A Generalized Approach for Reducing Expensive Distance Calls for A Broad Class of Proximity Problems

Jees Augustine
The University of Texas at Arlington
jees.augustine@mavs.uta.edu

Suraj Shetiya
The University of Texas at Arlington
suraj.shetiya@mavs.uta.edu

Mohammadreza Esfandiari
New Jersey Institute of Technology
me76@njit.edu

Senjuti Basu Roy
New Jersey Institute of Technology
senjutib@njit.edu

Gautam Das
The University of Texas at Arlington
gdas@cse.uta.edu

ABSTRACT

In this paper, we revisit a suite of popular proximity problems (such as, KNN, clustering, minimum spanning tree) that repeatedly perform distance computations to compare distances during their execution. Our effort here is to design principled solutions to minimize distance computations for such problems in general metric spaces, especially for the scenarios where calling an expensive oracle to resolve unknown distances are the dominant cost of the algorithms for these problems. We present a suite of techniques, including a novel formulation of the problem, that studies how distance comparisons between objects could be modelled as a system of linear inequalities that assists in saving distance computations, multiple graph based solutions, as well as a practitioners guide to adopt our solution frameworks to proximity problems. We compare our designed solutions conceptually and empirically with respect to a broad range of existing works. We finally present a comprehensive set of experimental results using multiple large scale real-world datasets and a suite of popular proximity algorithms to demonstrate the effectiveness of our proposed approaches.

CCS CONCEPTS

• **Theory of computation** → **Dynamic graph algorithms**; *Nearest neighbor algorithms*; • **Information systems** → Crowdsourcing.

KEYWORDS

Metric Space; Lower Bound Computation; Proximity Problems; Minimum Spanning Trees; k-Nearest Neighbour Graphs, Clustering

ACM Reference Format:
Jees Augustine, Suraj Shetiya, Mohammadreza Esfandiari, Senjuti Basu Roy, and Gautam Das. 2021. A Generalized Approach for Reducing Expensive Distance Calls for A Broad Class of Proximity Problems. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457303

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457303

1 INTRODUCTION

Given a set of n objects with distances defined between each pair of objects, various classical *proximity problems* have been investigated over the decades in data management research, such as k-nearest neighbor, clustering, shortest path, minimum spanning tree, and several others. In this paper, we consider the setting where the objects are in *a general metric space* and *distance computations are the dominant cost of algorithms* for these problems.

1.1 Novelty, Motivation, and Applications

The novelty and motivation behind our proposed approach hinges on the following three characteristics:

Characteristic 1: General Metric Spaces: We study proximity problems in general metric spaces. That is, we are given a collection of atomic objects, and a general distance function between pairs of objects that obeys the triangle inequality or relaxed triangle inequality. Beyond that, we do not assume special cases such as Euclidean or vector spaces, where the objects are further decomposed/represented as vectors of attributes, and the distance function is defined over pairs of vectors (e.g., Euclidean distance, Cosine similarity, etc). This renders most of the vast collection of existing proximity research in Euclidean/vector spaces inapplicable to our problem (Section 6 has further details).

Characteristic 2: Optimizing Distance Computations: We consider scenarios and applications where distance computations require access to a *distance oracle* which is an expensive function. Thus these applications benefit from specifically minimizing distance computation costs - sometimes at the expense of increased CPU computation. This unique optimization goal has not been addressed by prior general metric space proximity research; most such works do not separate distance computation from CPU computation costs (see related work in Section 6.)

Characteristic 3: Unified Framework with Exact Outputs: One of the significant highlights of our approach is, rather than redesigning all prior existing algorithms for the myriad proximity problems on a case-by-case basis, we provide a *unified framework* in the form of a general solution scheme. We show how such a framework can be easily applied to making minor modifications to prior proximity algorithms, resulting in a significant reduction in the number of calls to the distance oracle, at the expense of a comparatively small increase in the rest of the computation costs. Moreover, our framework *does not change the outputs of the original algorithm.* For example, if we use our framework to modify a classical general metric space k-nearest-neighbor algorithm, the

modified algorithm will produce the correct k-nearest-neighbors, but make fewer distance calls.

Applications: There are several important and emerging applications that have the above three characteristics, and consequently can leverage our framework. Notable examples include spatial applications that require geo-referencing, computer vision applications, as well as applications from bioinformatics and medical imaging. Also of interest are spatial applications that require calling third party API's (such as, Google[1] or Bing[2] map) to obtain distances between pairs of locations, GPS and other map based services that need to obtain point-to-point driving time (typically calculated based on distance, traffic condition, etc). Several compelling computer vision applications, including Hyperspectral Image comparison[1, 7], Image database search [8], Image comparisons under Hausdorff distance [22] or Video Database Searching [12] make use of triangle inequality. Additionally, DNA [48] sequence analysis, protein Database search [15] are some of the compelling bioinformatics applications that require expensive distance computations. In fact, as related works suggest, efficient comparison in metric space is even desirable for studying medical imaging technologies, like MRI Scan [32], fMRI Scans[43], CAT-SCAN[20] analysis. All these applications stand to benefit from our work.

1.2 Technical Contributions

The heart of our techniques is based upon the following observations. During the process of computation, most proximity algorithms repeatedly need to compare distances between various pairs of objects, or compare various distance aggregates such as sums of distances. For example, while computing the k-nearest neighbor of a query object u, existing algorithms iteratively check if there is any other object v whose distance from u is smaller than the object's distance from its current k-th nearest neighbor w (*i.e.*, whether $dist(u, v) < dist(u, w)$). If this answer turns out to be true, then the current k-th nearest neighbor is updated. However, for the algorithm, it is not necessary to always know the precise distances $dist(u, v)$ and $dist(u, w)$. It is just sufficient to know whether the linear inequality $dist(u, v) - dist(u, w) < 0$ is true. If true, then v can be safely discarded, thus saving on distance calls.

We leverage the above observations to make the following technical contributions:

Contribution 1: Linear Program Modeling (Section 2.2): Our first contribution is in identifying IF statements in proximity algorithms that compare linear distance expressions, and showing how they can be more efficiently redesigned without having to invoke expensive distance oracle calls by modelling them as *a system of linear inequalities*. For such IF statements, we provide guidelines for re-authoring them such that expensive distance oracle calls are replaced by linear constraints. We present a model, Direct Feasibility Test, that involves expressing the problem as a system of linear inequalities which can be solved by only using local CPU resources (Section 2). To the best of our knowledge, no prior work has presented this formalism before.

[1]https://cloud.google.com/maps-platform
[2]https://www.microsoft.com/en-us/maps/choose-your-bing-maps-api

Contribution 2: Graph-Theoretic Modeling and Efficient Algorithms (Sections 3 and 4): For scenarios where solving linear programs place unacceptable demands on local computation resources, we propose a simpler yet novel redesign of IF statements by mapping them to lower and upper bound distance computation problems. As an illustrative example, if the upper bound of $dist(u, v)$ can be shown to be smaller than the lower bound of $dist(u, w)$, then this implies that $dist(u, v) < dist(u, w)$. We show that such upper and lower bound problems can be mapped to interesting computation problems over *sparse weighted graphs*, which although suboptimal compared to the LP formulation (the former approach saves more distance calls), they allow for more efficient algorithms that make far less demands on local CPU resources.

We present a new lower bound estimation algorithm, referred to as *Shortest Path Based Solution Scheme* (`SPLUB` in short) which considers the sparsity of the graph while computing the lower bound improving the computational efficiently of the algorithm.

Then, we present an optimized "lightweight" bound estimation algorithm *Triangle Based Solution Scheme* (`Tri Scheme` in short) that is highly scalable, by constraining to search to a local neighborhood of paths of length 2 of the graph. We also present an expected case analysis for `Tri Scheme` in Section 4.2.2

Contribution 3: Extensive Experimentation (Section 5): Our final contribution lies in performing extensive experiments and outperforming the appropriately adapted current-state-of-the-art solutions with the help of real-world datasets. Besides demonstrating algorithmic efficiency, our experiments also highlighted the ease with which our proposed re-authoring methods can be adapted for a wide class of proximity algorithms.

2 DISTANCE COST MINIMIZATION

In this section, we study how existing proximity algorithms incur distance cost inside the computational loop and propose a general purpose model, Direct Feasibility Test to minimize that cost.

2.1 Working Principles of Proximity Algorithms

Proximity problems rely on establishing proximity relationship among different objects in order to decide the best set of outputs, and play fundamental roles in database research. Examples of such problems include the k-NN, computing Minimum Spanning Tree (MST), clustering problems, etc.

(1) **IF statements involving distance calls** - At the heart of the proximity problems, there exist repeated distance comparisons. Typically, one or more calls to the distance oracle are associated with every invocation of such comparison. As an example, consider any clustering algorithm with the overarching goal of putting similar objects together in the same group, and keeping dissimilar objects in different groups. These algorithms repeatedly compare distances between a set of objects to make such a decision.

```
if dist(o_i, o_j) ≥ dist(o_k, o_l)
    do something
else
    do something_else
```

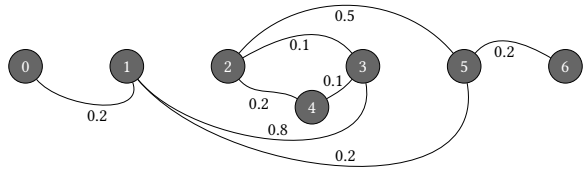


Figure 1: 7 objects and their corresponding known and unknown distances.

