Generalized Algorithmic Frameworks for Optimizing Distance Calls in Generalized Metric Space Proximity Problems and Methods for Realizing Efficient Signal Reconstruction

by

JEES AUGUSTINE

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
December 2020

To God, my Mother, my Father, Edward and Emmanuel, my Sister-in-Law and my Brother for their eternal sacrifices to provide me wings

ACKNOWLEDGEMENTS

I would like to acknowledge and express my deepest appreciation to my supervising professor and committee chair, Dr. Gautam Das. This dissertation would not have been possible without his constant support, invaluable advice, insightful discussions and endless patience. I learned significantly from him about succinct problem formulation, clear organization of thoughts, solution exploration starting with elementary cases, exploring edges, theoretical rigour and finally assertive yet very easy to read presentation style even for very complex scientific ideas. I was quite fascinated by his innate ability to skillfully see through tangled problems to come up with clear and easy to analyse analogies. He is an amazing storyteller, and there was always something more to learn from his scientific narration style. His grandiose vision for projects, efforts to develop independent researchers and ceaseless passion for teaching never ceased to amaze me. As an advisor, he trusted and supported his students on their quest for knowledge and always encouraged open discussions. These discussions with him in the last few years were intellectually stimulating and scientifically fulfilling.

I extend my sincere appreciation to Dr. Senjuti Basu Roy, my primary collaborator. Her keen sense of responsibility, willingness to help, and dedication, were pivotal to the success of my projects. Her hard work and enthusiasm for research motivated me to work harder. I would like to extend my warmest thanks to Dr. Divesh Srivastava, for his very careful perusal of the manuscript for errors, both grammatical and factual, insightful comments, attention to scientific detail, and his perennial

overjoyed appearance. I want to extend my appreciation to my kind collaborators, especially to Dr. Azade Nazi, Dr. Nan Zhang, and Dr. Nick Koudas.

I am highly indebted to highly esteemed committee members, Dr. Ramez Elmasri, Dr. Manfred Huber and Dr. Gergely Záruba for their willingness to be in my Ph.D. committee. Their indispensable suggestions and feedbacks have helped me to enrich the quality of this dissertation.

DBXLab was a home to me during my doctoral days and its members, a family. Especially, I would like to thank Suraj Shetiya, a good friend, a brilliant researcher and a wonderful collaborator. I really enjoyed and will always cherish the hours-long discussions on research problems. I am also thankful to Dr. Sona Hasani, for always motivating me and inquiring about my progress and to Shohedul Hasan, for his valuable advice on programming practices. I am hugely obliged to Dr. Abolfazl Asudeh, my research mentor, my senior and my collaborator and a crazy ambitious researcher. He believed in me, which gave me the confidence to face research challenges and was always a call away for any help, even after his graduation. I would also like to thank Dr. Saravanan Thirumuruganathan for his vastness of knowledge and his willingness to share. Similar to searching in google, I always start my literature survey for any research topic in computer science with him and, the insights he provided were stupendous.

I would like to thank my undergraduate advisers Dr. Jisha P Abraham and Dr. Surekha Mariam Varghese for their great guidance and motivation. I would also like to thank Jishi Abraham for her great support during the undergraduate projects and introducing me to the world of research.

I am thankful to my friends, who enriched my life with their gracious presence. Specifically, I like to thank Balan Ramesh, who was my roommate for the past five years. His welcoming lively demeanour, enthusiasm, unparalleled athleticism and

attention to detail helped to elevate my living standards. Dr. Abhishek Santra, my friend, made every pedestrian occasion a celebration, during my Ph.D. and always filled us with his culinary delights. I am thankful to Dr. Sajib Datta, a mentor to me and posed the idea of free and delightful living. I am thankful to Ashwin Raju for teaching me to push the boundaries to reach your full potential, both mentally and physically. I would like to thank my friends Avinash Ravi Menon for the lively vacation trips and philosophical discussions, Mullai Varadharasan for his handy bits of advice, Prajal Mishra for his characteristic pursuit of happiness, Sruthi Srinivasan for her integrity and genuineness and, Davis Thomas for his true friendship. My community life was supplemented though my friendship with Aby Cyriac a constant entrepreneurial, Sandeep Eldho James, the avid science and music enthusiast and Jiss Jacob Sebastian, the best multi-tasker and brilliant time manager that I have known.

I would like to thank Apple Computer Inc for designing such a magnificent product, the MacBook Pro. This machine offered a constant companionship and sturdy service during my doctoral studies. I would also like to thank Arlington Public Library for the invaluable collection of books, which helped me to grow over the course and, their deep dedication to improving the lives of residents through the wizardry of education without prejudice.

Finally, I am eternally grateful to God Almighty and to my family. I express my sincere gratitude to my parents, Celine Cybil and Augusty Tharayil Thomas, for their sacrifices to support me, and my dreams, with unconditional love, never-ending emotional support and consistent motivation. They taught me the values of education, humble beginnings, hard work and persistence at the face of struggle. I would like to thank my nephews, Edward and Emmanuel, for the immense joy, fulfilment and optimism they provided me, every single time I see or hear them. I am fully indebted to my Sister-in-Law, Anitta Jose, whom I consider and value as my

elder sister. During my doctoral days, she was a counsellor, a spiritual guide, a patient listener, a great supporter and a moral beacon. This acknowledgement will be short of its purpose without the mention of my elder brother, Dr. Charles Augustine, for instilling in me the values of hard work, consistency, dedication, curiosity, scientific inquiry, constant improvement, and competition. Beyond being a brother, he is a true friend, a great motivator, a good teacher, and a humble human being, from whom I learned a lot about life. I always looked up to him for motivation and inspiration. I em extremely lucky, proud and at the same time humbled to be a part of such a wonderful family.

December 08, 2020

ABSTRACT

Generalized Algorithmic Frameworks for Optimizing Distance Calls in Generalized Metric Space Proximity Problems and Methods for Realizing Efficient Signal Reconstruction

Jees Augustine, Ph.D.

The University of Texas at Arlington, 2020

Supervising Professor: Dr. Gautam Das

The exponential rise in data, along with its heterogeneity and complexity, helped individuals, businesses, hospitals, enterprises and even governments to thrive on data-driven decision making. However, as the size and complexity of the data surged, challenges in searching for similar objects within databases (*proximity search*) compounded.

As is well known, proximity search is the key and successful method used in Information Retrieval (IR) in vast databases including, Genomics Databases, Image Databases, Video Databases, Text databases, etc. The objective is to retrieve contents from the database, *similar* to a given object in the database.

This dissertation revisits a suite of popular and fundamental proximity problems (such as, KNN, clustering, minimum spanning tree) that repeatedly perform distance computations (sometimes by making calls to a third party distance oracle, such as Google Maps or Bing Maps) to compare the distance between a pair of objects during their execution. The chief 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.

The fundamental understanding of the structure of the proximity problem solutions reveals the underlying dependency on repetitive conditional statements predicated on the distance between pairs of objects. Thus, Direct Feasibility Test is designed to study how distance comparisons between two different pairs of objects could be modelled as a system of linear inequalities that assists in saving distance computations. Direct Feasibility Test offers the maximal savings for proximity problem solutions and to the best of the knowledge is the first attempt to provide a theoretical understanding of the problem.

Furthermore, the study also offers an alternative formalism with the goal of computing distance bounds. This work also develops a suite of graph-based algorithms that trade-off between running time and tightness of the produced bounds, whilst producing identical and exact solutions to these problems. A comparison of these designed solutions conceptually and empirically concerning a broad range of existing works is also presented in this work.

The work also presents 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 the proposed approach. To the best of the knowledge, the work is one of the first that takes a systematic stride to minimize repeated distance computation costs inside proximity problems.

Finally, a deep understanding of the challenges in large scale signal reconstruction is also addressed in this dissertation as a practical solution to large scale signal reconstruction problem.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

Introduction

1.1 The Proximity Problem

The Proximity Problems are a class of Computational Geometry problems, which involves the notion of distance between objects. Some of the classical problems that fall under this category include k-Nearest Neighbors, Minimum Spanning Tree, Clustering, Closest pair of points problem, diameter of a point set, etc. In all of these aforementioned problems, the distance between objects abstracts the (dis)similarity between a pair of objects.

For a given universe of n objects, computing all pair distances is quadratic in the number of objects, $\binom{n}{2}$ and, thus, is often computationally challenging. Besides, when soliciting the distance information for a specific object pair of objects, additional challenges arise from (i) involvement of computationally expensive algorithms for distance computation or (ii) involvement of expensive third-party APIs which provide distances on demand. Examples of the former include execution of complicated computer vision algorithms, which often requires human intervention in determining the actual distances[1, 2, 3, 4]. Alternatively, an example of the later include computation of proximity between a pair of Point-of-Interests (POIs), for which we often resort to Google or Bing Maps[1]. For brevity, in this dissertation, such expensive distance estimators are designated as Oracles.

[1]Both of these services offer the best estimate of distances and travel time based on numerous attributes, including GPS coordinate of POIs, traffic conditions, speed restrictions, historic traffic data, etc.

Consequently, the computation of all pair distances is neither efficient nor practical. Thus, as a good fit, reducing the number(overall) of distance calls becomes the natural choice for solving the proximity problems. In this model of solving proximity problems, instead of resolving a specific pair of objects, the model relies on the information about the range of values the distance can assume to make decisions. These range of values an actual distance can assume, referred to as bounds, helps to reduce the number of distance calls to Oracles.

Thus the theme of this dissertation is to focus on algorithmic schemes which provide the tightest bounds, with, however, minimal oracle calls. In the proposed model, for the tight estimation of bounds, we only assume the distances to obey metric property, in particular, the Triangle Inequality property, and thus a generalized solution framework. This assumption is quite realistic and is justified in a wide variety of real-world applications.

The immediate problem follows from re-authoring the proximity problem solution to take advantage of the bounds without losing the integrity of the solution. The re-authoring thus should be both general, and minimally invasive to the proximity solution. Therefore, we scrutinize the proximity algorithm to identify salient steps in the algorithm, which are both repetitive and involve multiple Oracle calls.

The techniques developed works for general metric space, and thus may apply to any proximity problems and is not restricted to a handful of applications discussed under the purview of this dissertation. Therefore, we expect similar results for any solution to proximity problems on metric space. Through our solution framework, even with lesser overall distance computation, we do not alter the outputs of the proximity algorithm, thus supporting the algorithmic integrity of the proximity solutions.

1.2 A Graph Based Approach as a General Plug-in for Optimizing Distance Calls for Metric Space Proximity Problems

In this research, our goal is to minimize the number of calls to an expensive distance oracle. This work views a set of objects as nodes and corresponding distances between objects as edges in a graph. We offer an upper and lower bound plug-in service for the proximity problems, which is sensitive to the sparsity of the graph. The proximity problem solutions, despite the introduction of the plug-in, reproduce the original answer without the plug-in.

This work, we make fundamental theoretical and practical advancements in two critical aspects. First, we provide a scheme, `SSLB`, which provides a tight lower bounds. Specifically, we design `SSLB-Tree` to provide lower bounds on any given tree. Departing from the traditional view of the problem, which never accounts for the sparsity of the graph, through `SSLB-Tree` , we account for it while computing the bounds. The proof of correctness of `SSLB-Tree` and an in-depth theoretical analysis are provided as a part of the manuscript.

However, `SSLB-Tree` only works when the given set of known edges forms a tree. Often, this is not a realistic assumption. To account for the graphs, which are common in real-life scenarios, we propose two distinct tree building approaches, with a goal is to extend `SSLB-Tree` to general-purpose graphs. The two approaches, `Bottom-Up Tree` and `Top-Down Tree` provides a loose lower bounds on the unknown edge distances, yet provide guarantees on running time, leaving the overall complexity of the scheme `SSLB` quadratic.

For a practitioner, the work also illustrates the necessary code changes to induce the plug-in service as part of any proximity problem solution. A comprehensive set of experiments are carried out with both real and synthetic datasets to demonstrate

the efficacy of the methods proposed in the light of three different class of proximity algorithms.

