mid \delta(u, v) < \delta(u, focus_i(u))\}.$$

To compute these sets efficiently for a vertex u, we look for vertices v such that $u \in Ball_i(v)$ as the expected number of vertices v is bounded. Thus, for any vertex $v \in landmark^{(i)}$, define the group as follows:

$$G_{i+1}(v) = \{w \in V \mid v \in Ball_i(w)\}.$$

Trim Dijkstra's Algorithm for Group Computation

A key observation is the following lemma:
Lemma: On the shortest path from vertex u to $v \in G_i(v)$, all the vertices must also belong to $G_i(v)$.

Thus instead of computing every ball from source, we reverse the order to update which node's ball a given vertex can belong to (as this can be bounded). Further, using the property that as Dijkstra progresses, the distance computed never increases, we need to visit only those vertices which belong to $G_i(u)$ at any point in Dijkstra. This means, an edge $(v,\ y)$ is relaxed using an addition condition from vertex v which is:

$$d(u, v) + \mathrm{weight}(v, y) < \delta(y, L(y))$$

This restriction ensures that vertices falling outside the group are not explored.

Since each edge relaxation using a min-heap takes $O(\log n)$ time, the total time for computing a single group $G_{i+1}(v)$ is proportional to the sum of the degrees of vertices in the group multiplied by $\log n$. Summing over all group at a given level yields an overall cost of $O\left(\sum_{u \in V} |\text{Ball}_i(u)| \log n\right)$. Given that the expected size of $\text{Ball}_i(u)$ is $O(n^{1/k})$, the expected cost for level i is $O(m\, n^{1/k} \log n)$. By processing all k levels, the total cost for computing ball sets is $O(k\, m\, n^{1/k} \log n)$. With Fibonacci heaps, one can reduce or eliminate the logarithmic factor.

Final Algorithm for Preprocessing

Algorithm 2 Preprocessing for Approximate Distance Oracle (with expected size $O(k\, n^{1+1/k})$)

```
procedure PREPROCESS(Graph G = (V, E), parameter k)
    CHOOSELANDMARKS
    for i ← 1 to k − 1 do
        MULTISOURCEDIJKSTRA(landmarks[i], i)
    end for                                   ▷ Skip i = k; focus assumed at infinity
    for i ← 0 to k − 1 do
        for each vertex v in landmarks[i] do
            if rank[v] ≥ i then
                TRIMMEDDIJKSTRA(v, i)
            end if
        end for
    end for
    For each vertex v and level i, add (focus[v][i], focus_distance[v][i]) to ball[v][i + 1].
        ▷ Special Case: The focus of the higher level is at the same distance but different.
    total_space ← HASHBALLS
    HASHBALLS                 ▷ For efficient retrieval, using a stronger hash function
end procedure
```

Overall Running Time and Space Analysis

Combining the time for sampling, computing focii, and forming the balls, the total expected running time for building the data structure is $O(k\, m\, n^{1/k})$,

6.7 Final Result

We thus conclude that, for a weighted undirected graph $G = (V, E)$ and an integer k, one can build a data structure of (expected) size $O\left(k\, n^{1+1/k}\right)$ in $O\left(k\, m\, n^{1/k}\right)$ time. This data structure enables us to report a $(2k-1)$-approximate distance between any two vertices in $O(k)$ time.

7 Implementation

C++ was chosen as the main language for experimentation as it provided greater control over memory allocation and internal data structures. A visualization module was also created, based on

initial prototyping in Python and for additional support. All charts were generated using Python and Jupyter Notebooks. Each directory includes a `README` file containing exact instructions on how to run and use the code.

Key Implementation Details

- Initially, we started with a direct implementation of the code. However, for larger graph sizes (> 1000 nodes), memory pressure became significant due to edge weights being of type `double` (8 bytes). Consequently, the codebase was restructured into two separate executables and following a full *object-oriented design*, separating graph, algorithm and preprocessing functions from the main ones. Further, it facilitated fast deallocation of heap memory and lower usage of the stack. A CMakefile was created to facilitate easy compilation and execution of the code.