② **Saving distance calls in IF statements** - When the distances are from a general metric space, there exists a relationship between the distances - our goal is to exploit that in saving distance calls.

Consider a set, $O = \{o_i, o_2, \ldots, o_n\}$ of n objects. We assume no two objects in O are the same. The underlying dissimilarity between each pair is the *distance* between them, represented by $dist(o_i, o_j)$.

Metric Spaces and Triangle Inequality: A Metric Space is an ordered pair $(\mathcal{M}, d)$ where $\mathcal{M}$ is a collection of objects and $dist$ is a *distance metric* on $\mathcal{M}$. In addition, for any triplets $(m_i, m_j, m_k) \in \mathcal{M}$,

$$dist(m_i, m_j) = 0 \implies (m_i = m_j)$$
$$dist(m_i, m_j) = dist(m_j, m_i)$$
$$dist(m_i, m_j) \leq dist(m_i, m_k) + dist(m_k, m_j) \text{ (}\triangle \textbf{ inequality)}$$

Informally, the triangle inequality implied that the distance between any pair of objects is less than or equal to the distance of a path between the same pair of objects that goes through any other object(s).

Example 2.1. (Running Example) : Consider a set of 7 objects $\{0, 1, 2, 3, 4, 5, 6\}$. Let us also assume that the distance between every pair of objects is between 0 and 1. Assume these distances satisfy the metric property, i.e., triangle inequality of distances.

The running example is shown in Figure 1. As shown in the figure, we also assume that 8 pairwise distances are known (i.e., the distance oracle has been called for each of these pairs). The solid lines between the objects represent the distance that is known.

2.2 Direct Feasibility Test

The triangle inequality relationship among the objects could be represented using a set of inequalities involving O.

Using the example in Figure 1, let us create $\binom{n}{2}$ variables of the form x_{ij}, where each variable represents the distance between the respective pair of objects. Next, we create linear inequalities that constrain the values these variables can have. For example, for the pair of objects (o_1, o_3) whose distance is known, we add two inequalities of the form $(x_{13} - 0.8 \leq 0)$ and $(-x_{13} + 0.8 \leq 0)$ (i.e., together equivalent to the equation $x_{13} = 0.8$). Similarly, for each pair of objects whose distance is unknown, for example, (o_1, o_2), we add constraints of the form $(x_{12} - 1 \leq 0)$ and $(-x_{12} \leq 0)$. Thus far, the system will have $(2 \times \binom{n}{2} \rightarrow 42)$ inequalities, with two inequalities corresponding to each x_{ij}.

Next, corresponding to each triangle, x_{12}, x_{23}, x_{13} we will have additional inequalities of the form, $(x_{12} - x_{23} - x_{13} \leq 0)$, $(-x_{12} - x_{23} + x_{13} \leq 0)$ and, $(-x_{12} + x_{23} - x_{13} \leq 0)$. There are $\binom{n}{3}(\binom{7}{3}$ in the example) number of triangles in a set of n objects. Each triangle gives rise to a set of 3 linear inequalities. Thus for our running example, the consideration of all triangles adds $(3 \times \binom{n}{3} \rightarrow 105)$ number of linear inequalities to the linear system.

For an IF statement such as $if\ dist(o_2, o_6) < dist(o_3, o_5)$, we formulate a corresponding additional constraint, $(x_{26} - x_{35}) < 0$. However, we should be checking for the absence of any feasible region for the reversed constraint, expressed as follows, $(-x_{26} + x_{35}) \leq 0$. This reversed constraint is added to the system of linear constraints.

Thus, in order to save distance calls in the IF statement, our approach is to solve the following decision problem: *Does there exist no feasible solution to the system of inequalities?* if the answer to that question is *YES*, the if condition is satisfied. If the answer is *NO*, then the proximity algorithm may call the distance oracle to obtain the exact distances and repeat the computation. This, in a nutshell, is the core idea of our proposed approach.

Solving the system of linear inequalities: Formally, the system of linear inequalities can be written as follows:

$$AX \leq b \tag{1}$$

where A forms the coefficient matrix, X is a vector of unknown distances, b is a vector of known coefficients.

Determining whether this system of linear inequalities has a feasible region or not could be solved using existing off-the-shelf linear programming tools. For example, SIMPLEX [50] could be used to solve this problem. However, the number of iterations for SIMPLEX in the worst case is exponential in the number of objects [28]. A more practical approach to linear programming through the ellipsoid algorithm [27] also could be used. However, solving linear inequalities through this method is in $O(n^6)$. These algorithms thus are not practical even for a small number of objects.

3 GRAPH THEORETIC APPROACH

Contrary to employing expensive linear programming to resolve the IF statements statement exactly - an alternative, less expensive approach is to redo the IF statements statement as follows:

```
if LBdist(o_i, o_j) ≥ UBdist(o_k, o_l)
    do something
else
    do something_else
```

This above formulation is designed to compute the *lower bound* (LB) of distance between o_i, o_j and compare that with the *upper bound* (UB) of distance between o_k, o_l. We emphasize that such a reformulation of the IF condition is *not the same* as the original IF condition; If the reformulated condition is true, the original condition is true, but not vice versa. In the vice versa case, the distance oracle has to be invoked to accurately resolve the IF statement.

The advantage of the reformulated condition is that it allows us to use much more efficient and scalable graph theoretic approaches for resolving the condition as compared to the linear programming approaches described earlier, thus resulting in dramatic savings in local CPU computations, at the cost of a small increase in the number of calls to the distance oracle.

Thus our next set of investigation hinges on finding lower and upper bounds of distances using a suite of computational techniques

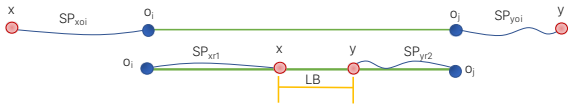


Figure 2: Geometric Interpretation of LB. [Top] Shortest Paths through Known Edge. [Bottom] Wrapping SP onto Known Edge.

that considers the underlying abstraction to be a complete graph on general metric spaces. Specifically, if indeed $LBdist(o_i, o_j) \geq UBdist(o_k, o_l)$, then two distance calls to the oracle $dist(o_i, o_j)$ and $dist(o_k, o_l)$ could be saved.

3.1 Data Model

Abstractly, the distance relationship over the given set of objects is abstracted as *a weighted complete graph*, $\mathcal{G}$. The nodes are defined over the set of objects (O), and every pair of nodes in the object set, (o_i, o_j) forms the edges in the graph whose edge weights are induced by the distance function, $dist(o_i, o_j)$. As before, the distance between the objects satisfy the metric property, i.e., the triangle inequality.

Definition 1. *The Tightest Upper Bound of the distance between (o_i, o_j) (or $TUB^{dist(o_i,o_j)}$), is the largest possible distance that an unknown edge can assume without violating the triangle inequality, considering all other known distances in $\mathcal{G}$.*

It is easy to see that the tightest upper bound of distance between o_i and o_j is the length of the shortest-path(sp) distance between those objects [45] that will go through additional intermediate objects. Note that, there might be other paths between (o_i, o_j) which also provide an upper bound on $UB^{dist(o_i,o_j)}$ but are not as tight as $TUB^{dist(o_i,o_j)}$).

$$TUB^{dist(o_i,o_j)} = sp(o_i, o_j) \leq UB^{dist(o_i,o_j)} \tag{2}$$

Definition 2. *The Tightest Lower Bound of the distance between (o_i, o_j) (or $TLB^{dist(o_i,o_j)}$), is the lowest possible distance that an unknown edge can assume without violating the triangle inequality, considering all other known distances in $\mathcal{G}$.*

The tightest lower bound $TLB^{dist(o_i,o_j)}$ involves computing LB between o_i and o_j considering every path and taking the maximum. For each path, the TLB could be computed using the generalized metric property proposed in [42] - which involves subtracting the weight of the rest of the path (computed by taking the sum of known distances) from the highest weight edge (let that be $dist(o_k, o_l)$ between o_k, o_l) in p. Similar to upper bounds, any other path could lead to a lower bound which might not be tightest as $TLB^{dist(o_i,o_j)}$, and, we refer to them as $LB^{dist(o_i,o_j)}$.

$$TLB^{dist(o_i,o_j)} = \max_{\forall p}\{dist(o_k, o_l) - path(o_i, o_k) - path(o_l, o_j)\} \tag{3}$$

$$LB^{dist(o_i,o_j)} \leq \forall p\{dist(o_k, o_l) - path(o_i, o_k) - path(o_l, o_j)\} \tag{4}$$

To explain further, we refer to Figure 2 to find out $TLB^{dist(X,Y)}$. Let SP_{X,o_i} be the shortest path between X and o_i (curved lines in blue). Similarly, SP_{Y,o_i} be the shortest path between Y and o_j. Thus, we can visualize the equation 4 in the light of the figure as shown by the wrapping of shortest paths from X and Y on to the known edge (o_i, o_j). The lower bound, $LB^{dist(X,Y)}$, obtained from this path [X-SP_{X,o_i}-(o_i, o_j)-$SP_{o_j,Y}$-Y], is the residue on edge length ($dist(o_i, o_j)$ - SP_{X,o_i} - $SP_{o_j,Y}$) from the wrap over (highlighted interval in yellow). By definition 2, $TLB^{dist(X,Y)}$, is the maximum of all such lower bounds over all paths available between X and Y. Recall Figure 1 again and note that an alternative representation of the figure is a weighted complete graph on metric space, for which some of the edges are known and the rest are unknown. Using Example 2.1, only 8 out of 21 $\binom{7}{2}$ edges are known (the solid lines), while the remaining 13 edges are unknown. As given in the figure, if the distance $dist(1,\ 3) = 0.8$ and $dist(3,\ 4) = 0.1$ then the tightest bounds for distance $d(1,\ 4)$ may be computed as follows:

$$|dist(o_1, o_3) - dist(o_3, o_4)| \leq dist(o_1, o_4) \leq dist(o_1, o_3) + dist(o_4, o_3)$$

i.e., $0.7 \leq dist(o_1, o_4) \leq 0.9$

3.2 Problem Definitions

In this section, we formally define the studied problems considering the underlying abstraction to be a complete graph:

> Problem 1. (Bounds Problem) : *Given a partial graph, $\mathcal{G}(O, E)$, and an unknown edge (o_i, o_j) in the graph, find the tightest (i) lower bound of distances (or $TLB^{dist(o_i,o_j)}$), and (ii) find the tightest upper bound of distances (or $TUB^{dist(o_i,o_j)}$), without violating the triangle inequality, considering all other known distances in $\mathcal{G}$ but avoiding any calls to the expensive distance oracle, O.*

For instance, from the discussion following Example 2.1, the query problem on the partial graph for the edge $dist(o_1, o_3)$, would yield, the tightest lower bound as 0.7 and tightest upper bound as 0.9.

A proximity algorithm may have to make two calls to the *distance oracle* if the produced bounds are not effective to follow either of the branches of the IF statements statement. Following each call to the distance oracle on an *unknown* edge and its subsequent resolution, the partial graph evolves by adding an additional *known* edge to the graph. The graph will be represented as an adjacency matrix or adjacency list representation. Consequently, upon a new distance resolution, we have to update respective edge information to the graph data structures. Correspondingly, after an edge resolution, data structures keeping track of upper and lower bounds also may have to be updated. Here, we define the update problem as follows,

> Problem 2. (Update Problem) : *Given a partial graph, $\mathcal{G}(O, E)$, the actual distance (from oracle call) of a newly known edge (o_i, o_j), update the data structures that keep track of the lower and upper bounds of the remaining unknown edges.*

In the next Sections, we present multiple solutions that trade-off between tightness of produced bounds and running time to solve the 1 and 2 problems.

4 BOUND COMPUTATION ALGORITHMS

Solutions to every proximity problems involve two fundamental steps which often works in tandem, contributing towards the progress of the algorithm, (i) a distance resolution procedure for estimating the unknown distance and, (ii) an update operation which adds the resolved edge to the graph and associated data structures.

Our proposed two solution schemes trade-off between time and tightness of the produced bounds during the update. Nevertheless, when used in conjunction with *any proximity algorithm*, they both produce *an exact and identical* solution as that of the original algorithm.

Discussion - Running Example - Let us take the example of the same two unknown edges (o_2, o_6) and (o_3, o_5). Let us also assume an IF statements in proximity problems need to evaluate is of the form IF $(o_2, o_6) > (o_3, o_5)$. Considering graph theoretic approaches, we restate this IF statements as, IF$(LB^{dist(o_2,o_6)} \geq UB^{dist(o_3,o_5)})$. From the definitions of upper and lower bounds earlier in this section, it could be shown that $LB^{dist(o_2,o_6)} = 0.3$ and $UB^{dist(o_3,o_5)} = 0.6$. Since $0.3 \not\geq 0.6$, it is evident from this example that a distance save up, which previously facilitated by Direct Feasibility Test , cannot be obtained here thus necessitating two oracle calls for $dist(o_2, o_6)$ and $dist(o_3, o_5)$.

4.1 Exact Algorithms

WE describe exact Algorithm SPLUB(*Shortest Path Based Lower and Upper Bound* algorithm) for bounds computation and the update problem. SPLUB is sparsity sensitive - hence its running time depends on the number of known edges when it is invoked.

4.1.1 *Algorithm Development:*

Recall from Definition 1 that, in any given graph, the tightest upper bound of the distance between objects o_i and o_j is the length of the shortest-path(sp) distance between those objects that will go through additional intermediate objects.

Similarly, by Definition 2, the tightest lower bound $LB^{d(o_i,o_j)}$ involves computing LB between o_i and o_j considering every path and taking the maximum.

Aforementioned definitions, their application in examples 2.1 and understanding the sparsity of the graph formally sets the foundation for the SPLUB algorithm.

Exact Upper Bound Algorithm - Our upper bound computation is inspired by Dijkstra's Algorithm [17].

To find out the TUB between an unknown pair of edge $(o_i,\ o_j)$, we start a shortest path algorithm from one endpoint, let us say from o_i to find the shortest path to the other endpoint o_j. This in turn solves the problem of upper bound using the Equation 2.

Developing Exact Lower Bound Algorithm - Essentially, for each of the unknown edge in the graph, the lower bound can be estimated with the help of known edges. As established earlier, we run two shortest path algorithms from each endpoint of the unknown edge for which we need to compute the lower bound. Thus, for each known edge in the graph, we compute the shortest path from both the endpoints of the unknown edge to both the endpoints of the known edge. Since the tightest lower bound is largest of the all available lower bound distances, we only keep track of the current largest value at each iteration.

Algorithm 1 SPLUB

Input : graph $\mathcal{G}$ = (O,E), unknown edge (o_i, o_j), Dijkstra's sp algo $SP_{Dijk}()$

Output : $TLB^{d(o_i,o_j)}$, $TUB^{d(o_i,o_j)}$

1: $lb \leftarrow 0; ub \leftarrow 1$
2: $sp_{o_i} \leftarrow SP_{Dijk}(o_i)$
3: $sp_{o_j} \leftarrow SP_{Dijk}(o_j)$
4: **for** edge(k, l) in E **do**
5:

$$lb = max(lb, dist_{(o_k,o_l)} - (sp_{o_i}[o_k] + sp_{o_j}[o_l]), dist_{(o_k,o_l)} - (sp_{o_j}[o_k] + sp_{o_i}[o_l]))$$

6: **end for**
7: $ub = min(ub,\ sp_{o_i}[o_j])$
8: $TLB^{d(o_i,o_j)} = lb$
9: $TUB^{d(o_i,o_j)} = ub$
10: **return** $TLB^{d(o_i,o_j)}, TUB^{d(o_i,o_j)}$

Further Illustration - SPLUB. As an example, let us consider the unknown edge (o_1, o_2) from Example 2.1, for which Algorithm SPLUB is to be invoked to compute its TUB and TLB.

The lower and upper bounds for the edge is initialized as '0' and '1' respectively. After that, steps 2 and 3 of SPLUB computes all pair shortest paths from o_1 and from o_2. Next, for each of the 8 known edges in the graph, the LB is updated using Equation 4 and the TLB of (o_1, o_2) is 0.7, the *max* of all lower bound values.

The TUB of (o_1, o_2) is 0.7, computed in line 7 of the algorithm as the shortest path between these two objects.

Lemma 4.1. *The bound computed by the Lower Bound Algorithm in* SPLUB*, produces exact tightest lower bounds for the unknown edge.*

Proof. Assume that we do no produce the tightest lower bound on the given unknown edge (o_i, o_j) in the graph. This also means that we have not investigated all the shortest paths from all the known edges in the graph to the nodes o_i and o_j. However, from each edge of the unknown edge, from o_i and from o_j, we find all pairs shortest paths. In subsequent steps, the algorithm goes over each of the known edges in sequence assuming that edge it longest in its shortest path and subtracting the shortest path from its length. From Equation 4 for lower bounds, and by going over the shortest paths through known edges, we have investigated all the shortest paths through all available known edges implicitly. This contradicts our assumption that we did not investigate all the known edges in the graph, thus proving the tightness of the lower bound produced. □

For efficiency, we can package both the upper and lower bound algorithms as a single algorithm. The details of the algorithm are given in Algorithm 1 as SPLUB.

Running Time Analysis for SPLUB

We shall show here, that the running time of SPLUB depends on the sparsity of the underlying graph $\mathcal{G}$.

Upon examination, it is clear that the step 2 and step 3 are the time consuming steps which are the execution of shortest path algorithms from both endpoints of the unknown edge. Dijkstra's

Algorithm [17] with its standard implementation taken $O(m + n \log n)$ time to run. Thus we estimate the overall running time of the combined steps 2 and 3 as $O(m + n \log n)$.

The remainder of the steps, step 6 and step 7, are executed only as the number of known edges in the graph, m. Thus the total time of SPLUB is estimated as $O(m + n \log n) + O(m)$. The leading term is the first term and thus we claim the overall running time of the algorithm to be $O(m + n \log n)$.

Update Algorithm - Given the simplicity of the algorithm and absence of any intermediate data structures, updates are rather straight forward in SPLUB Scheme. Once a previously unknown edge is resolved, the only data-structure that needs an update is the underlying graph structure. The update to the graph data-structure, in any representational format (adjacency list or adjacency matrix), is a constant order operation, thus, obtaining the overall complexity of update operation as $O(1)$.