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

Proximity search algorithms repeatedly invoke the expensive Oracles for their progress forms the impetus for this work. On scrutiny of the proximity problem solutions, we recognize conditional statements within the solution framework, which are characteristic in these algorithms. All of such proximity problem solutions (specifically that we considered in this dissertation and, in general for proximity problems) makes use of a distance comparison between two different pairs of objects for deciding its further course. For example, in Prim's [5] algorithm for Minimum Spanning Tree(MST) on a weighted undirected graph, the decision to add next edge to the MST is predicated on choosing the shortest edge that is reachable from the partial tree constructed. This choice thus requires distance computation for unknown edges(for each of the outgoing edges from the latest node added to the partial tree) through Oracle calls. Once computed, these edge lengths are compared with others already resolved and not yet added to the tree. The lowest among these edges is selected and added to the MST.

Thus distance computation and comparison form the basic building block of these algorithms. Since we focus on minimizing the distance computations without impacting the results of comparisons, we identified and replaced the distance computation with the bounds obviating the need for actual Oracle calls.

The fundamental and the most challenging question is, can we make use of the local computation to reduce the overall expensive distance Oracles calls. In response to this question, and for fundamental understanding, we formulate the problem of

finding the exact bounds(lower and upper) for each unknown distance as a linear inequality, all of which together forms a system of linear inequalities.

These linear inequalities, formed in the light of Triangle Inequalities, can be solved by any linear system solvers to obtain the theoretically possible lower and upper bounds possible for each unknown distance. While this formulation yields the tightest bounds on the unknown distance, because of the computational expense of linear program solvers, the solution is not scalable and therefore limiting its application to only a setting where there are only few hundred objects or less.

We thus focus our attention to a scalable and yet efficient solution to the problem of minimizing the number of Oracle calls in proximity problems. Here we seek to pursue an alternate Graph-Theoretic approach for the problem of reducing the number of distance calls. To model the graph, we consider the objects as nodes and their corresponding distances, the edges.

Once modelled as a graph, we briefly introduce the notion of the upper and lower bounds for unknown edges. The work by Wang and Shasha [6], notes that the upper bounds are the length of the shortest path between the ends points of the unknown edges. Computation of lower bounds, as established in this work, is not straightforward and requires enumeration of all paths between a pair of nodes whose edge value is unknown. This work employs a cubic, dynamic programming algorithm, `ADM` , to compute the bounds for all the unknown edge in the graph.

Through formulation as a graph, we provide a geometric interpretation of the lower bounds. Since the estimation of tightest bounds yields the highest distance saves, we propose a Dijkstra's [7] shortest path-based algorithm, `SPLUB` , for computation of both upper and lower bounds. We also prove that the bound computed by our algorithm produces exact tightest lower bounds for any given unknown edge. In addition to computing the exact bounds, the proposed solution is significantly faster.

The overall running time of our algorithm is in the order of the number of missing edges. In a sparse graph, this is in the order of second power of the number of objects.

This guided approach improved the scope of the proximity solutions to graphs with few million edges. However, for large scale algorithms, often with a few hundreds of million edges, finding shortest paths inefficient and computationally expensive.

As the last part of the algorithmic contribution, we focus on large scale algorithms, which can work with approximate bounds. It is worthwhile to note here that, we relax the tightness of bounds, and the proximity algorithm, yet produces the identical solution as the one without the bounds. We propose a novel bounding algorithmic scheme, `Tri Scheme` , which is inspired by the lower bound estimation algorithm, by shortening the length of paths explored. `Tri Scheme` yields looser bounds, however, in practice, as evident from the experiments, shows only a marginal increase in the number of distance calls. We also provide rigorous theoretical analysis for the Expected running time for `Tri Scheme` to lookup an edge.

Extensive experiments, in light of three different class of proximity algorithms, evaluating five different algorithms with a variety of real-world datasets demonstrate the effectiveness of proposed algorithmic schemes, `SPLUB` and `Tri Scheme` .

1.4 Orca-SR: A Real-Time Traffic Engineering Framework leveraging Similarity Joins

This work investigates the far-reaching effects of database techniques developed for specific database query optimization, extending beyond the real of database applications. Here, in this work, we consider the challenging Signal Reconstruction Problem (SRP) with the varied application. The primary objective of SRP is to reconstruct a high dimensional signal from its low dimensional observations. Earlier works by Abol et.al. [8, 9, 10], lays the foundations of using database techniques such

as similarity joins and selectivity estimation for set similarity queries in solving the SRP. In these works, first, the SRP problem is posed as a solution to a system of linear equations and is later reformulated to a Lagrangian Formulation of SRP.

Even though the solutions proposed in these work are scalable, to make it practical and realistic, there are multitudes of challenges to overcome. To simulate the SRP over billion size datasets, the complete simulation of the network is necessary.

This work discusses the practical challenges in simulating the network and along with it a routing scheme akin to hierarchical routing, which improves the intractability of the solution significantly. We introduce a real-time system for Signal Reconstruction, with multi-user interaction, scalability and billion size network generation[11].

1.5 Dissertation Organization

In Chapter 2, we consider a novel problem of minimizing the distance Oracle calls through upper and lower bound computation.

Here, our principal contribution is an sparsity sensitive algorithmic scheme, offered as a plug-in, which minimizes the over distance computations for proximity problem solution algorithms. We also add a practitioners guide on to fit our plug-in inside various proximity problem with minimal code changes. We provide comprehensive theoretical analysis for our schemes, as well provide extensive experimental results on many real and synthetic datasets.

In Chapter 3, we address the scalability challenges involved in graph-based methods developed in chapter 2[12]. For the first time in literature, we provide a theoretical formulation of the problem in the form of linear inequalities and offer a solution for the tightest bounds. Later, we propose a graph-theoretic approach for bound computation. We introduce the shortest path based exact solution scheme for bounds and provide the theoretical analysis. For large settings, we propose a Triangle-

based approach which works well in practice and data structure optimizations for bound estimation and updates. We also present results from comprehensive set of experiments with real-life datasets.

Finally, in Chapter 4, we introduce the challenges in large scale signal reconstruction. Our technical contribution includes the design a real-time signal reconstruction system ORCA-SR which in multi-user interactive. ORCA-SR offers a window to various database optimization applied in producing the reconstruction. Our system can scale to large network setting of billion sizes. We simulate these billion sized networks and present the interactive system as a web-application.

In addition to these works, the author was also a co-author of papers, "Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries" [13], and "Multi-attribute selectivity estimation using deep learning" [14], which is not a part of this thesis.

CHAPTER 2

A Graph Based Approach as a General Plug-in for Optimizing Distance Calls for Metric Space Proximity Problems

2.1 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. For example, computing the distance between a pair of objects may involve running expensive computer vision algorithms, involving actual human experts (inferring dissimilarity between two CAT-SCAN images may be hard to capture using a distance function and may require human involvement), or may even require calling third party applications (e.g., map APIs) which may impose monetary costs per number of queries or put constraints on query rate, e.g., limit the number of queries a user can make in a given time period.

In an abstract sense, we assume access to a *distance oracle* which is an expensive function that takes as input any pair of objects and outputs the distance between them. Our objective is to consider existing algorithms for various classical proximity problems, and redesign these algorithms such that they are refocused towards minimizing number of distance calls rather than overall computation cost.

At the heart of our techniques is the observation that when distances satisfy the triangle inequality, upper and lower bounds can be calculated for some of the

distances that may not yet be precisely known. For example, given a set of three objects satisfying triangle inequality, if the distances between two pairs of objects are 4 and 10, then the third distance pair has to be between 6 and 14.

Prior work [6, 15] has shown that given n objects in a general metric space, if only a subset of the $\binom{n}{2}$ distances are known, then upper and lower bounds can be computed for each of the remaining unknown distances. Our *first contribution is to provide a general recipe for how the bounding algorithm [6, 15] can be incorporated inside any classical proximity algorithm as a convenient "plug-in", which refocuses the proximity algorithm towards reducing distance oracle calls.* Essentially, during the process of computation, this plug-in estimates lower and upper bound of distances between pairs of objects, and these bounds are often adequate to make subsequent decisions. For example, in an algorithm for computing the nearest neighbor of an object u, consider two candidates v and u such that $d(u, v)$ and $d(u, w)$ have not been precisely computed yet. However, if the current best lower bound for $d(u, v)$ turns out to be larger than the current best upper bound for $d(u, w)$, then v can be eliminated from further consideration.

Our *second contribution is algorithmic.* The best known algorithm for upper and lower bounds estimations of unknown distances are proposed by Wang and Shasha [6, 15]. Although designed more than two decades ago, to the best of our knowledge, these algorithms are the state-of-the-art solutions to compute tightest lower and upper bounds in general metric space. The main limitation of their proposed framework `ADM`, which computes lower and upper distance bounds of all unknown edges at the same time, is its high computational complexity (cubic in number of nodes). In fact, unlike classical shortest path algorithms (e.g., Dijkstra's [7] Algorithm) [1] whose asymptotic running times depend both on the number of nodes and on the number

[1] Bound estimation problems are directly related to the Shortest Path problems.

of known edges, the running time of these algorithms only depends on the number of nodes. Therefore, when the underlying graph is sparse (which is the case for most metric space proximity problems), that is, it has small number of known edges, this prior work [6] incurs high computational cost for bounds estimation. *As one of our algorithmic contributions, we present a new lower bound estimation algorithm, referred to as Sparsity Sensitive Lower Bound Estimation (*`SSLB` *in short) that is an improvement over this state-of-the-art prior work [6] in asymptotic running time (see Section 2.3.2). In the case the underlying graph of known edges form a tree, we prove that* `SSLB` *is guaranteed to produce the tightest lower bound.* Following this, we show how to extend `SSLB` to graphs of any arbitrary topology. We also present upper bound estimation algorithm that produce tightest bound and beats the cubic complexity of `ADM`. Nevertheless, both the prior work and `SSLB` search the entire graph to estimate the distance bounds. To avoid a search of the entire graph, in Section 2.3.3 we present parameterized "lightweight" bound estimation algorithms, that are highly scalable, but constrained to search inside a local neighborhood of the graph (parameterized as part of inputs). These latter algorithms compromise the tightness of estimated bounds to enable higher computational efficiency. To summarize, we present a suite of algorithms that make a trade-off between "tightness" of estimated bounds and computational efficiency.

Our third contribution is to provide insights for practitioners to understand how existing algorithms for proximity problems can make use of this plug-in in an on-demand fashion. We consider 3 different proximity problems and 6 different algorithms using 6 different large scale datasets. Our experimental results indicate that our plug-in is equally capable of saving distance calls for all proximity algorithms, producing lower and upper bounds that are highly comparable to that of the state-of-the-art [6, 15], and is significantly faster in running time. We show when proximity

algorithms use our proposed distance estimation algorithms as plug-in, it shows on an average 5× and as much as 20× improvement in saving up distance calls, while increasing only a few seconds of overall CPU time.

2.2 Datamodel and Formalism

We first present our data model and formalize the problems and present running examples.

2.2.1 Data Model

Objects and Object Graph. We are given with a set $\mathcal{O}$ consisting of n objects. The dissimilarity (distance) between any pair of objects induces an underlying complete undirected weighted graph $\mathcal{G}(\mathcal{O}, E)$ whose nodes $\mathcal{O}$ are the objects and the edges E are all pairwise distances between the nodes. This graph satisfies the metric property, in particular, triangular inequality.

Triangle Inequality Property For every three objects (o_i, o_j, o_k) that comprise a triangle $\triangle_{o_i,o_j,o_k}$, $d(o_i, o_j) \leq d(o_i, o_k) + d(o_k, o_j)$ and $d(o_i, o_j) \geq |d(o_i, o_k) - d(o_k, o_j)|$.

Distance Oracle: The complete edge set E is not given ahead of time. Instead, we assume there exists a distance oracle which, given two objects o_i and o_j, returns the true distance $d(o_i, o_j)$. The actual implementation of the oracle is application dependent and orthogonal to our framework. We only assume that calls to the oracle are expensive.