- STL-based containers were primarily used, as they provided robust and efficient performance for our use case. Since the default `unordered_map` in STL does not guarantee worst-case constant-time lookup, we opted for an alternative implementation using the FPH library [1]. Hashing was performed using the FCH algorithm with a `MixSeedHash` function, as part of the `metaFPHMap` table, resulting in very low output query time.

- To test various edge cases, we added support for custom landmark selection. For checking robustness, we manually tested for different input-output pairs and special cases.

- For computing true distances between vertices, two methods were implemented: one using the Floyd-Warshall algorithm and the other using Dijkstra's algorithm for each query. We also explored implementing bidirectional version of Dijkstra and A* search, but they were not greatly useful as we had multiple queries and choosing a heurestic function for different graphs was cumbersome. This step was one of the most time-consuming. The final implementation used Dijkstra's algorithm for each query, which was fast enough for our needs.

- A Python script was created to generate queries, either exhaustively or by uniformly sampling the required number of queries, especially for large graphs.

Choice of Datasets

1. We shortlisted following representative graphs from categories:
   (a) RoadNet California [3]
   (b) Facebook Social Circles [4]
   (c) Enron Email Cluster and Hierarchical Structure [5]
2. We explored various *publicly* available datasets and made the following observations:
   - Most real-world datasets were unweighted.
   - They were either too small (a few KBs to MBs) or too large (around 10GB).
   - Many datasets were unclean, containing inconsistencies such as duplicate edges, incorrect metadata (e.g., number of nodes), and format anomalies. Cleaning these required significant manual preprocessing.

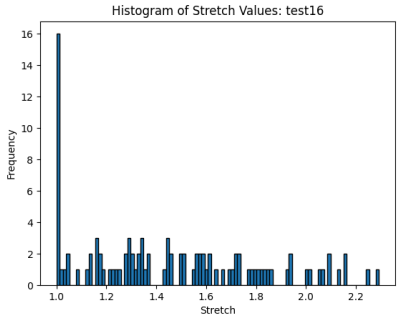
3. To address the limitations of real-world datasets, we generated synthetic graphs using LFR-Benchmarks [6] to obtain fine-grained control over parameters such as average degree and number of nodes.

4. We attempted to run our algorithm on large datasets (around 10GB) on CSE servers, but due to long runtimes, the process frequently got killed becuase of network issues. Eventually, we restricted ourselves to graphs with up to 50,000 nodes. *Note:* 2 of the servers had incompatible cmake configurations, and executables wouldn't get compiled as well. Without sudo access, we couldn't install the required libraries. However, we did carry out the testing for sufficiently large graphs on local machine on MacOS and AppleClang.

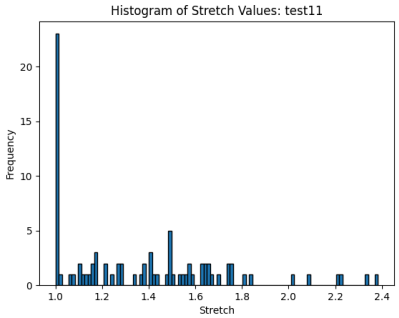
5. Approximately 40 runs were conducted on different graph datasets. A summary of the results is provided in the following section.

Key Observations

1. For a fixed $k = 3$ and average degree, as n increases, the stretch distribution shifts to the right; however, the persistent mass of stretch values close to 1 remains high. Thus, the randomized algorithm scales gracefully—its worst-case stretch may worsen on huge graphs, but in practice most distance queries remain highly accurate even as n increases.

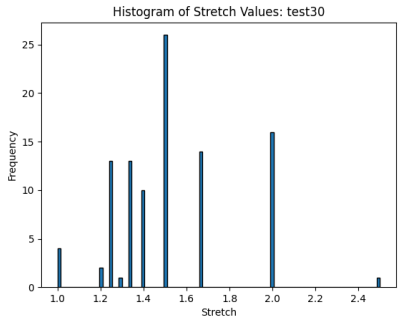


(a) Image 1 (n = 100)