4.2 Approximate Algorithms

In this section we strive to answer the following questions: *Can one design solutions that produce* not the tightest bounds*, yet are highly scalable and faithfully produce the exact solutions to the proximity problems? Can we design efficient and effective data structure update schemes supporting the approximate bounds?*

Let us assume that instead of going through all the known edges in the graph and their shortest paths, we only restrict ourselves to a subset of the known edges. From Equation 4, it is evident that the bound obtained will not be tight. As an example, considering only the path $[o_1 \rightarrow o_3 \rightarrow o_2 \rightarrow o_4]$ to compute the LB of (o_1, o_4), the lower bound will be 0.5.

An important point to note here is that, to develop practically viable algorithms, the developed solution must avoid the following two bottlenecks (i) the Shortest Path computations, (ii) exploration of all the known edges in the graph. We propose a highly scalable yet effective heuristic Tri Scheme to that end.

4.2.1 Triangle Induced Solution Scheme. The overall idea of Tri Scheme is to restrict ourselves to small neighborhoods (in particular triangles) and use the relationship imposed by the triangles in producing bounds.

Upper and Lower Bounds: Basically, Tri Scheme looks at every triangle between o_i and o_j and computes lower and upper bounds. However finding every triangle which are incident on the unknown edge (o_i, o_j) and whose other two sides are known is also computationally challenging. To further explain, we wanted to find out all $\triangle_{o_i,o_j,o_l}$, where (o_i, o_l) and (o_j, o_l) are known, solving the bounds problem efficiently.

Updates: As seen above, in Tri Scheme, for answering queries, we need to access the triangles, whose two sides (edges) are known and the edge being queried is the only missing edge. We use an adjacency list representation of the graph to speed up the search for such triangles. We take the lists corresponding to two endpoints of the unknown edge o_i and o_j, and find their intersection to find such triangles. Finding intersections of two lists by direct comparisons are in the *O(size of the list)*. In adjacency list corresponding to each node in the graph, we use a balanced binary search tree[14] to make comparisons faster. However, this scheme has increased the new edge insertions to be in $O(\log(n))$, which updates two binary search trees one corresponding to each endpoint of the resolved edge in the adjacency list with the edge value.

Algorithm 2 Tri Scheme

Input : graph $\mathcal{G}$ = (O,E), unknown edge (o_i, o_j)
Output : $LB^{d(o_i,o_j)}, UB^{d(o_i,o_j)}$

```
1:  adj_i = AdjacencyList(o_i)
2:  adj_j = AdjacencyList(o_j)
3:  lb = 0
4:  ub = 1
5:  while i ≤ len(adj_i) and j ≤ len(adj_j) do
6:      if adj_i[i] == adj_j[j] then
7:          lb = max(lb, |E[o_i, adj_i[i]] − E[o_j, adj_j[j]]|)
8:          ub = min(ub, E[o_i, adj_i[i]] + E[o_j, adj_j[j]])
9:          i, j = (i + 1), (j + 1)
10:     else
11:         if adj_i[i] > adj_j[j] then
12:             j = j + 1
13:         else
14:             i = i + 1
15:         end if
16:     end if
17: end while
18: LB^{d(o_i,o_j)} = lb
19: UB^{d(o_i,o_j)} = ub
20: return LB^{d(o_i,o_j)}, UB^{d(o_i,o_j)}
```

Further Illustration - Tri Scheme. Consider unknown edge (o_3, o_5) and Algorithm 2, Tri Scheme, to understand how LB and UB of (o_3, o_5) are produced. The algorithm looks up the corresponding adjacency lists, adj_3 and adj_5 from the graph. From adj_3 and adj_5, it iteratively finds the common endpoints in both lists, here, o_1 and o_2. For each of such endpoints, which forms a triangle with o_3, o_5, it computes the lower and upper bounds using triangle inequalities. 0.6, which is the *max* of all lower bounds obtained from each endpoints is the $LB^{dist(o_3,o_5)}$. Similarly, 0.6, the *min* of all the upper bounds obtained is returned as the $UB^{dist(o_3,o_5)}$.

One can note from pseudo-code of Tri Scheme, Algorithm 2, that, the computational bottleneck from SPLUB are entirely avoided to generate a simpler and practical algorithm. We present some theoretical properties of Tri Scheme next.

4.2.2 Expected Case Analysis for Tri Scheme.

Theorem 4.2. *Expected running time for* Tri Scheme *to lookup an edge is* $O(m/n)$

Proof. By design, the algorithm Tri Scheme is proximity algorithm agnostic. Thus, it works for any general metric space proximity problems. The proximity algorithm can choose any edge and query for the upper and lower bounds. The expected time to lookup an edge can be written as,

$$E[time] = \sum_{(u,v) \in E'} P[sample\ (u,v)] * lookup(u,v)$$

where the probability is for the event of sampling the unknown edge (u, v) and *lookup* represents the amount of time taken by Tri Scheme for looking up the bounds for edge (u, v).

Under the assumption that any one of the unknown edges could be queried next with equal probability (uninformed prior) by the proximity algorithm, there is a uniform probability of sampling any of the unknown edges. Hence, the probability of looking up any of the unknown edge is $1/(n^2 - m)$. `Tri Scheme` uses a balanced *BST* in order to perform set intersection and needs to go over all the edges incident on both u and v to obtain the bounds on edge (u, v). Hence the time taken for resolving the bounds for the edge (u, v) is $d_u + d_v$ where d_u stands for the degree of edge u.

By making use of the above formulation in the expectation formula,

$$E[time] = \sum_{(u,v)\in E'} \frac{1}{n^2 - m}(d_u + d_v)$$

For every missing edge that is incident on u, d_u is added to the expected time. There a $n - d_u$ number of unknown edges incident on u. Hence, the expectation amounts to,

$$E[time] = \sum_{i=1}^{n} \frac{1}{n^2 - m} d_i(n - d_i) = \sum_{i=1}^{n} \frac{nd_i - d_i^2}{n^2 - m}$$

To create an adversarial case, we would like to maximize the above formula to obtain an upper bound on the expected time. Also, we know that there are a total of m known edges and hence the total sum of degrees should amount to $2m$. Hence, the constraint $\sum_{i=1}^{n} d_i = 2m$, needs to be satisfied.

$$maximize \sum_{i=1}^{n} \frac{nd_i - d_i^2}{n^2 - m} \quad s.t.\ 2m = \sum_{i=1}^{n} d_i$$

The expected time is maximized when the negative term, d_i^2 is minimized. As the constraint $2m = \sum_{i=1}^{n} d_i$ exists, the term d_i^2 is minimized when $d_i = 2m/n$. Hence,

$$E[time] = \sum_{i=1}^{n} \frac{nd_i - d_i^2}{n^2 - m} \leq \frac{2nm - \sum_{i=1}^{n} (2m/n)^2}{n^2 - m} = \frac{2nm - 4m^2/n^2 \sum_{i=1}^{n} 1}{n^2 - m}$$

$$E[time] \leq \frac{2nm - 4m^2/n}{n^2 - m} = \frac{4m}{n} \frac{n^2/2 - m}{n^2 - m}$$

Replacing $n^2/2$ with n^2 in the numerator, we get,

$$E[time] \leq \frac{4m}{n} \frac{n^2/2 - m}{n^2 - m} \leq \frac{4m}{n} \frac{n^2 - m}{n^2 - m} = \frac{4m}{n} \in O(\frac{m}{n})$$

Thus, proved. □

Bootstrapping `Tri Scheme` through Landmarks: In Section 4.2, we have developed the `Tri Scheme`, a scalable algorithm. Our goal here is to study how `Tri Scheme` could be designed in conjunction with landmark based solutions, such as, `LAESA` [36] to bootstrap `Tri Scheme`. Landmark based solutions, as described in Related Works is a pivot based solutions that use a specified number of nodes and resolve the distances between them to obtain a tighter bound on the rest. Recall our problem setting described in Section 3.1 that assumes m edges are resolved at the beginning of the algorithm. We use an initialization of the graph $\mathcal{G}$ by bootstrapping it with `LAESA` inside every proximity algorithm, for different values of m. Later in the experiment section, Section 5 we shall show the effectiveness of our schemes due to this initialization.

5 EXPERIMENTAL EVALUATION

Algorithms are developed in Python 3.6 and the experiments are conducted on an Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz running a Linux distribution, Ubuntu 18.04.5 LTS using 64 GB. Our code and data can be found at https://github.com/jeesaugustine/metric-space-proximity-algo/.

5.1 Experimental Setup

5.1.1 Datasets. We use 3 real datasets, vary different parameters considering various proximity problems, summarized in Table 1. The actual pairwise distances (i.e., ground truth) are known.

5.1.2 Implemented Baselines. (1) We implement `ADM` [42] algorithm which provides the exact upper, lower bounds and updates. We refer to the baseline algorithm as, `ADM`. (2) We implement landmark based algorithm `LAESA` [36]. (3) We implement `TLAESA` [35], a follow up work of `LAESA`.

These baselines are compared with (i) DFT in Section 2.2, (ii) `SPLUB` in Section 3 and, (iii) `Tri Scheme` in Section 4.2. We use $k = \log(n)$ landmarks unless otherwise mentioned.

5.1.3 Proximity Algorithms. We consider 3 classes of metric space proximity problems (i) kNNG construction, (ii) Minimum Spanning Tree Construction (MST) and, (iii) Clustering and evaluate how they could benefit from our proposed approach in saving distance computation and overall cost wrt multiple competitors.