Since we focus on applications where calls to the oracle are expensive, our goal is to minimize the number of calls to the oracle. Intuitively, in our framework, instead of querying the distance oracle to precisely resolve a particular edge, the proximity algorithm first tries to compute *upper and lower bounds* of the edge, by leveraging

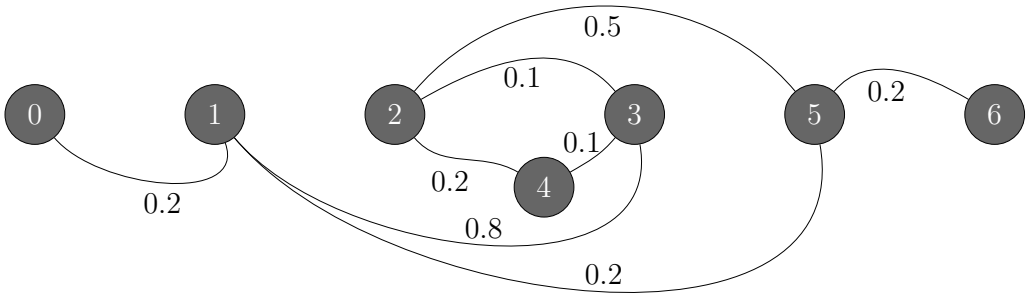


Figure 2.1: 7 objects and their corresponding known and unknown distances.

the fact that all distances have to satisfy metric space properties, in particular the triangle inequality. If the bounds information is useful to the algorithm, it avoids making the oracle call. If the bounds information is not useful, then it makes the oracle call. This way, the overall number of calls are reduced.

Example 1 *Consider a set of* 7 *objects with distances between* 0 *and* 1*. Assume these distances satisfy the metric property, i.e., triangle inequality of distances. The distances between the object pairs fall into one of two kinds: (a) it is known/resolved (b) it is unknown.*

Using Example 1, only 8 out of 21 edges are known (the solid lines), while the remaining 13 edges are unknown (the dashed lines). If the distance $d(1,3) = 0.8$ and $d(3,4) = 0.1$ then the the bounds for distance $d(1,4)$ may be computed as follows:

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

i.e.,

$$0.7 \leq d(o_1, o_4) \leq 0.9$$

2.2.2 Formalism

Problem 1 *Compute upper bound of the distance between (o_i, o_j) (or $UB^{d(o_i,o_j)}$), which is the largest possible distance that an unknown edge can get 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 [6] that will go through additional intermediate objects.

$$UB^{d(o_i,o_j)} \leq sp(o_i, o_j) \tag{2.1}$$

Using Figure 3.1, when the distance between (o_3, o_6) is unknown, the upper bound for could be estimated by considering a path $o_3 \rightarrow o_4 \rightarrow o_2 \rightarrow o_5 \rightarrow o_6$, or $o_3 \rightarrow o_2 \rightarrow o_5 \rightarrow o_6$, or $o_3 \rightarrow o_1 \rightarrow o_6$, and taking the one that is smallest (i.e., $o_3 \rightarrow o_2 \rightarrow o_5 \rightarrow o_6$). Thus $UB^{d(o_3,o_6)} = 0.8$ is the length of the shortest path.

Problem 2 *Compute lower bound of the distance between (o_i, o_j) (or $LB^{d(o_i,o_j)}$), which is the smallest possible distance that an unknown edge can get without violating the triangle inequality, considering all known and unknown distances in $\mathcal{G}$.*

Unlike the upper bound, computing the lower bound is more complicated. 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. For each path, the LB could be computed using the generalized metric property proposed in [15] - 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 $d(o_k, o_l)$ between o_k, o_l) in p.

$$LB^{d(o_i,o_j)} \geq Maximum_{\forall p}\{d(o_k, o_l) - path(o_i, o_k) - path(o_l, o_j)\} \tag{2.2}$$

Using Example 1, the lower bound of the distance $LB^{d(o_3,o_6)}$ needs to consider: (1) $o_3 \rightarrow o_4 \rightarrow o_2 \rightarrow o_5 \rightarrow o_6$, (2) $o_3 \rightarrow o_2 \rightarrow o_5 \rightarrow o_6$, and (3) $o_3 \rightarrow o_1 \rightarrow o_6$. For the

first path, the lower bound can be calculated to be $0.5 - (0.1 + 0.2 + 0.2) = 0$. The second path results in a lower bound of $0.5 - (0.1 + 0.2) = 0.2$ and lastly, from the third path we get $0.8 - (0.2 + 0.2) = 0.4$. The $LB^{d(o_3,o_6)} = max\{0, 0.2, 0.4\} = 0.4$.

2.2.3 Different Proximity Problems

Proximity Problem 1: k-Nearest Neighbor Graphs: Given a set of objects $\mathcal{O}$ and an integer value k, with distances between each pairs of objects in $\mathcal{O}$, compute the top-k nearest neighbor graph (k-NNG), which is a graph in which each object has k directed edges to its k nearest neighbors.

Proximity Problem 2 : l-Medoid Clustering: The *medoid* of a collection of n points $C_i = \{o_1, o_2, \ldots, o_n\}$, is defined as a point $o_m (1 \leq m \leq n)$, which is a representative center of the elements of C_i that minimizes its distance with all other points of C_i.

$$o_m = argmin_{o \in C_i} \sum_{i=1}^{n} d(o, o_i)$$

The objective of l-Mediod Clustering is to find a partition for a given set of objects around l medoids which minimizes the sum of distances between each point and its corresponding medoid. More formally, given a set of objects $\mathcal{O}$ with similarities/distances between each pairs of objects, and an integer l, partition $\mathcal{O}$ into l clusters $C_1, \ldots, C_l$ with each cluster C_i having object o_i as its medoid, such that the following function is minimized:

$$Cost(C^1, ..., C^l) = \sum_{j=1}^{l} \sum_{i \in C^j} d(o_i, C_j)$$

where o_i is the medoid of cluster C_i.

Problem 3 : Minimum Spanning Tree: Given a set of objects $\mathcal{O}$ with similarities/distances between each pairs of objects, compute the minimum weight

spanning tree $MST(G)$, i.e., a connected weighted subgraph of $n-1$ edges that connects all n nodes together, such that the total weight of the edges in the tree is minimized.

2.3 Distance Estimation Algorithms

This section is organized as follows. We begin in Section 2.3.1 by describing a prior work ADM [6], which we refer to as the *Baseline Algorithm*. Although ADM is more than two decades old, to the best of our knowledge, it is the state-of-the-art solution to compute the tightest lower and upper bounds in general metric space. The main limitation of ADM though is its high computational complexity, especially for lower bound estimation (cubic in number of nodes). In fact, unlike classical shortest path algorithms (e.g., Dijkstra's [7] Algorithm) [2] whose asymptotic running times depend both on the number of nodes and on the number of known edges, the running time of ADM only depends on the number of nodes. Therefore, when the underlying graph of known edges is sparse (which is usually the case in proximity problems), that is, it has small number of known edges, this prior work [6] incurs high computational cost for upper and lower bound estimation.

We next present in Section 2.3.2 one of our main algorithmic contributions, a new lower bound estimation algorithm that we refer to as the *Sparsity Sensitive Lower Bound Estimation algorithm* (`SSLB`), which is an improvement over ADM. We prove that `SSLB` is guaranteed to produce the tightest lower bound when the underlying graph of known edges form a tree. Following this, we describe how to extend `SSLB` to graphs of any arbitrary topology.

[2] Bound estimation problems are directly related to the Shortest Path problems.

One disadvantage of both ADM and `SSLB` is that they need to traverse the entire graph to estimate the distance bounds. In Section 2.3.3, we present a parameterized "lightweight" solution of these bound estimation problems, that is highly scalable by constraining the traversal inside a local neighborhood of the graph (parameterized as part of inputs).

To summarize, our goal in this section is to present a suite of algorithms that make a trade-off between tightness of estimated bounds and computational efficiency. Nevertheless, when used as a plug-in inside a proximity problems, they always produce the exact answer - i.e., the outputs of the proximity algorithm is the same irrespective of whether the plug-in is used or not.

2.3.1 ADM - Baseline Algorithm by Wang and Shasha

Wang and Shasha [6] propose a dynamic programming based framework `ADM` that can estimate upper and lower bound of all the unknown distances, based on the set of known distances, using the generalized triangle inequality property presented in Definitions 1 and 2.

`ADM` keeps track of two matrices, LB and UB for lower and upper bounds, respectively. Each entry of $LB_{i,j}$ ($UB_{i,j}$) contains either the known distance between o_i and o_j or the latest lower bound (upper bound) of distance between o_i and o_j, namely, $LB^{d(o_i,o_j)}$ ($UB^{d(o_i,o_j)}$).

It runs in n iterations and gradually builds the two aforementioned matrices. At the i-th iteration, this estimation takes place by considering i intermediate objects, i.e., $(i+1)$ length paths from o_i to o_j. While updating a single entry (i,j) in those matrix during the $i+1$-th iteration, it checks (in constant time) if it is worth to consider this new object to update the bounds, or the bounds obtained in i-th iteration is best so far. The algorithm also assigns unique indexes to the objects to ensure that

they are not considered redundantly during the run. The pseudo-code is described in Algorithm 1.

Limitations: While `ADM` indeed produces the tightest bounds, the main challenge is that its computational complexity is cubic in the number of nodes, and ais not sensitive to the number of edges. If the underlying graph is a large sparse graph (that is, with a relatively small number of known edges), invoking this algorithm many times as a plug-in is computationally expensive.

Algorithm 1 Algorithm `ADM` : Algorithm for Computing Tightest Bounds

inputs: graph G, a set of pre-computed distances

output: LB and UB

Initialize LB and UB, set them to true distances; otherwise $LB_{i,j} = 0$ and $UB_{i,j} = \infty$

$n = |\mathcal{O}|$

for $k = 1$ to n **do**

  for $i = 1$ to n **do**

    for $j = 1$ to n **do**

      $LB_{i,j} \leftarrow max\{LB_{i,j}, LB_{i,k} - UB_{k,j}, LB_{j,k} - UB_{k,i}\}$

      $UB_{i,j} \leftarrow min\{UB_{i,j}, UB_{i,k} + UB_{k,j}\}$

    end for

  end for

end for

2.3.2 Sparsity Sensitive Bound Estimation

In this section, we present significantly more efficient bound estimation algorithms, both for upper and lower bounds. Our new lower bound estimation algorithm, referred to as *Sparsity Sensitive Lower Bound Estimation* (`SSLB` in short) is an improvement over the state-of-the-art `ADM` in computational time. The upper bound estimation, on the other hand, is simpler and involves multiple shortest path computation. We conclude this section (Section 2.3.2.3) by discussing this latter algorithm. First, we discuss `SSLB` considering the underlying graph with known edges forms a tree (as `SSLB` guarantees the *tightest lower bound on trees*), and then describe how to extend it to graphs of arbitrary topology.

2.3.2.1 SSLB On Tree

Here the given graph with known edges forms a tree $\mathcal{T}$ with n objects and $(n-1)$ edges, similar to one shown in Figure 2.2. The algorithm `SSLB-Tree` comprises of the following steps: it first chooses a object arbitrarily from $\mathcal{T}$ and assigns it as the root. It then starts exploration of the tree from the selected root object, o_{root}. The `UpdDist` sub-routine then updates the distances from the root to every object in the tree. `SSLB`uses the paths that go through the root, to compute the *Lower Bound Matrix* referred to as LB. Finally, `SSLB`is recursively invoked on each of the sub-trees of the root. Algorithm 2 contains the pseudocode.

To illustrate the algorithm further, we refer to Figure 2.2 and use the sample tree as a running example through out the section. As seen from the example tree, root node o_1 has three children (o_0, o_5 and o_3). Each of them, in turn, are roots of their sub-trees below them. We denote these sub-trees as (U_1, U_2 and U_3) (generalized to o_l sub-trees with respect to the number of children of the parent object.). One of such sub-trees U_2 is shown within the box in the figure.