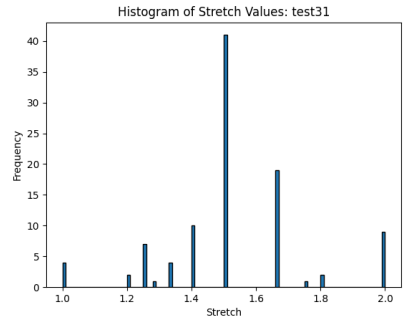


(b) Image 2 (n = 36692)

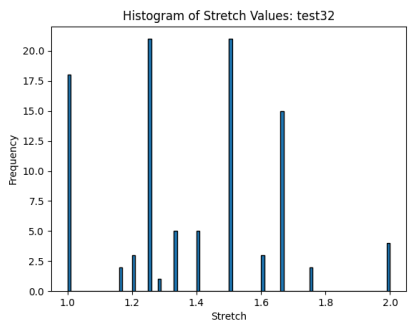
2. For a fixed $n = 4039$ and average degree, as k increases, the stretch distribution shifts to the right; however, the frequency of stretch values close to 1 remains high. By raising k we effectively compress the oracle (use less space or time) but loosen the distance guarantees; yet in practice even a very "loose" setting (k=9) returns exact distances for a large fraction of queries, showcasing the dataset's high compressibility.



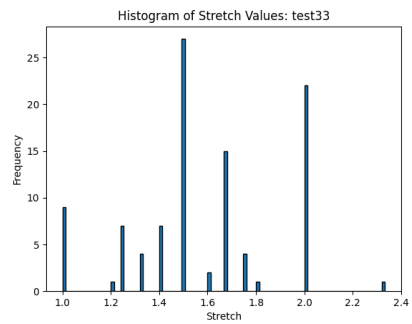
(a) Image 1 (k = 2)



(b) Image 2 (k = 3)

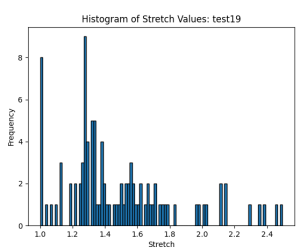


(a) Image 3 (k = 5)

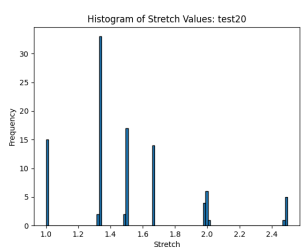


(b) Image 4 (k = 9)

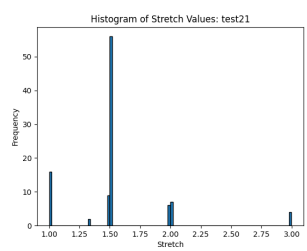
3. As the average degree decreased, instances of higher stretch became more frequent. However, clusters start appearing in the histogram because more connectivity yields better approximate routes of roughly the same stretch.



(a) Image 1 (deg = 15)



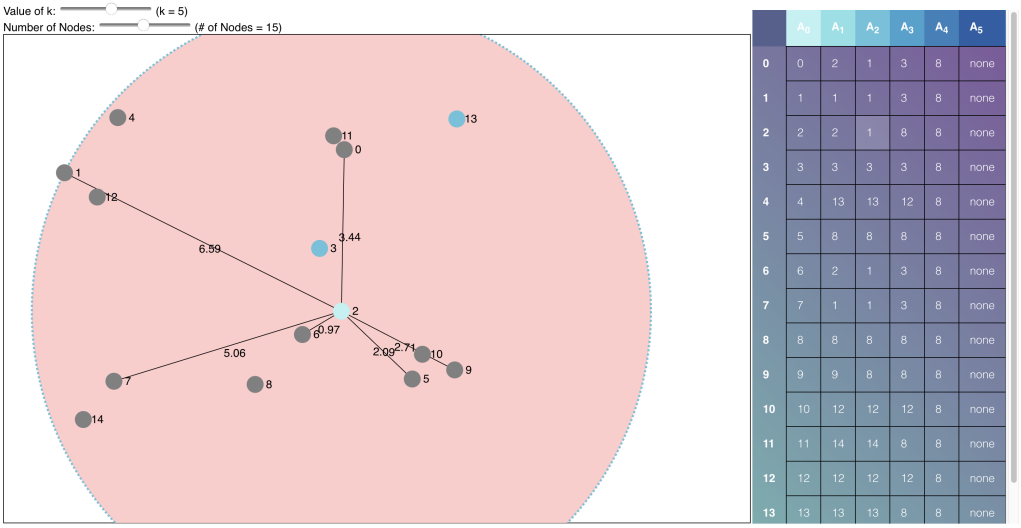
(b) Image 2 (deg = 50)



(b) Image 3 (deg = 100)

4. The use of the custom hashing function maintained an average (query time / k) of approximately `357ns`, with a standard deviation of `200ns` proving usefulness of our hashing library.