(i) **k Nearest Neighbor Graph (k-NNG) Construction:** We implement `KNNrp`, a popular and recent k-NNG proposed in [39] that computes the k-NNG of a given set of objects. (ii) **MST:** We implement the popular *Prim*'s [17] and *Kruskal*'s [29] algorithm for evaluation. (iii) **Clustering:** We implement two popular centroid based swapping algorithms, *PAM* [26] and *CLARANS* [38].

5.1.4 Experimentation Goals. **(Subsection 5.3)** Analyzing Direct Feasibility Test (through `Prim's` Algorithm) to demonstrate its effectiveness and limitations.
(Subsection 5.4) Comparison between proposed graph theoretic techniques (`SPLUB` and `Tri Scheme`), and compare with `ADM` and `LAESA` and `TLAESA`, on the following parameters. (i) Quality of bounds and, (ii) In computation time.
(Subsection 5.5) Comparison between `Tri Scheme` and `LAESA` and `TLAESA`, in saving distance calls for various proximity algorithms.
(Subsection 5.6) Comparison between `Tri Scheme`, `LAESA`, `TLAESA`, and the original algorithm in overall running time by varying the cost of distance oracle.
(Subsection 5.7) Varying proximity algorithms parameters l and k and its effect on CPU overhead and Distance Calls.

5.1.5 Evaluation Measures. Our main investigation here is to study how `SPLUB` and `Tri Scheme` compare with `ADM` and `LAESA` in producing distance bounds, as well as their effectiveness in saving distance calls and overall running time inside different proximity algorithms. [1] *Relative Error & CPU Overhead*: We present relative error of the produced bounds of different algorithms wrt `ADM`. CPU

Dataset	Algorithm	Num. Objects	Num. of Edges	Dimension	Distance Function
SF POI	k-NNG, Clustering, MST	21, 048	221, 498, 628	2	Google Maps API
Flicker1M	k-NNG	$10k$	49995000	256	Eucledian
UrbanGB	k-NNG, Clustering, MST	360, 177	64, 863, 555, 576	2	Google Maps API

Table 1: Dataset Description

overhead is captured as the difference between the total time and the total distance oracle time. 2 *Percentage Save-ups*: We compute the percentage of the distance calls save-up of our algorithms wrt the baselines. 3 *Proximity Algorithm Completion Time*: We capture the overall running time of the proximity algorithms after they are augmented with SPLUB, Tri Scheme, LAESA and TLAESA.

Additionally, we present some deeper analysis that compares Tri Scheme with LAESA and ADM qualitatively and running time-wise.

Please note here that, for brevity, we only present a subset of the results that are representative.

5.2 Summary of Results

(1) Consistent with our theoretical analysis, Direct Feasibility Test provides the largest savings in distance calls, better than the best known algorithm (ADM) (2× improvement even for moderate size graphs). However, this formulation is computationally expensive, as discussed in Section 2.2, and thus, is limited to graphs of smaller sizes, which are in the order of a few hundreds of edges.
(2) On every proximity algorithm SPLUB produces *the exact* bounds as ADM, while being significantly faster in computation time than ADM. Consequently, the number of distance calls solicited is identical to that of ADM but takes less time to complete. While ADM only runs in smaller graphs (in the order of a few thousand edges), SPLUB can be scaled easily to moderate sized graphs (a few hundred thousand edges). ADM, a cubic algorithm, requires more than 2× more time than that of SPLUB.
(3) **Tri Scheme** turns out to be the unanimous choice to be used in large scale third party applications where distance calls are expensive. Tri Scheme runs significantly faster in running time compared to SPLUB with comparable quality of bounds. Compared to LAESA and TLAESA, Tri Scheme produces tighter bounds at the expense of a marginal increase in CPU time. While the actual number of calls saved depends on the proximity algorithm, under all settings and algorithms, Tri Scheme saves more distances compared to LAESA and TLAESA. On average, Tri Scheme saves 42%($\approx 2.4\times$) and 36% distance calls compared to LAESA and TLAESA across datasets and across the algorithms. However, for Prim's using UrbanGB, the savings go up to 70% and beyond for very large graphs with 33M edges for LAESA and 62% for TLAESA. When used inside proximity algorithms, Tri Scheme on an average takes only 40% of the time compared to LAESA and 33% of the time compared to TLAESA making it an excellent choice to be used as a plug-in. In some cases, proximity algorithms using Tri Scheme take half the time of that of using TLAESA.

5.3 Direct Feasibility Test (DFT)

We implemented DFT using the linear programming solver CPLEX[3] through its Python API and integrated inside different proximity algorithms. For brevity, we include results from the Prim's algorithm. We compare the results with the exact state-of-the-art solution ADM. The distance oracle considered for these experiments consumes 10^{-5}s for obtaining a pairwise distance.

To illustrate concretely, for a graph with 45 edges, a variable is defined per edge. Since each edge is normalized between $[0, 1]$, this process adds 90 linear inequalities to satisfy the ranges. Additionally, for each triangle involving 3 edges, we add two constraints to satisfy the triangle inequality. In summary, the total number of constraints for solving DFT for a graph of 45 edges involve satisfying 450 linear constraints. Clearly, DFT could only be implemented on very small graphs and the results are presented in Section 2.2.

Figures 4a and 4b present the average of 10 runs of Prim's algorithm and compare DFT with ADM. Figure 4a exhibits that DFT consistently requires a smaller number of distance calls compared to ADM. With an increase in the number of edges, the percentage of distance calls savings increases consistently for DFT(between 27% and 58% empirically).

On the other hand, Figure 4b presents the CPU time of running Prim's using DFT. For graphs with 325 edges and 496 edges, it takes about 3 hours and more than 8 hours, respectively. The computational bottleneck lies in satisfying a massive number of linear constraints during each run of DFT , limiting its use as a practical solution in real-world settings.

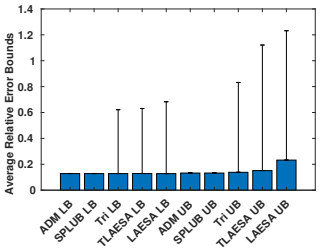
To summarize, these experiments corroborate our theoretical analysis, (1) DFT is an alternative formulation to the problem, produces tightest bounds, and outperforms ADM in saving distances (43% on an average). (2) DFT incurs substantial CPU cost and is not scalable to large graphs (takes 1.6 hours on an average for graphs with a few hundreds of edges).

5.4 Tightness of Bounds and Running Time

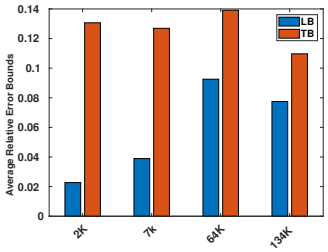
Figure 3a demonstrates SPLUB produces the exact same bounds (upper and lower) as ADM and the error bar is virtually collapsed. Bounds produced by Tri Scheme(Figure 3b) are looser than ADM , however much tighter compared to LAESA and TLAESA. These results corroborate that the Tri Scheme is a practical yet viable solution for our studied problems. Figure 3c and Figure 5a provide additional insights: ADM , even though produces the best bounds, is not scalable. Neither SPLUB nor ADM is suitable for larger graphs in terms of CPU overhead. Figure 5a shows that even though LAESA is the fastest among all the algorithms, the relative error is much higher(Figure 3a).

5.4.1 Limitation of LAESA *and* TLAESA. Limitations of LAESA and TLAESA, discussed in Section 4.2 are experimentally corroborated in figure 5b. We experimentally observe that the optimal value of the number of landmarks for LAESA and TLAESA for Prim's algorithm using SF dataset 1.9M configuration is $3 \times \log(n)$. But this varies largely across datasets and proximity algorithms and there is no obvious way to determine this parameter. Additionally, TLAESA maintains a tree data-structure which aids in estimating

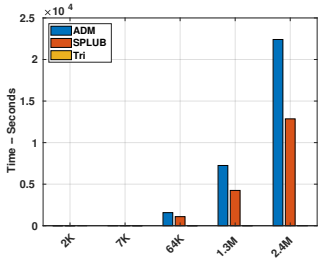
[3]https://www.ibm.com/products/ilog-cplex-optimization-studio/



(a) [SF 135K][Bounds] Relative Error is 0 for SPLUB with ADM & for Tri Scheme lesser than LAESA & TLAESA, especially UB.

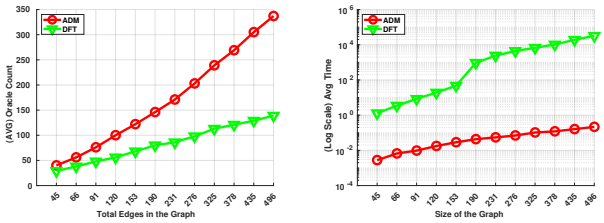


(b) [SF] Tri Scheme Bounds Varying # Edges. With increasing edges, (LB-UB) gap reduces drastically(3.3× across 2k & 134k)



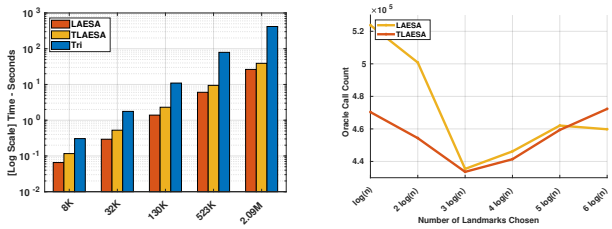
(c) [SF] Time: ADM is not scalable. SPLUB produces same bound with 50% of time. Tri Scheme improves time by 99% (highest: 23s)

Figure 3: Bounds Comparison for UB & LB (a) Relative Error 0 for SPLUB with ADM and for Tri Scheme much lesser than LAESA



(a) [Prim's] Distance calls of DFT and ADM (b) [Prim's] [Logscale] Running time of DFT and ADM

Figure 4: DFT vs ADM distance calls & run time comparison



(a) [SF] fast but loose bounds of LAESA and TLAESA (b) [SF] ideal # landmarks selection problem in LAESA and TLAESA

Figure 5: Limitations of LAESA, TLAESA

the upper and lower bounds, however, the construction of which incurs additional distance computations.