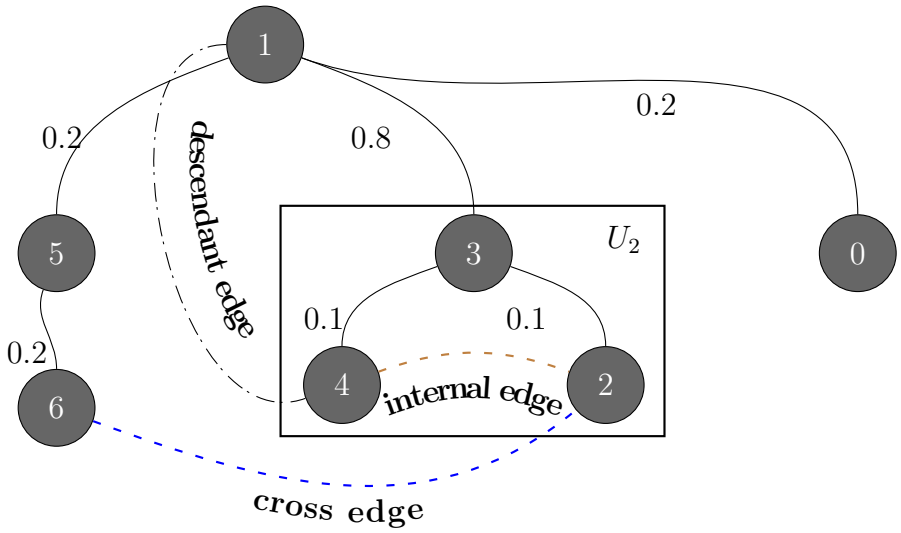


Figure 2.2: Sample Tree

Any unknown edge, can be categorized into three types as follows:

***Cross edges*:** Edges between pairs of objects, across the sub-trees. As in the example tree edges of type, ($o_2 \in U_2$, $o_6 \in U_1$).

***Descendant edges*:** Edges between root of the tree, and all its descendants, similar to the ones shown by the edge (o_2, o_6) in the example tree.

***Internal edges*:** Edges between all pairs within a sub-tree. An example is the edge (o_2, o_4) in U_2.

A naive algorithm to estimate tightest lower bound of all unknown edges will work as follows: there are $\mathcal{O}(n^2)$ pairs of unknown edges. To compute the lower bound for each of such edge, we need to find a path between all such pairs. Since finding the path between a pair of objects in a tree takes linear time, this path computation takes $\mathcal{O}(n^3)$ time. For each path, the lower bound can be computed by subtracting the weight of the rest of the path from the highest weight edge, as defined in equation 3.5. So the naive algorithm is no better than `ADM`.

`SSLB-Tree` described in Algorithm 2, however, does not follow this naive approach. `SSLB`stores auxiliary information at every object to compute the lower bound for any unknown edge in constant time. For every object n_i, our algorithm maintains two auxiliary *values*, namely (i) The sum of the edge weights along the path between the root and n_i denoted as $len\_path_{o_i}$ (ii) the maximum of the edge weight among the edges along the path between the root and n_i denoted by max_{o_i}. From the stored len_path and max, `SSLB` directly finds the path-length and the largest edge along the path between pairs of objects. The lower bound of any unknown edge could thus be computed as a difference between the maximum edge and the reminder of the path length. `SSLB` maintains the LB, and updates the lower bound $LB_{i,j}$ corresponding to any unknown edge (o_i, o_j) in the graph.

Calculating Lower Bound Distances: A close analysis of the tree reveals that the lower bounds on any *cross edge*, (o_x, o_y) can be computed by utilizing the stored values in the objects. Let the root of our `SSLB`be represented by o_{root}. Let max_{o_i} represents the edge weight with the maximum weight along the path between o_i and o_{root}, and $len\_path_{o_i}$, represents the total path weights from object o_i to o_{root}.

Thus from our running example, if we consider the object $o_4 \in U_2$, $max_{o_i} = 0.8$ and $len\_path_{o_i} = 0.8 + 0.1 = 0.9$, on a tree rooted at 1. For every *cross edge* ($o_x \in U_i, o_y \in U_j$) between U_i and U_j, ($i \neq j; 1 \leq i, j \leq l$), the lower bound is computed as follows,

$$LB_{x,y} \leftarrow 2 * max(max_{o_x}, max_{o_y}) - len\_path_{o_x} - len\_path_{o_y} \tag{2.3}$$

In our running example, the lower bound for one such *cross edge* (o_2, o_6), could be calculated as, $LB_{o_2,o_6} = 2 * max(0.8, 0.2) - 0.9 - 0.4 = 0.3$. Note that the sum of the edge weights between (o_x, o_y), $(len\_path_{o_x} + len\_path_{o_y})$ already accounts for

the maximum edge and henceforth we added a correcting factor of 2 to the maximum edge.

Since $len\_path_{o_{root}}$ and $max_{o_{root}}$ are both 0, the lower bound calculation for any *descendant edge*, (o_x, o_{root}), from Equation 2.3 simplifies as follows:

$$LB_{root,x} \leftarrow 2 * max_{o_x} - len\_path_{o_x}$$

Using our example, $LB_{o_1,o_4} = 2 * 0.8 - 0.9 = 0.7$.

The root, o_{root} partitions the tree into l sub-trees, $U_1, \ldots, U_l$. Recursively, `SSLB` computes the lower bound values for each *internal edge*, $(o_x, o_y) \in U_i$ updating the LB for all such edges. As from our running example, when `SSLB` recursively runs on sub-tree U_2 with root 3, $len\_path_{o_4}$ and max_4 are updated for object o_4 as 0.1 and 0.1 respectively. The lower bound of the internal edge could be computed as $LB_{o_2,o_4} = 2 * max(0.1, 0.1) - 0.1 - 0.1 = 0$, and thus did not improve the lower bound and is also the tightest lower bound from the tree.

Lemma 1 `SSLB-Tree` *always produces the tightest lower bound for any unknown edge of any given tree.*

Proof 1 *(Sketch) Tightest lower bound is the smallest feasible value that an unknown edge of a graph can have without violating the metric property. For any given path between a given pair of objects, equation 2.3 can be used to obtain a lower bound between the objects. Each path between a pair of objects thus provides a lower bound on the edge value. The tightest lower bound is thus the largest among all these lower bounds.*

In a tree there exists only one unique path between every pair of objects. *Tightest lower bound, as described earlier, is therefore the largest among such lower bounds, which* `SSLB` *captures. Hence the proof.*

Lemma 2 *Computing lower bounds of all unknown edges take* $\mathcal{O}(n^2)$ *time in* `SSLB-Tree` .

Proof 2 *The lower bound matrix, LB has exactly* n^2 *entries. Our* `SSLB`*algorithm, computes* $n(n-1)/2$ *lower bound entries. Given a* o_{root}*, the calculation of* max_{o_x} *and* $len\_path_{o_x}$ *for all the objects in the tree takes* $\mathcal{O}(n)$ *time. As our algorithm is recursive, and since there are n objects in the tree,* `SSLB`*completes in* $\mathcal{O}(n^2)$ *time.*

Algorithm 2 `SSLB-Tree` : Algorithm for Computing Lower Bounds Distances on Trees

Input: Tree(T)

Output: *lower bound matrix*(LB)

for child in T.children **do**

`SSLB-Tree` (child)

end for

`UpdDist`(T, 0, 0)

for $\{ \forall (c_1, c_2) \in T.children \mid c_1 \neq c_2 \}$ **do**

Compute lower bound for all cross edges between objects in c_1, c_2

end for

for child in T.children **do**

Compute lower bound for all edges between $root, child$

end for

2.3.2.2 SSLB On Graphs

In this section, our goal is to extend `SSLB` to general purpose graphs - meaning that the graph of known edges is not necessarily a tree. Our proposed Algorithm `SSLB-Graph` on a given graph works by generating two classes of trees, namely bottom-up and top-down trees. `SSLB-Tree` Algorithm acts on these generated trees to

Algorithm 3 `UpdDist` : Subroutine for Updating *max* and *path* Distances Between a Given Root & All its Children

Input: object(o), *max_path*, *distance*

Output: None

o.path = *distance*

o.max = *max_path*

for c in o.children **do**

`UpdDist`$(c, o.path + distance, max(o.max, max\_path))$

end for

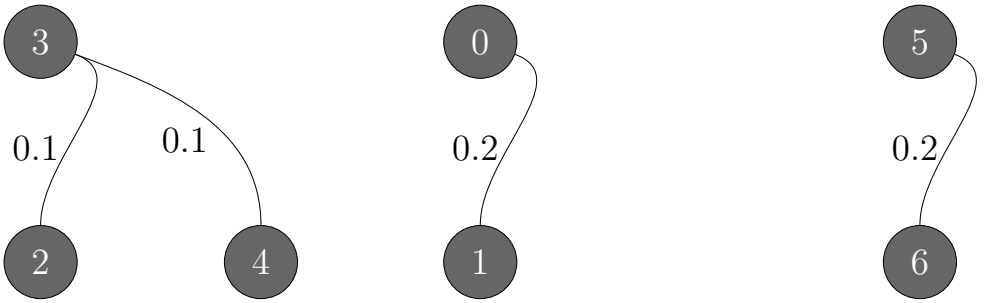


Figure 2.3: First Step of `Bottom-Up Tree`

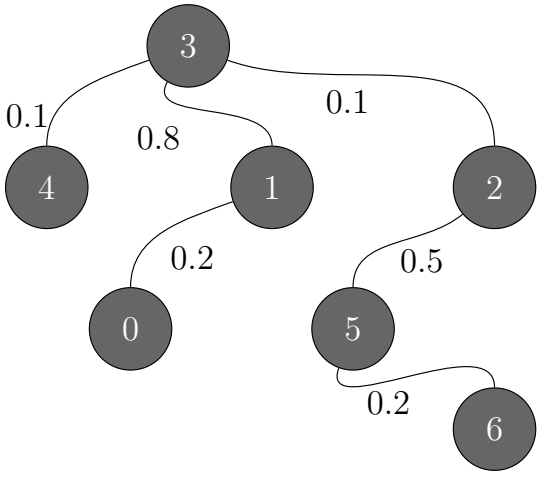


Figure 2.4: `Bottom-Up Tree`

update the lower bound matrix. As there are a large number of spanning trees and exploring all spanning trees to obtain the tightest lower bound is not feasible, `SSLB-Graph` remains to be a heuristic on general graphs. Algorithm 4 contains the pseudo-code.

Illustrating further, the intuition of how to build effective trees in `SSLB-Graph` comes directly from Equation 2.3. To get "tight" lower bounds, the idea is to increase the weight of the largest edge while keeping the rest of the path as small as possible. Both tree building algorithms use this intuition. `Bottom-Up Tree` follows a bottom-up approach for the construction of the tree. `Bottom-Up Tree` initializes the tree as a forest of objects, with no edges in it.It then merges pairs of trees, through addition of edges based on the edge weight connecting them. Through repeated addition of edges and merger it maintains a relatively balanced set of trees at each level. The second type of tree builder, `Top-Down Tree`, initializes the tree construction by choosing the largest edge to be a part of the tree. From either ends of the chosen edge, two shortest path trees are generated. A single tree is then formed my merging the two trees thus formed based on a proximity measure. By continuing the same principle of edge selection and merger, `Top-Down Tree` construction completes, growing the tree downwards. Finally the two disjoint trees are merged to form a single tree at the highest level. `SSLB-Graph` then invokes `SSLB-Tree` on `Bottom-Up Tree` and `Top-Down Tree` independently to obtain lower bounds.

Next we describe `Bottom-Up Tree` and `Top-Down Tree` in detail.

Bottom-Up Tree Construction: Next we describe the bottom up tree construction. The pseudo-code is presented in Algorithm 5. We first present Lemma 3 that forms the basis for designing `Bottom-Up Tree`.