5. The number of words used to represent graphs was exponentially smaller than the $O(n^2)$ worst-case expectation across all test cases (an average 11% compression.)



	A_0	A_1	A_2	A_3	A_4	A_5
0	0	2	1	3	8	none
1	1	1	1	3	8	none
2	2	2	1	8	8	none
3	3	3	3	3	8	none
4	4	13	13	12	8	none
5	5	8	8	8	8	none
6	6	2	1	3	8	none
7	7	1	1	3	8	none
8	8	8	8	8	8	none
9	9	9	8	8	8	none
10	10	12	12	12	8	none
11	11	14	14	8	8	none
12	12	12	12	12	8	none
13	13	13	13	8	8	none

Figure 3: Visualisation

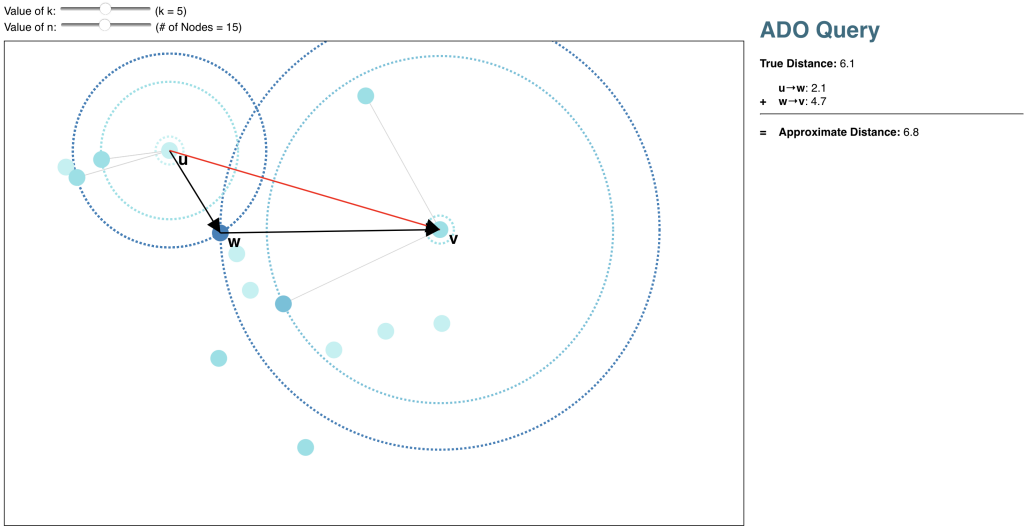


Figure 4: Visualisation

8 Future Work

- **Parallelization for Fast Pre-Processing:** One promising direction is to parallelize the preprocessing phase. Since computing local distance information and forming the clusters or ball sets for each vertex can be done independently, distributing this work across multiple processors or cores can significantly reduce the overall runtime. Further optimisations for memory allocation and cache usage can also be explored to improve performance.
- **Dynamic Updates:** Enhancing the data structure to support dynamic updates—such as insertions, deletions, or weight modifications—without requiring a full re-preprocessing of the graph would make the oracle more practical for real-time or evolving networks.

References

[1] `https://github.com/renzibei/fph-table/tree/master`

[2] `https://github.com/TylerYep/approx-distance-oracle/`

[3] `https://snap.stanford.edu/data/roadNet-CA.html`

[4] `https://snap.stanford.edu/data/ego-Facebook.html`

[5] `https://snap.stanford.edu/data/email-Enron.html`

[6] `https://github.com/eXascaleInfolab/LFR-Benchmark_UndirWeightOvp`