5.5 Tri Scheme for Distance Counts

In this subsection, we turn the attention to our practical approach, Tri Scheme, and study how it saves distance calls inside various proximity algorithms wrt baselines LAESA and TLAESA.

These experiments confirm our previous findings. Proximity algorithms augmented with Tri Scheme *shows significant improvement in saving the distance calls when compared with the other two baselines. We also note that as the size of the dataset grows the gap between the number of calls made widens in the context of all proximity algorithms.*

We compare our results against the empirically found the best (lowest) count for distance calls in LAESA and TLAESA.

5.5.1 Evaluation of MST Algorithms. We compare the classical Prim's and Kruskal's algorithms for the MST problem with their augmented versions through Tri Scheme varying number of objects. Table 2 and Table 3 present comprehensive results.

Column 'TS-NB' represents the number of oracle calls for the completion of Prim's for Tri Scheme with no bootstrap. Column 'Bootstrap' represents the number of oracle calls for bootstrapping with LAESA. The percentage saving for the Tri Scheme in completion of Prim's with bootstrap wrt LAESA and TLAESA are given in corresponding Save(%) columns. The number of landmarks used for bootstrapping can be found within parenthesis.

Save-ups is increased with increasing size of the datasets, shown in bold as a percentage of distance calls saved in Tri Scheme compared to LAESA and TLAESA, demonstrating the efficacy of Tri Scheme. TS-NB outperforms LAESA and TLAESA always. While TS-NB performs better than Tri Scheme in many cases, there are certain configurations where the opposite is true in Table 3.

Figure 6a represent the distance save up. It is interesting to note that, proximity algorithms, in general, are sensitive to the total number of pairwise distances. The efficacy of Tri Scheme in saving the distance calls is evident in both the figure and Tables.

5.5.2 Evaluation of Clustering. We compare the two l-medoid ($l = 10$) algorithms PAM and CLARANS with their augmented versions with Tri Scheme and compare with the baselines. Overall, algorithms augmented with Tri Scheme use on average one third the number of distance calculations.

Figures 6c, 6d, 7a, 7b, and 7c exhibit that, as the number of objects grows, the number of distance calculations increases, and the save-up for Tri Scheme compared with baselines also grows.

We observe the maximum saving up of 36%(20%) for SF and a save up of 55%(43%) for the UrbanGB datasets for LAESA(TLAESA). We also note that the perceived large running time of PAM is due to its inherent nature, and not due to Tri Scheme.

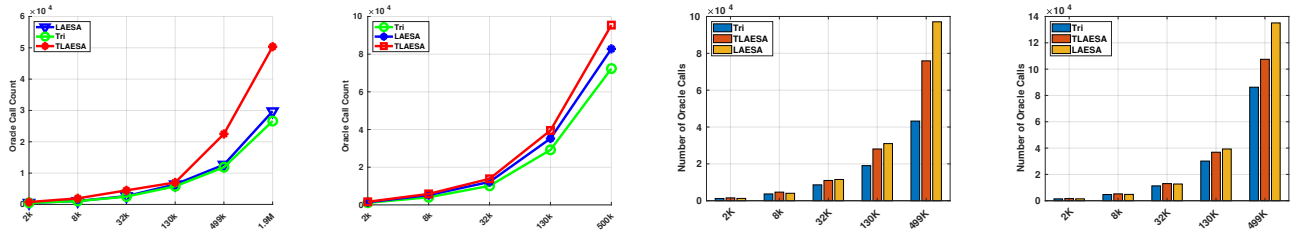
5.5.3 Evaluation of k-NNG. The objective of this set of experiments is to compare the vanilla KNNrp [37] ($k = 5$) with the KNNrp augmented by the algorithmic scheme, Tri Scheme developed in this work in saving distance calls. Figure 6b describes the number of

UrbanGB Dataset [Oracle Call Count]								
Prims Algorithm [$k = \log_2(n)$]								
# of Edges	Without Plug	TS-NB	Bootstrap	**Tri Scheme**	LAESA	**Save (%)**	TLAESA	**Save (%)**
2016	2016	916	363	999 (6)	1097 (6)	**8.93**	1184	**15.63**
8128	8128	2819	868	2980 (7)	3343 (7)	**10.86**	3583	**16.83**
32640	32640	9454	2012	10017 (8)	15123 (18)	**33.76**	12718	**21.24**
130816	130816	29043	4563	30045 (9)	59619 (27)	**49.60**	38302	**21.56**
499500	499500	82419	9945	86199 (10)	160306 (30)	**46.23**	117906	**26.89**
1999000	1999000	259237	21934	280004 (11)	606517 (22)	**53.83**	462207	**39.42**
7998000	7998000	779707	47922	800985 (12)	2198589 (24)	**63.57**	1650752	**51.48**

Table 2: # of expensive Oracle Calls by Prim's Algorithm with TS-NB, Tri Scheme , LAESA and TLAESA along with parameters

SF Dataset [Oracle Call Count]								
Prims Algorithm [$k = \log(n)$]								
# of Edges	Without Plug	TS-NB	Bootstrap	**Tri Scheme**	LAESA	**Save (%)**	TLAESA	**Save (%)**
2016	2016	1216	363	1230 (6)	1408 (6)	**12.64**	1408	**12.64**
8128	8128	3681	868	3670 (7)	3813 (7)	**3.75**	4238	**13.40**
32640	32640	11966	2012	12081 (8)	13212 (8)	**8.56**	14102	**14.33**
130816	130816	40115	4563	40547 (9)	48317 (18)	**16.08**	46835	**13.43**
499500	499500	138179	9945	143122 (10)	182600 (40)	**21.62**	171003	**16.30**
1999000	1999000	372863	21934	384059 (11)	542937 (33)	**29.26**	499760	**23.15**
7998000	7998000	1326373	47922	1399769 (12)	2298327 (48)	**39.10**	2049026	**31.69**

Table 3: # of expensive Oracle Calls by Prim's Algorithm with TS-NB, Tri Scheme , LAESA and TLAESA along with parameters



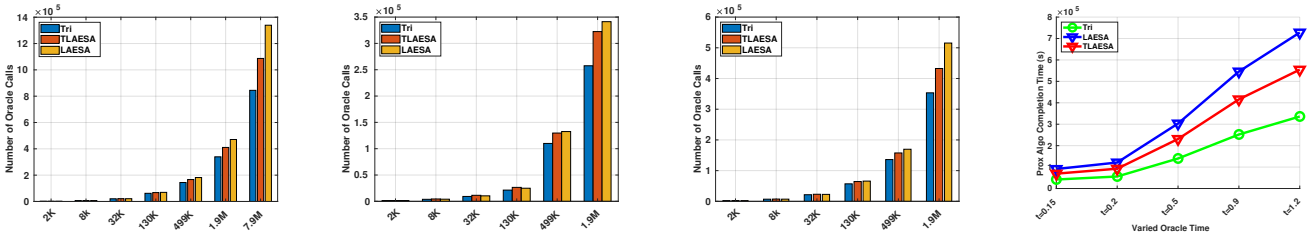
(a) [UrbanGB] Kruskal's distance save varying dataset size(47% save - TLAESA & 10% - LAESA for 1.9M).

(b) [UrbanGB] KNNrp distance save. Tri Scheme bounds match SPLUB bounds

(c) [UrbanGB] PAM Vary Size - With increasing dataset size, distance save(%) increases for Tri Scheme

(d) [SF] PAM Vary Size. With increasing dataset size, thedistance save % increases for Tri Scheme

Figure 6: Number of Expensive Oracle Calls for completion of Kruskal's, KNNrp and PAM Algorithms



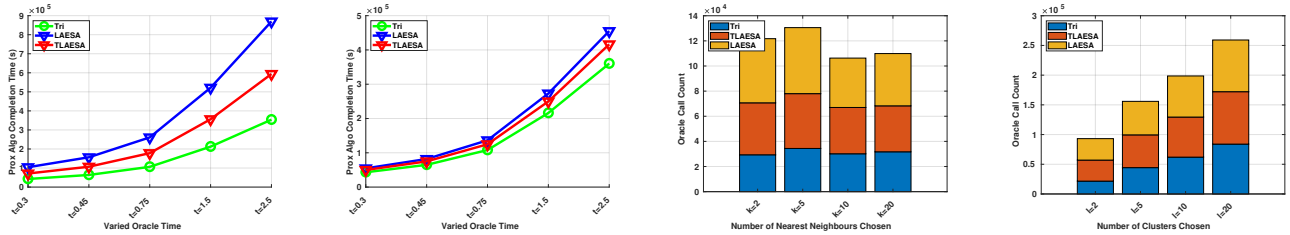
(a) [SF]CLARANS Var Size. Scale to large graphs(no save compromise) 36%(22%) save in LAESA(TLAESA)

(b) [Flicker1M] PAM Varying Size. 20%(24%) save up in largest setting for LAESA(TLAESA)

(c) [UrbanGB] CLARANS Algorithm Varying Size. 31%(18%) save up for LAESA(TLAESA)

(d) [UrbanGB 1.99M][Prox Time] Prim's 53%(39%) save up LAESA(TLAESA) for 1.2s oracle

Figure 7: (a,b,c)Number of Expensive Oracle Calls For Algorithms PAM and CLARANS (d) Completion Time of Prim's



(a) [Flicker1M 1.99M] [Prox Time] 59% (40%) save for LAESA(TLAESA) **(2.5s oracle) for completing** PAM

(b) [SF 1.99M][Prox Time] 20% (14%) save for LAESA(TLAESA) **2.5s oracle on completion of** CLARANS

(c) [SF 130K][Oracle Count] Stability of `Tri Scheme` **to variance of l in** PAM

(d) [SF 130K][Oracle Count] Stability of `Tri Scheme` **to variance of l in** CLARANS

Figure 8: (a,b)Actual Proximity Algo Completion Time for PAM **&** CLARANS **varying distance oracle cost (c,d) Effect of varying l in** PAM **&** CLARANS **on distance counts.** CLARANS **improves dist calls but stability across l is limited by choices in randomized search**

distance calls made by the algorithm. The findings are similar to other proximity algorithms.