Algorithm 4 `SSLB-Graph` : Algorithm for calculating lower bounds in a graph

Input: Graph($\mathcal{O}, \mathcal{E}$), depth(d)

Output: *Lower Bound Matrix* (LB)

BottomUpTree $\leftarrow$ `Bottom-Up Tree`($\mathcal{O}, \mathcal{E}$)

$LB \leftarrow \max(LB,$ `SSLB`(BottomUpTree))

for Tree in `Top-Down Tree`($\mathcal{O}, \mathcal{E}$, d) **do**

$LB \leftarrow \max(LB,$ `SSLB`(Tree))

end for

return LB

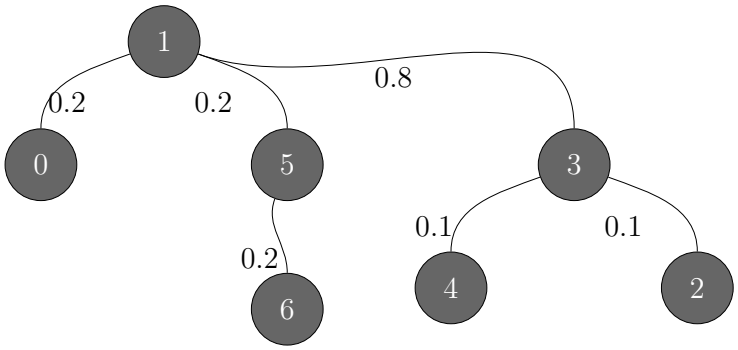


Figure 2.5: `Top-Down Tree`

Lemma 3 *Given a rooted tree, the total number of lower bound entries updated through cross and descendant edge exploration is bounded by the cardinality, C.*

$$C = \sum_{\substack{1 \le i,j \le l \\ i \ne j}} |U_i| \times |U_j| + \sum_{1 \le i \le l} |U_i|$$

where $|U_i|$ is the number of objects in the sub-tree U_i.

Proof 3 *Let us consider o_{root} as the root of the tree with sub-trees $U_1, \ldots U_l$. To account for the total number of* cross edges*, let us first consider the unknown edges between the two sub-trees U_i and U_j. The edge between any object $o_x \in U_i$ and $o_y \in U_i$ is unknown. Note that if there were to be such an edge, then, the path o_x to o_{root}, o_{root}*

to o_y and finally the edge (o_x, o_y) would form a loop, which is a contradiction. Thus, the contribution of cross edges *between U_i and U_j to the the number of lower bound updates is $|U_i| \times |U_j|$. The above property holds true for all pairs of sub-trees, thus bringing the total contribution from* cross edges *to $\sum_{\substack{1 \leq i,j \leq l \\ i \neq j}} |U_i| \times |U_j|$. The* descendant edges *from the sub-tree U_i are of the form (o_{root}, o_x), where $o_x \in U_i$, which account to a total $|U_i|$. Hence, the total number of unknown* descendant edges *is $\sum_{1 \leq i \leq l} |U_i|$. Summing the totals up we get $C = \sum_{\substack{1 \leq i,j \leq l \\ i \neq j}} |U_i| \times |U_j| + \sum_{1 \leq i \leq l} |U_i|$. Hence, proved.*

An observation that we make on the tree is that, irrespective of the distribution of objects in the tree, $\sum_{1 \leq i \leq l} |U_i| = n - 1$. Hence, we conclude that Cardinality C, could only be maximized, through maximizing the number of cross edges, which is achieved by a balanced tree. We then start with all the objects of the graph as a set of single object trees. Note that the initial set of trees are balanced. After that, we merge pairs of trees thus keeping the set of trees balanced. For merger of the trees, we choose the shortest edges creating stubs which are in-turn balanced. However, absence of some edges between pairs of objects leaves some of the trees unconnected rendering the tree unbalanced. To ensure relative balance of the tree, we assign the un-merged trees to one of the existing trees. Figure 2.3 presents the first step of the merge process, on a sample graph given in Figure 3.1.

Thus at each level of the tree, we combine two trees from the level below to form a single tree, propagating all the way to single merger which connects all trees to form a single tree, akin to a fully balanced binary tree. However, referring to Equation 2.3, one would prefer to have larger edges in the tree to maximize the number of lower bound updates through cross edge updates. Thus, during the final merger, we choose the largest edge to merge the two sub-trees.

In order to obtain edges with lower weights for the merger, we sort the edges by their weight in ascending order. Afterwards, we select the shortest edge from the

sorted list of edges. We merge the two trees that is connected by this edge, only if neither of the two trees were merged in this iteration. Continue merging pairs of trees, until the edge list is exhausted. As there might be trees that were not merged in this iteration, we go over the sorted edge list to merge these trees where ever possible. However, when there are a very few trees to merge meaning at the top levels, instead of merging based on small edges, we use the larger edges for the tree merger completing our `Bottom-Up Tree` algorithm.

Algorithm 5 (Sketch)`Bottom-Up Tree` : Algorithm for construction of `Bottom-Up Tree`

Input: Graph$(\mathcal{O}, \mathcal{E})$

Output: Tree

Edges $\leftarrow$ Sort($\mathcal{E}$, key:*edge weight*, descending=$True$)

Forest $\leftarrow$ $Set(Tree(i))\forall i < |\mathcal{O}|$

while (—Forest— ¿ 1) **do**

for (i, j) in Edges **do**

if i and j were un-merged, *merge* $tree_i$, $tree_j$

end for

If any $tree_i$ was un-merged merge with closest $tree_j$

if (—Forest— $\leq$ 3) **then**

Edges $\leftarrow$ Reverse(Edges)

end if

end while

return Merged Tree

Top-Down Tree Construction: As one observers from Equation 2.3, a good lower bound is obtained by maximizing the largest edge $max(max_{o_x}, max_{o_y})$ while minimizing the $len\_path_{o_x} + len\_path_{o_y}$. In order to maximize the first quantity, we must choose a large edge to be added to the tree. Following that, to achieve a minimization of $len\_path_{o_x}$ and $len\_path_{o_y}$, the distance from the root to each of the objects must be minimized.

`Top-Down Tree`works as follows (Algorithm 6 contains the pseudo-code). First, we consider all the known edges from the graph and add the largest edge, (o_p, o_q) to the tree. This maximized the first quantity of Equation 2.3. We choose either o_p or o_q as the root of our tree. In the reminder of the section we consider o_p for explaining the algorithm. Consider edge (o_1, o_3) to be the firs edge to be added in the `Top-Down Tree` from Figure 3.1 and pick o_1 as the root of the tree, as one can see in Figure 2.5, which is a fully constructed `Top-Down Tree` on the sample graph. To minimize the distance from `Top-Down Tree` root to all other objects in the tree, we run *Dijkstra's* shortest path algorithm from o_p as well as from o_q. The next step is to merge the two trees from *Dijkstra's* algorithm, based on the proximity to o_p or o_q. For every object o_x, among the two shortest paths, $SP(o_x, o_p)$ and $SP(o_x, o_q)$, we retain only the path whose length is smaller.

Figure 2.6 and 2.7 represent the shortest path trees from o_1 and o_3 respectively.

In Figure 2.5, let the shortest path sub-trees that are connected to o_p be $U_{p_1}, \ldots U_{p_l}$ and the shortest path sub-trees that are connected to o_q be $U_{q_1}, \ldots U_{q_l}$. Every lower bound calculation on the cross edges ($o_x \in U_{p_i}, o_y \in U_{q_j}$) involves a path through the edge (o_p, o_q). As edge (o_p, o_q) is a large edge, we therefore obtain a "good" lower bound for all the cross edges. The tree generated from `Top-Down Tree` is optimized for cross edges and hence internal edges of the generated tree may be ineffective in providing a good lower bound. To overcome this, we partition the

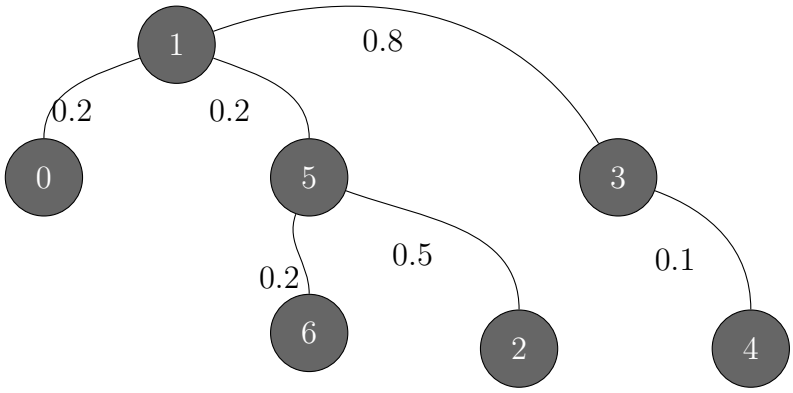


Figure 2.6: Shortest path tree root at object 1

objects into two groups, around the largest edge (o_p, o_q) and recursively performs `Top-Down Tree` on each of the partition separately. Moreover, as the effectiveness of the algorithm reduces as the levels of recursion increases, we limit the levels of recursion using a parameter d. The algorithm for the generation of `Top-Down Tree` is given in Algorithm 6.

Algorithm 6 `Top-Down Tree` : Algorithm for construction of `Top-Down Tree`

Input: Graph($\mathcal{O}, \mathcal{E}$), depth (d)

Output: `Top-Down Tree`

$e_{i,j} \leftarrow$ Max($\mathcal{E}$, key:*edge weight*)

$sp_i =$ `Dijkstra(`o_i`)`

$sp_j =$ `Dijkstra(`o_j`)`

Tree $\leftarrow$ merge sp_i, sp_j

$(\mathcal{G}_i, \mathcal{E}_i), (\mathcal{G}_j, \mathcal{E}_j) =$ partition Graph around the edge $e_{i,j}$

Parent Pointer$_{(i)} \leftarrow$ `Top-Down Tree` $((\mathcal{G}_i, \mathcal{E}_i), d-1)$

Parent Pointer$_{(j)} \leftarrow$ `Top-Down Tree` $((\mathcal{G}_j, \mathcal{E}_j), d-1)$

return *Parent Pointer*

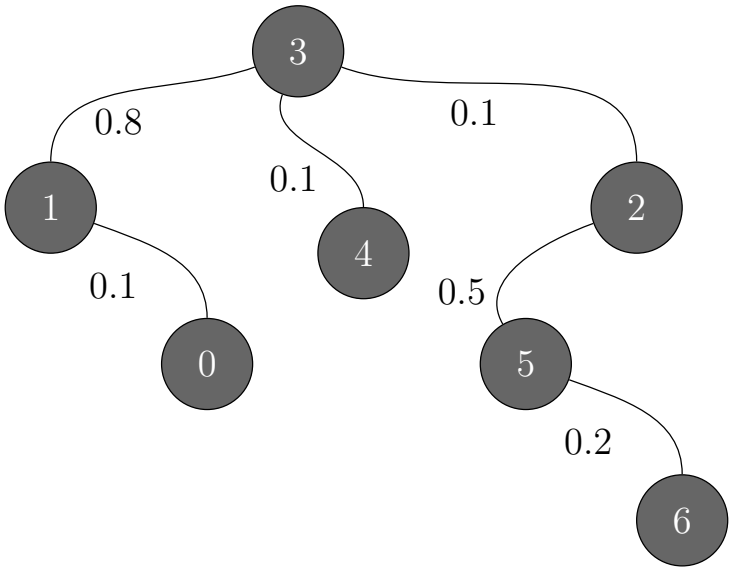


Figure 2.7: Shortest path tree root at object 3

Lemma 4 `SSLB-Graph`*takes* $\mathcal{O}(n^2)$ *to compute lower bounds of all unknown edges take time on sparse Graphs.*

Proof 4 `SSLB-Graph` *has three main computational costs, (i)*`SSLB-Tree` *, (ii)* `Top-Down Tree` *(iii)* `Bottom-Up Tree`. *The first component,* `SSLB-Tree` *takes* $\mathcal{O}(n^2)$ *time to complete as described in Section 2.3.2.1.* `Top-Down Tree`*, a recursive tree generation algorithm, works by ensuring that all the* cross edges *in the generated tree to contain a predetermined large edge. The recursive nature of the* `Top-Down Tree` *algorithm consumes* $\mathcal{O}(|E| + n\log(n))$ *time at each level. As depth of recursion is controlled by the parameter* d, `Top-Down Tree` *algorithm is performed* $2^d - 1$ *times. As the recursive depth increases the advantage in lower bound estimation reduces dramatically, while increasing the computation costs exponentially. To circumvent this, we limit the value of* d *to a small number like* 2. *Hence, the overall complexity of* `Top-Down Tree` *algorithm remains* $\mathcal{O}(|E| + n\log(n))$. `Bottom-Up Tree` *works by aggregating the trees at a lower level as unified tree by connecting the shortest edge thus far available for exploration. As the algorithm works on ascending edge weights, we perform a sort*

on the edge weights which is in the order of $\mathcal{O}|E|\log|E|$. *The overall complexity of merger operation, is in order of* $\mathcal{O}(|E|)$ *as demonstrated by the two for loops. However, number of times the merger operation takes place in* $\mathcal{O}(\log(n))$. *Which brings the overall complexity to* $\mathcal{O}|E|\log(n)$. *Thus the overall complexity of our* `SSLB` *Algorithm on graphs is* $\mathcal{O}(|E| + n\log(n) + |E|\log(n) + n^2)$. *Since the graph is sparse, the running time is dominated by the term* $\mathcal{O}(n^2))$, *which is better than* $\mathcal{O}(n^3))$ *running time of* `ADM`.

2.3.2.3 Upper Bound Estimation

We note that `ADM` computes upper and lower bounds of all the unknown edges at the same time, primarily because one bound is required to update the other. In our case, that is not necessary. As described in Section 3.3.2, the tightest upper bound of distance between o_i and o_j is the length of the shortest path distance between those objects as given in equation 2.1. Therefore, one can use classical Shortest Path algorithms, such as, *Dijkstra*'s [7] algorithm to calculate the shortest path from a given source to all objects of the graph which is in the order of $\mathcal{O}(|E| + n\log(n))$. To compute the tightest upper bound between o_i and o_j, we run Dijkstra's algorithm either from o_i or o_j, as the source. The complexity of this is $\mathcal{O}(|E| + n\log(n))$, since the graph is sparse, the second term dominates leading the complexity to $\mathcal{O}(n\log(n))$. This is obviously better than the$\mathcal{O}(n^3)$ running time of `ADM`. If the goal is to compute tightest upper bound for every unknown edge, then we run all pair shortest path, which will have the running time of $\mathcal{O}(n^2\log(n))$, and outperforms the running time of `ADM`.

2.3.3 Parameterized Neighborhood Search

In this section, we present a different type of algorithm (`LocalSPSearch`) that constrains the search space within a specified (by the parameter) neighborhood to obtain the lower and upper bounds of distance. Specifically, to estimate the lower and upper bound of distances between o_i, o_j, this algorithm takes an additional parameter t as input that specifies the *maximum path length of the local neighborhood* between o_i, o_j and confines its search inside this neighborhood for bounds estimation. Unlike the algorithms presented before, `LocalSPSearch` can trade-off between computation time and "tightness" of estimated bounds in a flexible manner, based on the value t.

`LocalSPSearch` performs a depth first search (DFS) [16] (DFS) from o_i to o_j of at most length t and then compute bounds. It's easy to see that by increasing the length of the path t, the running time of `LocalSPSearch` increases, while the produced bounds are likely to become "tighter". Also, when $t = n-1$, `LocalSPSearch` produces tightest bounds, identical to that of `ADM` [6].

Algorithm 7 Algorithm `TriSearch` : Finding upper and lower bounds considering Triangles

inputs: unknown distance pair o_i, o_j, graph $\mathcal{G}$, t

for $k = 1, 2, \ldots n-2$ **do**

Compute $UB^{d(o_i,o_j)|o_k} = d(o_i, o_k) + d(o_k, o_j)$

Compute $LB^{d(o_i,o_j)|o_k} = |d(o_i, o_k) - d(o_k, o_j)|$

end for

$UB^{d(o_i,o_j)} = \min_{\forall k} UB^{d(o_i,o_j)|o_k}$

$LB^{d(o_i,o_j)} = max_{\forall k} LB^{d(o_i,o_j)|o_k}$

Running time of `LocalSPSearch`**:** Worst case running time of `LocalSPSearch` is $O(|\mathcal{O}| + |E|)$. However, based on the value of t, the DFS will touch only a subset of nodes and edges - and hence is much faster in practice.

2.3.3.1 A special Case

It is obvious that the running time of this parameterized solution increases with increasing t. In our experimental analysis, we observe that, when $t = 2$, i.e., if we constrain the search to "triangles", only, i.e., looking for every possible triangles that involve o_i, o_j, and produce bounds based on that, even though the estimated bounds are not as tight' as `ADM` [6] or `SSLB`, but in practice, they are equally effective in saving distance calls. The advantage is that this triangle algorithm, we refer to as `TriSearch` is significantly lightweight in computational time and space and is well-suited to be invoked repeatedly.

Basically, `TriSearch` looks at every triangle between o_i and o_j and computes lower and upper bounds as described in the last two lines of Algorithm 7.

Algorithm `TriSearch` is extremely "lightweight" computationally and can produce tightest bounds in certain conditions (refer to the two lemmas below). Therefore, our experimental analysis only presents `TriSearch` results from the parameterized neighborhood based algorithms.

Lemma 5 `TriSearch` *runs in linear time on the number of objects.*

Proof 5 *We note that for an unresolved edge between o_i and o_j, the number of such triangles is at most $n - 2$, i.e., it is linear in the number of objects (recall $|\mathcal{O}| = n$). Since processing each triangle takes a constant amount of time, therefore* `TriSearch` *takes $O(n)$ times.*

2.4 Distance Estimation Algorithms inside Proximity Problems

In this section, we describe how different distance estimation algorithms (from Section 2.3) can be adapted to be used as a plug-in inside different proximity problems.

Orthogonality with proximity problems: The proposed distance estimation algorithms are applicable to save distance calls inside *any algorithm for proximity problems designed on general metric space, as long as they compare a set of distances with each other*. Thus our effort in designing these distance estimation algorithms and use that as a plug-in, in principle, is orthogonal to the specifics of any particular algorithm for proximity problems, as its general utility could be applied in saving the distance calls. To showcase the versatility of the plug-in, we consider three classes of proximity problems, namely, k-Nearest Neighbor Graphs, l-Medoid Clustering and Minimum Spanning Tree. We study the proposed plug-in in conjunction with different algorithms for these three problems, from classical to recent ones.

Principle of using bound estimation algorithms: The fundamental principle in using our bound estimation algorithms is to invoke these algorithms to compare two different unknown distances, such as, $LB^{d(o_i,o_j)}$ and $UB^{d(o_k,o_l)}$. Algorithms for the proximity problems then check, if $LB^{d(o_i,o_j)} > UB^{d(o_k,o_l)}$. If the condition comes true, then $d(o_i, o_j) > d(o_k, o_l)$ and the actual distance calls $d(o_i, o_j)$ and $d(o_k, o_l)$ can be saved. In fact, some proximity algorithms perform more complex distance comparisons than just comparing two distances. Generally speaking, given an arbitrary set of objects, they could be specified in the following form: **IF** $d(o_i, o_j) + d(o_j, o_k) + d(o_k, o_l) .. + \ldots > d(o_p, o_q) + d(o_q, o_r) + d(o_r, o_s) + \ldots$**, then take some actions**. As a concrete example, a partition-based clustering algorithm such as PAM, has to repeatedly decide whether a current medoid object is to be swapped with a non-medoid object, by comparing the sum of several distances before and after

the swap. This decision making process does not necessarily require actual distance computations. In fact, if PAM can figure out that the *lower bound of the sum of distances after the swap is greater than the upper bound of the sum of distances before the swap*, then this swap must not be made.

Even though we consider 3 different proximity problems (Refer to Section 2.2.3) and 5 classical algorithms, for brevity, we limit our detailed discussion to only one of these five algorithms, namely `KNNRp` algorithm [17], designed for the k-Nearest Neighbor Graph (k-NNG) Computation . Other 5 algorithms are briefly described in Section 2.5, before they are experimentally evaluated.

Algorithm 8 `KNNRp` algorithm [17],

Require: KNN (Integer k, ObjectSet $\mathbb{U}$)
Stage 1: Initialize NHA and construct the index I
for each u $\in \mathbb{U}$ **do**
$NHA_u \leftarrow \{(\perp, \infty), \ldots, (\perp, \infty)\}_k$
end for
Create $\mathcal{I}$, all computed distances populate symmetrically NHA
Stage 2: Complete the NN_k(u) for all u $\in \mathbb{U}$
$\mathcal{COH} \leftarrow \{(u, curCR_u), u \in \mathbb{U}\}$
for each $(u, curCR_u) \in \mathcal{COH}$, in increasing $(u, curCR_u)$ order **do**
Create the candidate set $\mathcal{C}$ according to $\mathcal{I}$ // exclude NHA_u
while $\mathcal{I} \neq \emptyset$ **do**
$c \leftarrow$ extract a candidate from $\mathcal{C}$
if (**then**$\mathbb{U}$ is fixed does not apply for u & c)
$d_{u,c} \leftarrow d(u, c)$, try insert c into NHA_u
try to insert u into NHA_c, update c in COH(symmetry)
use NHA_u as a graph and $\mathcal{I}$ to discard objects from $\mathcal{C}$
end if
end while
end for
return NHA as a graph

Algorithm 9 `KNNRp + Plug-in` algorithm

Require: KNN (Integer k, ObjectSet $\mathbb{U}$)
Stage 1: Initialize NHA and construct the index I
for each u $\in \mathbb{U}$ **do**
$NHA_u \leftarrow \{(\perp, \infty), \ldots, (\perp, \infty)\}_k$
end for
Create $\mathcal{I}$, all computed distances populate symmetrically NHA
Stage 2: Complete the NN_k(u) for all u $\in \mathbb{U}$
$\mathcal{COH} \leftarrow \{(u, curCR_u), u \in \mathbb{U}\}$
for each $(u, curCR_u) \in \mathcal{COH}$, in increasing $(u, curCR_u)$ order **do**
Create the candidate set $\mathcal{C}$ according to $\mathcal{I}$ // exclude NHA_u
while $\mathcal{I} \neq \emptyset$ **do**
$c \leftarrow$ extract a candidate from $\mathcal{C}$
if $LB[u][c] > curCR_u + c_{pr}$ **then** continue
if (**then**$\mathbb{U}$ is fixed does not apply for u & c)
if (**then** $LB[u][c] < curCR_u$)
$d_{u,c} \leftarrow d(u, c)$, try insert c in NHA_u
try to insert u into NHA_c, update c in COH(symmetry)
end if
use NHA_u as a graph and $\mathbb{I}$ to discard objects from $\mathbb{C}$
end if
if $UB[u][c] > curCR_u + c_{pr}$ or $d_{u,c} > curCR_u + c_{pr}$ **then**
Discard descendants of c from $\mathbb{C}$
end if
end while
end for
return NHA as a graph

2.4.1 Plug-in Inside `KNNrp`

[17] `KNNrp`, a popular k-NNG proposed in [17] computes the k-NNG graph in two steps. At first, it builds a binary tree like index structure by keeping objects that are close as part of the same sub-tree. Then, it explores the index to compute the exact k-NNG. Algorithm 8 presents the verbatim pseudo-code of this prior work and Algorithm 9 shows its adaptation considering our proposed plug-in. The main distance calls save up takes place place in the exploration step, which we describe in detail.

During the index exploration phase, `KNNrp` maintains three key information about each object u, (i) $curCR$ - distance of k^{th} object from u (ii) NHA - the top-k nearest objects that node has seen so far (iii) u_{pr} - the distance of a node to its known farthest $d$escendent object. As the exploration phase proceeds we explore the object u with the smallest $curCR$, which are maintained in a heap COH and find a candidate set to compare u. The algorithm prunes the candidate set using the index to save computation cost.

Using our solution as a plug-in, `KNNRp + Plug-in` judiciously deciding which of the unknown distances that span across sub-trees, must be queried to compute the k-NN of an object and which distances could be estimated. `KNNRp + Plug-in` makes the following comparison invoking our distance estimation algorithms: (i) by checking if the LB value of an unknown edge is larger than the $curCR_u + u_{pr}$, meaning, if the LB value of an unknown edge from u is bigger than first NHA node and other end of the edge's $_{pr}$, we can safely prune the node and its children from further exploration, similarly, (ii) we discard all descendants of a node from exploration, if we find, the upper bound of the distance is larger than the $curCR_u$ and u_{pr} or if the actual distance of the edge is more.

2.5 Experimental Analysis

Our experimental analysis are run on a machine with Intel Core i7 4GHz CPU and 64GB of memory with Linux operating system and Python 3.6.8 used for algorithmic development. All numbers are presented as the average of ten runs.