5.6 `Tri Scheme` for Running Time

We present the end-to-end completion of the proximity algorithms, when augmented by `Tri Scheme`, LAESA, and TLAESA by varying the cost of distance computation. *We observe that the overhead induced by our algorithms, are nominal when compared with the results from* LAESA *and* TLAESA. *However, when induced with expensive oracle calls, owing to a large number of distance calls, the time spent in completion is higher than the baseline algorithms.*

5.6.1 Evaluation of MST Algorithms - Time. Figure 7d presents the overall time taken by `Prim's` varying oracle time. `Tri Scheme`, on an average, saves time by 53%(only takes half the time) compared to LAESA and 39% compared to TLAESA, even when each distance computation takes 1.2s.

5.6.2 Evaluation of Clustering - Time. Figures 8a and 8b show the running time of PAM and CLARANS ($l = 10$) in conjunction with `Tri Scheme`, LAESA, and TLAESA varying oracle cost.

The overall time save-up of `Tri Scheme` is 39% wrt LAESA and 26% wrt TLAESA. The savings for PAM goes up to 59% with LAESA and 40% with TLAESA for an oracle of 2.5 seconds.

5.6.3 Evaluation of k-NNG - Time. Finally, we take the k-NNG ($k = 5$) problem using Urban dataset with 1.99M settings. These results indicate that by leveraging triangle inequality `Tri Scheme` outperforms the baselines.

5.7 Varying Proximity Parameters

Proximity algorithms are sensitive to their parameters of choice. Clustering algorithms like PAM and CLARANS are required to accept the number of clusters (l) as a part of their inputs. Similarly, kNNG needs the number of neighbours, k.

5.7.1 Clustering (varying l) - Count, CPU overhead. Here we vary l and compare the number of distance calls of `Tri Scheme` against LAESA and TLAESA. Figure 8c presents the results of the PAM algorithm. As the number of objects is fixed, increasing the number of clusters results in more local minima for the PAM algorithm which in turn makes the algorithm converge faster. Figure 8d presents the results from the CLARANS algorithm which shows as the number of clusters increases, the number of distance calls also increases.

Figure 9b and Figure 9c show the CPU overhead for PAM and CLARANS algorithms respectively. As expected, when l increases, we see an increase in the CPU overhead in response to the number of additional upper and lower bound comparisons.

5.7.2 k-NNG (varying k)- Count and CPU overhead. We present the results by varying k for `KNNrp` algorithm here. Figure 9a shows that the number of distance estimations increases with increasing k, as the algorithm needs to resolve more candidates to determine the nearest neighbours. Figure 9d shows the same effect in CPU overhead, as described in Section 5.7.1.

6 RELATED WORK

Our work mainly focuses on *three* key aspects. (1) The objects are atomic (are not a collection of vectors) and defined in general metric spaces. (2) The distance computation between pairs of objects is expensive and a leading cost. (3) Designed solutions could be integrated inside a variety of proximity problems and return *exact answers* as that of the original algorithms.

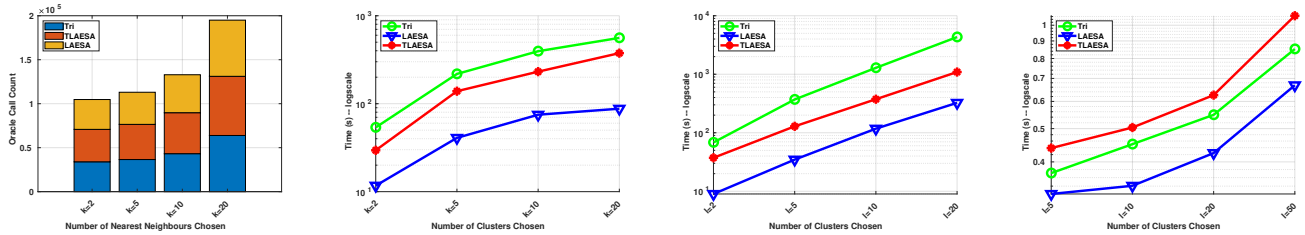
The related work could be broadly classified into one of the three kinds and our work fits the first one.

6.1 Metric Space Based

The state-of-the-art solution is proposed by Sasha and Wang [45] that develops ADM to produce tightest upper and lower bounds of distances that are then demonstrated to save up different querying cost. *However, the computational cost of* ADM *is $O(n^3)$, making it an impractical choice for repeated invocation.*

Landmark based algorithms store partial information of nodes in an array form to answer nearest neighbour queries in metric space. The representative algorithms are AESA, LAESA [36, 41], TLAESA [35] and variants of TLAESA [21, 34] which extend the idea of pivot based methods in various ways. In AESA, all pairwise distances are precomputed and stored in a matrix. In LAESA, an extension to AESA, a set of base prototypes are chosen and all the pairwise distances between them and reminder of the objects are stored. In TLAESA and its variants, in addition to selected base prototypes, the algorithms maintain a tree data-structure to expedite the nearest neighbor search.

Pivot based algorithms select a set of pivots to divide the space into smaller sub-spaces, grouping similar objects. BKT [10], a pivot based data structure, is designed for similarity search which recursively builds a tree based on the distance to other objects. FQT,



(a) [SF 130K][Oracle Count] Stability of `Tri Scheme` to variance of l in `KNNrp`

(b) [SF 130K][Time] Local CPU Computation Impact of `Tri Scheme` to variance of l in `PAM`

(c) [SF 130K][Time] Local CPU Computation Impact of `Tri Scheme` to variance of l in `CLARANS`

(d) [SF 130K][Time] Local CPU Computation Impact of `Tri Scheme` to variance of l in `KNNrp`

Figure 9: (a) Effects of varying $l(k)$ on the number of distance calls for `KNNrp` (b,c,d) local CPU overhead on varying $l(k)$ (disconnecting the problems of distance compute and CPU compute and reducing distance compute (↓) at the expense of CPU compute (↑)) for `PAM` , `CLARANS` & `KNNrp`

FHQT [2], and FQA [11] are follow up works that offer improvements to this. Vantage Point Tree(VPT) [47] index structure solves the problem of nearest neighbor queries in general metric space.

Voronoi diagrams, commonly used in proximity queries in vector spaces, have inspired data structures in metric spaces, namely GNAT [9] and M-tree [13]. GNAT [9] introduces an indexing structure for nearest neighbour queries in large metric spaces. M-Tree [13] is a balanced tree indexing structure for metric space similarity search and the *k-NN* problems. M-Tree, which works by partitioning the space, is built in a bottom-up manner, and has a fixed number of objects in each node, giving rise to a balanced structure.

These aforementioned related works focus to reduce overall CPU time and do not distinguish between distance computation cost and the CPU time, unlike the focus of our research. Moreover, landmark based require # landmarks as inputs, and these solutions are specifically designed for the nearest neighbor search problems, thus, they do not easily generalize to all proximity problems. Both ADM and landmark based solutions are adapted to be used as baselines in Section 5.

6.2 Vector Space Based

Vector Space Based Methods use the coordinate information of the objects to create data structures to answer a large spectrum of distance queries, where distance may be based on Euclidean, cosine similarity, general L_p norms, and so on. Popular solutions in low to moderate dimensional space include *K-B-D-tree* [40], *kd-tree* [5], *R-tree* [19], *R*-tree* [4], *SS-tree*[46] or more recent *X-tree* [6], *UB-tree*[3], *SR-tree* [25]. All these methods use the domain object feature vectors to measure the distance between objects and create a similarity index. These indexes are primarily built to answer the similarity queries.

In high dimensions, to address the curse of dimensionality, various randomized and approximation techniques have been proposed, including Locality Sensitive Hashing [18, 23] and Locality Preserving Hashing [24, 49], both bucketing similar objects with high probability, and Random Projections [30, 31], which projects high dimensional objects to a low dimensional projection to enable similarity search.

These approaches are specifically designed for vector space proximity problems, and do not adapt to general metric spaces, as objects in general metric spaces do not have conveniently available co-ordinates or features/dimensions - hence cannot be modelled as vectors. Moreover, the high dimensional approaches produce approximate answers, whereas, our work focuses on returning exact answers for proximity problems.