2.5.1 Datasets and Experimental Setup

We first present the used datasets, experimental setting, and evaluation measures. **Datasets:** We use 6 different datasets by varying properties that are appropriate for the various proximity problems under investigation and summarized in Table 2.1. For each dataset, the actual pairwise distances (i.e., ground truth) are known. For generating the topology of synthetic networks we used the network generator package, igraph[3]. We sample 4 kinds of graphs namely Barabasi, Renyi Erdos, Geometric as well as Forrest Firing.

Dataset	Algorithm	Num. Objects	Num. of Edges	Dimension	I
Maps Dataset	k-NNG, Clustering, MST	$10k$	49995000	2	C
20 Newsgroups Dataset	k-NNG, Clustering	18846	177576435	128	I
Flicker1M	k-NNG	$10k$	49995000	256	I
Synthetic Uniform	kNNG, Clustering, MST	$10k$	49995000	4	I
Synthetic Gaussian	kNNG, Clustering, MST	$10k$	49995000	4	I
Synthetic Zipfian	kNNG, Clustering, MST	$10k$	49995000	4	I

Table 2.1: Dataset description

Goals: The goal of our experimental analysis is to provide insights to the following questions:

(Subsection 2.5.3.) Sensitivity analysis of our proposed distance estimation al-

[3]https://igraph.org/python/

gorithms and their comparison with the state-of-the-art `ADM`. We compare them in produced distance bounds and running time.

(Subsection 2.5.4.) Application of our proposed bound estimation algorithms as a plug-in inside 5 different proximity algorithms (Described in Subsection 2.5.2). We measure two things in this context - %distance call save up and the corresponding running time. For brevity, we only present a subset of results that are representative.

2.5.2 Implemented Proximity Algorithms

We evaluate 5 different algorithms from three classes of proximity problems, namely - (a) k-NNG Computation: `KNNRp` [17] (which is described in depth in Section 2.4) (b) Minimum spanning tree algorithms (MST) : (ii) $Prim$' [5], (iii) $Kruskal$'ss [18] , and (c) Clustering Algorithms: (iv) PAM [19] and (v) $CLARANS$ [20].

In addition to `KNNRp`, described in detail in section 2.4, here we briefly describe how other classical algorithms adapt our solutions as plug-in.

MST: $Prim$'s algorithm generates a MST when provided with a graph and a starting index. It maintains a sketch from which it chooses the next edge to add the graph, our distance estimation algorithms are plugged in to decide how to extend the graph. Whereas, MST by $Kruskal$'s algorithm is generated by adding the least weighted edge to the graph that does not form a loop. Our algorithms are used to compare the top edge contenders (based on LB) and determine whether to add the edge or explore further.

Clustering: Inside l-medoid clustering algorithms, PAM and $CLARANS$ are algorithms are using to decide whether the current centroid is to be swapped or not.

2.5.3 Sensitivity Analysis of the Distance Estimation Algorithms

Our main investigation here is to compare `SSLB-Graph` (abbreviated in the Figures as SSLB), `TriSearch`(abbreviated in the Figures as TS) with `ADM` in produced distance bounds and running time.

Our experimental results indicate that `SSLB-Graph`*is significantly faster than* `ADM`*, while producing comparable distance bounds. These results corroborate our theoretical claims - that is* `SSLB-Graph`*is quadratic on sparse graphs, whereas,* `ADM` *is cubic.* `TriSearch`*, on the other hand, being a heuristic, produces much looser bounds comparing to the other two algorithms, while being the fastest. Again this observation is in conformance with our theoretical claims.*

2.5.3.1 Tightness of bounds and running time

Since `ADM` is not scalable owing to its cubic nature, we consider at most $n = 1024$, and vary % of known edge (from 0.01%) to 0.52%). We compare the algorithms in difference in distance bounds (both upper and lower), and running time. For brevity, we only present the graphs on lower bounds, as `SSLB-Graph` is designed for lower bound. We note that the upper bound calculation follows identical trends.

Figures 2.8 & 2.9 present both LB and running time of these algorithms. First and foremost, we observe that as objects in the dataset increases, there is virtually no distinguishable difference between the bounds of `SSLB-Graph`and `ADM`, while `TriSearch`is significantly inferior in producing bounds. Another important observation is in the running time of these algorithms. Figure 2.8 presents the average running time in generating the bounds for an unknown edge. As seen from the bottom figure in Figures 2.8 & 2.9(in orange line), one can easily understand the cubic nature of `ADM` and also ability of our solutions in providing a relatively tight lower bound while

being computationally much faster. `SSLB-Graph`takes more than 34 hours to complete when there are $4k$ objects. Due to this cubic nature of `ADM` , comparison with with larger set of objects is not possible.

In figure 2.10 the average lower bound for these three algorithms are presented by varying the percentage of known edge. As it could be seen, `ADM` exhibits very high running time, as the graph becomes more dense. It also presents the running time corresponding to different sizes of graphs. As expected, `TriSearch` performs the worst in producing bounds but comes out to be the fastest choice in running time.

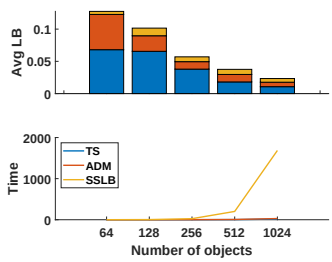


Figure 2.8: LB comparison uniform distribution

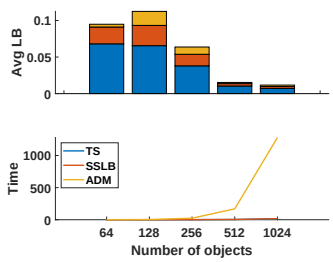


Figure 2.9: LB comparison normal distribution

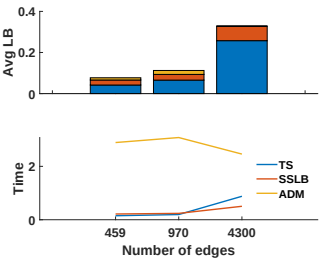


Figure 2.10: Varying % of known edge

2.5.4 Distance Estimation Algorithms As Plug-in

Our main investigation here is how our distance estimation algorithms perform as plug-in inside various proximity algorithms in saving distance calls and running time. We present distance calls as well as CPU time in seconds. In addition to the three algorithm, only for the comparison of running time, we implement the original versions of the algorithms without the plug-in, that we refer to as ORC.

These experiments confirm that proximity algorithms augmented with our bound estimation algorithms as plug-in shows on average $5\times$ *and as much as* $18\times$ *improve-*

ment in total number of distance calls, while incurring minimal to no additional CPU time.

Parameters: We intend to vary k and n for the k-NNG, n and l (the number of medoids) for the clustering problems, and only n for the MSTs. The default values of $n = 128$, $k = 5$. Running times are measured in seconds unless explicitly stated.

2.5.4.1 Evaluation of k-NNG

The objective of these set of experiments is to compare `KNNrp` and `KNNrp+Plug-in` in saving distance calls and CPU time varying different parameters. The former measure is presented as the proportion of the number of calls the algorithms make using the plug-in vs the original quadratic number of distance calls.

Varying n: We vary the number of objects to study the effect of number of distance comparisons and the time taken for completing the algorithms. As seen in Figure 2.11, one can clearly see the effect of varying n and its impact total distance computation. Both `SSLB-Graph`and `ADM` save about 70% distance calls, whereas, `TriSearch`performs quite poorly.

Varying k: Figure 2.12 also shows the effect of parameter k on the number of distance calculations and running times. As [17] corraborates, the parameter k does not have an effect on the number of distance calculations. Again `ADM` and `SSLB-Graph`perform equally well varying k, but `ADM` is significantly slower in running time.

2.5.4.2 Evaluation of Clustering

The objective of these set of experiments is to compare the two l-medoid algorithms `PAM` and `CLARANS` with their augmented versions with our proposed algorithms and `ADM`. Our measures of interests are same as before.

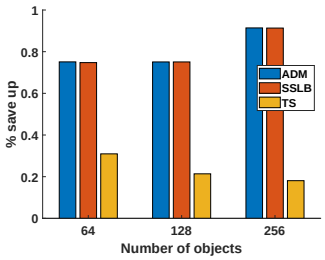


Figure 2.11: kNNrp Varying n : Distance Save up

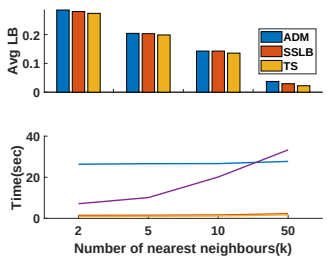


Figure 2.12: kNNrp: Varying k

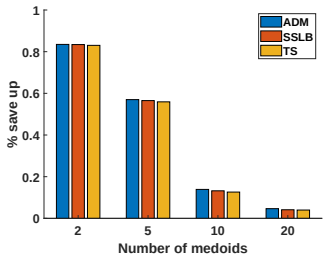


Figure 2.13: CLARANS Varying k : Distance Save up

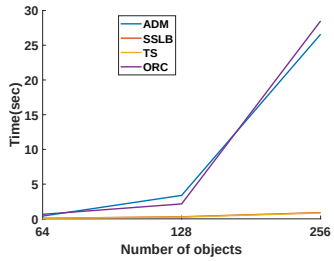


Figure 2.14: CLARANS Varying n : running time

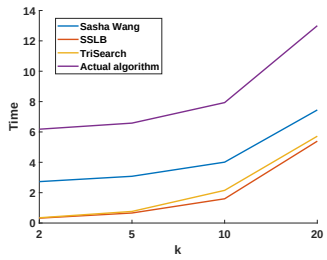


Figure 2.15: CLARANS Varying k : running time

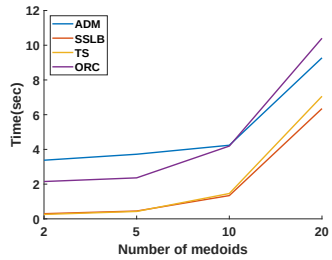


Figure 2.16: PAM Varying Medoids : running time

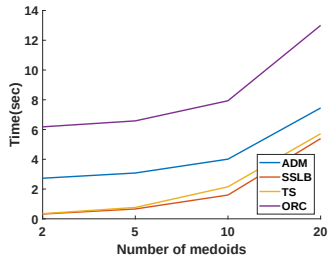


Figure 2.17: PAM Varying Medoids : running time

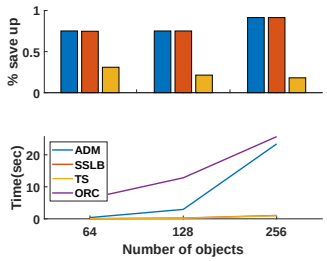


Figure 2.18: Prim's Varying n : Distance save ups and running time

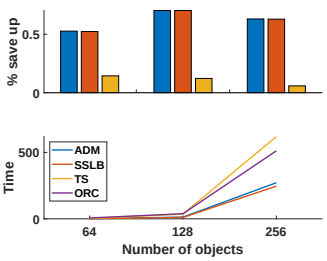


Figure 2.19: Kruskal's Varying n : Distance save ups and running timel

Varying n: Figures 2.13 present the effect of number of objects on the two algorithm `PAM` and `CLARANS` with their augmented versions. Overall, algorithms augmented with our plugin use on average one third the number of distance calculations.. For both algorithms, as the number of objects grows, the number of distance calculations also increases. Figures 2.16 present the running times of the algorithms. Notice

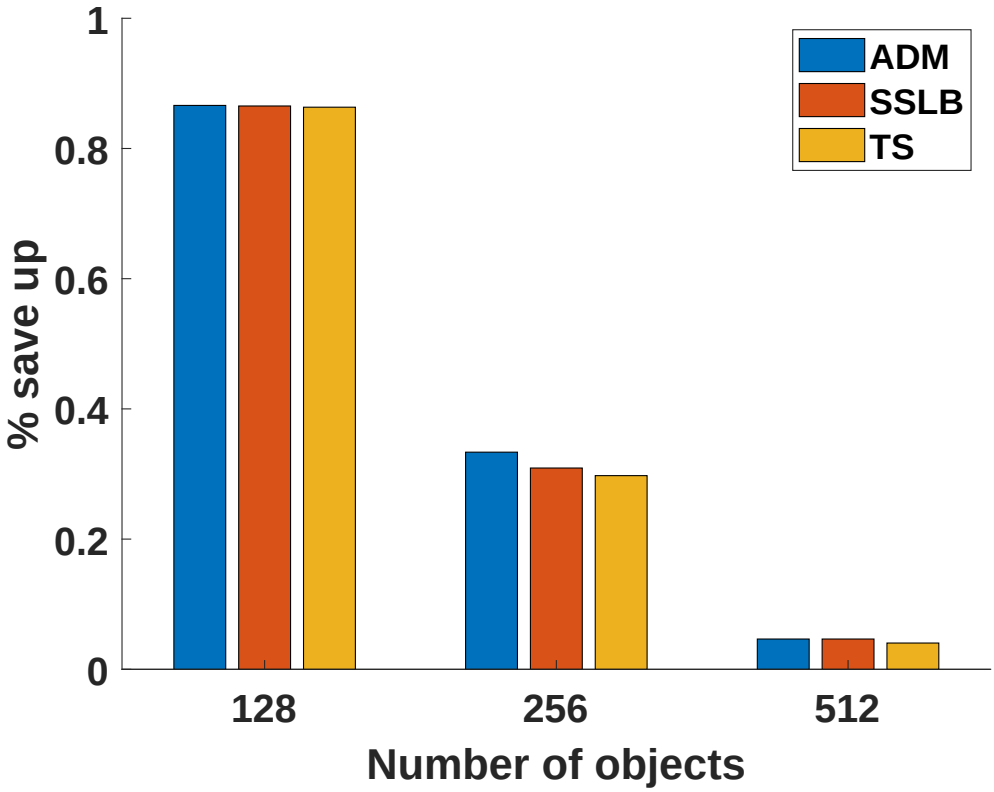


Figure 2.20: CLARANS Varying n : Distance save ups

that the augmented algorithms perform very similar to the original algorithms and on average it save of about 10× time saving for the `SSLB-Graph`plugin on the original algorithms.