6.3 Metric space Transformed into Vector Space

Embedding Spaces commonly used in transforming the given set of objects in metric space to a vector space is another common approach. Metric Embedding [33, 44] and Multidimensional Scaling [16] are some of the representative techniques in that space. *After transformation, these methods produce approximate distances between objects, leading to approximate answers for the proximity problems. In contrast, our focus is to return exact answers.*

7 CONCLUSION AND DISCUSSIONS

In this work, we propose a suite of principled solutions that trade-off between tightness of the produced bounds and computational time to minimize distance computation cost for various proximity problems in general metric spaces. Our proposed algorithms range from expensive linear inequality based exact bounds, to graph theoretic approaches producing exact and approximate bounds. However, our proposed techniques, when used inside the proximity algorithms always return exact results. We compare our designed solutions conceptually and empirically wrt a broad range of existing works through comprehensive experimentation.

We believe that our proposed framework adapts to more sophisticated optimization problems, related to graph partitioning, facility allocation, traveling salesman problems, just to name a few. The idea would be to substitute expensive distance comparison within these algorithms by our proposed upper and lower bound computation techniques to see if that serves the purpose. We intend to study these aspects in the future.

ACKNOWLEDGMENTS

The research of Gautam Das was supported in part by grants #2008602, #1745925, and #1937143 from the National Science Foundation. The work of Mohammadreza Esfandiari and Senjuti Basu Roy are supported in parts by the National Science Foundation grants #1942913, #2007935, #1814595, and by the Office of Naval Research Grant No:N000141812838.

REFERENCES

[1] Irwin E Alber, Ziyou Xiong, Nancy Yeager, Morton Farber, and William M Pottenger. 2001. Fast retrieval of multi-and hyperspectral images using relevance feedback. In *IGARSS 2001. Scanning the Present and Resolving the Future. Proceedings. IEEE 2001 International Geoscience and Remote Sensing Symposium (Cat. No. 01CH37217)*, Vol. 3. IEEE, 1149–1151.

[2] Ricardo Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. 1994. Proximity matching using fixed-queries trees. In *Annual Symposium on Combinatorial Pattern Matching*. Springer, 198–212.

[3] Rudolf Bayer. 1997. The universal B-tree for multidimensional indexing: General concepts. In *International Conference on Worldwide Computing and Its Applications*. Springer, 198–209.

[4] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. 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*. 322–331.

[5] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[6] S Berchtold, D Keim, and HP Kriegel. 1996. The X-tree: An efficient and robust access method for points and rectangles. In *Proc. 1996 Int. Conf. Very Large Data Bases*. 28–39.

[7] Andrew P Berman and Linda G Shapiro. 1998. Triangle-inequality-based pruning algorithms with triangle tries. In *Storage and Retrieval for Image and Video Databases VII*, Vol. 3656. International Society for Optics and Photonics, 356–365.

[8] Andrew P Berman and Linda G Shapiro. 1999. A flexible image database system for content-based retrieval. *Computer Vision and Image Understanding* 75, 1-2 (1999), 175–195.

[9] Sergey Brin. 1995. Near neighbor search in large metric spaces. (1995).

[10] Walter A. Burkhard and Robert M. Keller. 1973. Some approaches to best-match file searching. *Commun. ACM* 16, 4 (1973), 230–236.

[11] Edgar Chávez, José L Marroquín, and Gonzalo Navarro. 2001. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications* 14, 2 (2001), 113–135.

[12] S-S Cheung and Avideh Zakhor. 2005. Fast similarity search and clustering of video sequences on the world-wide-web. *IEEE Transactions on Multimedia* 7, 3 (2005), 524–537.

[13] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 1997. M-tree: An E cient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd VLDB conference, Athens, Greece*. Citeseer, 426–435.

[14] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[15] AFW Coulson, JF Collins, and A Lyall. 1987. Protein and nucleic acid sequence database searching: a suitable case for parallel processing. *Comput. J.* 30, 5 (1987), 420–424.

[16] Michael AA Cox and Trevor F Cox. 2008. Multidimensional scaling. In *Handbook of data visualization*. Springer, 315–347.

[17] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[18] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[19] Antonin Guttman. 1984. *R-trees: A dynamic index structure for spatial searching*. Vol. 14. ACM.

[20] Steven Heilman. 2010. THE RADON AND FOURIER TRANSFORMS: THE MATHEMATICS OF X-RAYS AND CT-SCANS. (2010).

[21] Selene Hernández-Rodríguez, J Fco Martínez-Trinidad, and J Ariel Carrasco-Ochoa. 2008. On the selection of base prototypes for laesa and tlaesa classifiers. In *2008 19th International Conference on Pattern Recognition*. IEEE, 1–4.

[22] Daniel P Huttenlocher, Gregory A. Klanderman, and William J Rucklidge. 1993. Comparing images using the Hausdorff distance. *IEEE Transactions on pattern analysis and machine intelligence* 15, 9 (1993), 850–863.

[23] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 604–613.

[24] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. 1997. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 618–625.

[25] Norio Katayama and Shin'ichi Satoh. 1997. The SR-tree: An index structure for high-dimensional nearest neighbor queries. *ACM Sigmod Record* 26, 2 (1997), 369–380.

[26] Leonard Kaufman and Peter Rousseeuw. 1987. *Clustering by means of medoids*. North-Holland.

[27] Leonid G Khachiyan. 1996. Rounding of polytopes in the real number model of computation. *Mathematics of Operations Research* 21, 2 (1996), 307–320.

[28] Victor Klee and George J Minty. 1972. How good is the simplex algorithm. *Inequalities* 3, 3 (1972), 159–175.

[29] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.

[30] Ke Li and Jitendra Malik. 2016. Fast k-nearest neighbour search via dynamic continuous indexing. In *International conference on machine learning*. PMLR, 671–679.

[31] Ping Li, Trevor J Hastie, and Kenneth W Church. 2006. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 287–296.

[32] Tian Liu, Weiyu Xu, Pascal Spincemaille, A Salman Avestimehr, and Yi Wang. 2012. Accuracy of the morphology enabled dipole inversion (MEDI) algorithm for quantitative susceptibility mapping in MRI. *IEEE transactions on medical imaging* 31, 3 (2012), 816–824.

[33] Jirí Matousek. 2002. *Lectures on discrete geometry*. Graduate texts in mathematics, Vol. 212. Springer.

[34] Luisa Micó and Jose Oncina. 2014. Dynamic Insertions in TLAESA Fast NN Search Algorithm. In *2014 22nd International Conference on Pattern Recognition*. IEEE, 3828–3833.

[35] Luisa Micó, José Oncina, and Rafael C. Carrasco. 1996. A fast branch & bound nearest neighbour classifier in metric spaces. *Pattern Recognit. Lett.* 17, 7 (1996), 731–739. https://doi.org/10.1016/0167-8655(96)00032-3

[36] María Luisa Micó, José Oncina, and Enrique Vidal. 1994. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. *Pattern Recognition Letters* 15, 1 (1994), 9–17.

[37] Gonzalo Navarro and Ricardo Baeza-Yates. 1997. Proximal nodes: A model to query document databases by content and structure. *ACM Transactions on Information Systems (TOIS)* 15, 4 (1997), 400–435.

[38] Raymond T. Ng and Jiawei Han. 2002. CLARANS: A method for clustering objects for spatial data mining. *IEEE transactions on knowledge and data engineering* 14, 5 (2002), 1003–1016.

[39] Rodrigo Paredes, Edgar Chávez, Karina Figueroa, and Gonzalo Navarro. 2006. Practical construction of k-nearest neighbor graphs in metric spaces. In *WEA*, Vol. 6. Springer, 85–97.

[40] John T Robinson. 1981. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 10–18.

[41] Enrique Vidal Ruiz. 1986. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters* 4, 3 (1986), 145–157.

[42] Dennis Shasha and Tsong-Li Wang. 1990. New techniques for best-match retrieval. *ACM Transactions on Information Systems (TOIS)* 8, 2 (1990), 140–158.

[43] Larissa Stanberry, Rajesh Nandy, and Dietmar Cordes. 2003. Cluster analysis of fMRI data using dendrogram sharpening. *Human brain mapping* 20, 4 (2003), 201–219.

[44] Kevin Verbeek and Subhash Suri. 2016. Metric embedding, hyperbolic space, and social networks. *Comput. Geom.* 59 (2016), 1–12. https://doi.org/10.1016/j.comgeo.2016.08.003

[45] Jason Tsong-Li Wang and Dennis E Shasha. 1990. Query Processing for Distance Metrics.. In *VLDB*, Vol. 90. 602–613.

[46] David A White and Ramesh Jain. 1996. Similarity indexing with the SS-tree. In *Proceedings of the Twelfth International Conference on Data Engineering*. IEEE, 516–523.

[47] Peter N. Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. In *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, Vijaya Ramachandran (Ed.). ACM/SIAM, 311–321. http://dl.acm.org/citation.cfm?id=313559.313789

[48] Changchuan Yin, Ying Chen, and Stephen S-T Yau. 2014. A measure of DNA sequence similarity by Fourier Transform with applications on hierarchical clustering. *Journal of theoretical biology* 359 (2014), 18–28.

[49] Kang Zhao, Hongtao Lu, and Jincheng Mei. 2014. Locality preserving hashing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 28.

[50] Guus Zoutendijk. 1960. *Methods of feasible directions: a study in linear and non-linear programming*. Elsevier.