Varying l: In Figure 2.13 2.20 show the effect of parameter l on the number of distance calculations and running times. For the `PAM` algorithm, as the number of clusters increase, the number of distance calculations decreases until $l = 25$. As the number of objects is fixed, increasing the number of clusters will results in more local minima for the `PAM` algorithm which in turn makes the algorithm converge faster since it cannot find new candidates for changing the centers. `CLARANS` on the other hand does not suffer from this phenomena and as the number of clusters increases

the number of distance calls also increases. Overall the two augmented algorithms perform better by a factor of 2.3 without incurring more than 1 seconds of CPU time.

2.5.4.3 Evaluation of MST Algorithms

In this section we compare `Kruskal's` and `Prim's` algorithms for the MST problem. Figure 2.19 2.18 presents these results. As expected, the original algorithms need to exhaustively calculate all the distances required to build the graph in order to finish the minimum spanning tree. Augmented algorithms on the other hand only need a small portion of the distance (most of the time less than 10%). This gives the augmented algorithm a huge advantage in terms of number of distance calls. As presented in Figure 2.18, the running time of `Prim's` algorithm is not affected by the plug-in (increase of at most 1.2 times the original algorithm). This does not hold for `Kruskal's` algorithm and the running time increases by a factor of 1.7.

2.6 Related Work

Exact solutions for the proximity problems in metric space can be divided into two broad categories according to a popular survey paper [21].

I. Transforming Metric Space into Vector Space: The *first kind* transforms the metric space into a vector space by transforming each object to a *finite-dimensional vector*. The distance function is then defined based on the distance between these vectors. Some of the most useful distance functions are Euclidean distance, Manhattan distance, or L_∞. Popular solutions in this space include *kd-tree* [22],*R-tree* [23], or more recent *X-tree* [24]. These approaches usually use the coordinate information of the objects. *We do not investigate this genre of work, as they do not directly optimize distance calls.*

II. Dealing with Metric Space Directly: The *second kind* deals with metric space directly and can further be divided into different sub-categories depending on their algorithmic offering. Many of these are limited to Euclidean space only and do not extend to general metric space.

II.a. Pivot based solutions: The first kind selects a set of pivots in order to divide the space into smaller sub-spaces, essentially grouping similar objects together. In pivot based approaches, like BKT [25],FQT and FHQT [26, 27],VPT [28],GNAT [29] FQA [30], Bisector Tree [31],M-Tree [32], Voronoi Tree [33], the objective is to find a subset of points acting as pivot and dividing the space into sub-regions of objects that are closer to each pivot. As an example, VPT or vantage point tree recursively chooses the pivots based on the median distances. In M-Tree, all the objects are stored in leaf nodes and for each subtree, an object is chosen as the representative.

There are two fundamental assumptions about these solutions that makes them incompatible to our work. The first one is the assumption that the object set is different from the query set, and secondly these approaches choose the pivot points based on the query object.

II.b. Matrix based solutions: The second set of algorithms for metric space are algorithms that use a matrix to keep track of the edges between the objects. The representative algorithms in this sub-class are AESA and LAESA [34, 35] which take the idea of pivot based methods to the extreme. It calculates all pairwise distances between objects and store them in a matrix. This matrix is then used during query time to expedite processing time. *These works are not designed to minimize distance calls. In fact it has been shown empirically that the number of calculated distances for AESA is constant.*

II.c. Solutions to minimize distance calls: [6] assumes that the prominent cost of solving proximity problem lies in making distance calls, thus propose `ADM` to

estimate tightest upper and lower bounds of distances that are then demonstrated to save up different querying cost. *Thus, we compare* `TriSearch` *with* `ADM` *to demonstrate their difference in saving distance calls and computational overhead.*

2.7 Conclusion

In this paper, we propose a *general purpose plug-in* for estimating distance bounds in proximity problems in general metric spaces. We present a suite of algorithms for estimating these bounds that are computationally "light weight", starting from algorithms that constraint the search space to local neighborhood of an unknown edge to the ones that do not. One important technical result of our work includes a new lower bound estimation algorithm that produces the tightest bound (when the underlying weighted graph of known edge weights form a tree) and improves the previously known best algorithm for lower bound estimation in asymptotic running time. After that, we show how classical algorithms for proximity problems can non-trivially adapt the proposed plug-in and save distance calls. We experimentally analyze the proposed plug-in in 3 different proximity problems (k-NNG construction, clustering, minimum spanning tree construction), considering 5 classical algorithms, using 6 different datasets. Our experiment results demonstrate an order of magnitude reduction in the number of calls to the distance oracle (average $5\times$, maximum $20\times$), at the expense of comparatively small increase in the rest of the computation costs.

CHAPTER 3

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

3.1 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. For example, computing the distance between a pair of objects may involve running expensive computer vision algorithms, involving actual human experts (inferring dissimilarity between two CAT-SCAN images may be hard to capture using a distance function and may require human involvement), or may even require calls to third party applications (e.g., map APIs) which may impose monetary costs per number of queries or put constraints on query rate, e.g., limit the number of queries a user can make in a given time period.

In an abstract sense, we assume access to a *distance oracle* which is an expensive function that takes as input any pair of objects and outputs the distance between them. Our objective is to consider existing algorithms for various classical proximity problems, and propose minor redesign of these algorithms such that they are refocused towards minimizing number of distance calls at the expense of local CPU computations.

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 added to prior algorithms, resulting in significant reduction in the number of calls to the distance oracle, at the expense of comparatively small increase in the rest of the computation costs. Our framework is very general, and the only assumption we make about the unknown distance function is that it should be a metric, i.e., it satisfies the triangle inequality or relaxed triangle inequality property - these assumptions are intuitive and justified in a wide variety of real-world applications [36, 37, 38, 39, 40]. Moreover, we emphasize that our solution framework does not change the outputs of the original algorithm. The same outputs are produced, but with fewer distance calls.

The heart of 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 this algorithm to run correctly, 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 this is determined to be true without having to invoke the distance oracle, then v can be safely discarded, thus saving on distance calls.

The design of efficient algorithms for various classical proximity problems have been investigated for decades (an elaborate discussion on the related body of work is deferred to Section 3.6). However, the characteristics of most of these algorithms are limited to being specific to the proximity problem being considered, and do not easily generalize across different proximity problems. Secondly, the algorithmic optimizations are blended and do not separate the expensive distance calls minimization from local computations.

3.1.1 Technical Contributions

Contribution 1: Linear Program Modeling (Section 3.2.2): 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 modeling them as *a system of linear inequalities*. We identify that such `IF` statements form the heart of the proximity algorithms and are the key steps where distance computations appear. 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 that involves expressing the problem as a system of linear inequalities which can be solved by only using local CPU resources (Section 3.2). To the best of our knowledge, no prior work has presented this formalism before.

Contribution 2: Graph-Theoretic Modeling and Efficient Algorithms (Sections 3.3 and 3.4): 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 $dist(u, w)$ without invoking any distance calls, 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 make several algorithmic contributions, among them 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 lower bound algorithm.

As a second algorithm contribution, we present an optimized "lightweight" bound estimation algorithm. The "lightweight" solution framework is highly scalable, by constraining to search to a local neighborhood of the graph, limiting the search to paths of length 2. We refer to this algorithmic scheme as the *Triangle Based Solution Scheme* (`Tri Scheme` in short). In addition to defining a lightweight scheme, we also present an expected case analysis for `Tri Scheme` in section 3.4.2.2

These latter algorithms compromise the tightness of estimated bounds (and are thus suboptimal compared to the LP approach for saving on distance calls), but enable higher computational efficiency which can make them practical for certain application scenarios. To summarize, we present a suite of algorithms that make a trade-off between savings on distance calls and computational efficiency.

Contribution 3: Extensive Experimentation (Section 3.5): Our final contribution lies in performing extensive experiments and outperforming the appropriately adapted current-state-of-the-art proximity solutions with the help of real world datasets. Besides demonstrating algorithmic efficiency, our experiments also high-

lighted the ease with which our proposed re-authoring methods can be adapted for a wide class of proximity algorithms.

3.2 Distance Cost Minimization in Proximity Algorithms

In this section, we study how existing proximity algorithms incur distance cost inside computational loop and propose a general purpose model DIRECT FEASIBILITY TEST to minimize that cost.

3.2.1 General 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 decision.

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

```
}
```

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

Consider a set, $\mathcal{O} = \{o_i, o_2, \ldots, o_n\}$ of n objects. We assume no two objects in $\mathcal{O}$ 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 $(m_i, m_j, m_k) \in \mathcal{M}$,

$$dist(m_i, m_j) = 0 \implies (m_i = m_j)$$

$$dist(m_i, m_j) = dist(m_j, m_i)$$

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

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).

Our goal, through this work, is to develop general computational techniques that leverage the triangle inequality and save actual distance calls to the expensive oracle inside different proximity problems.

Example 2 *(*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.*

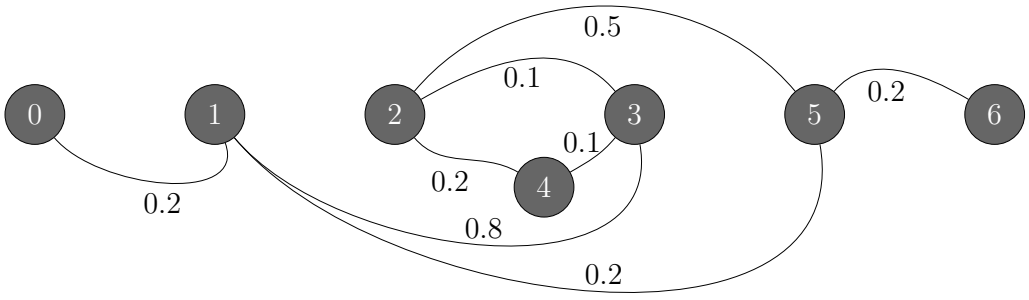


Figure 3.1: 7 objects and their corresponding known and unknown distances.

The running example is shown in Figure 3.1. 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 are known.

3.2.2 Direct Feasibility Test

The triangle inequality relationship among the objects could be represented using a set of inequalities - in fact, involving all triangles induced by the objects in $\mathcal{O}$.

Using the example in Figure 3.1, 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_3), 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:

$$AX \leq b \tag{3.1}$$

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 [41] could be used to solve this problem. However, the number of iterations for SIMPLEX in the worst case is exponential in number of objects [42]. A more practical approach to linear programming through the ellipsoid algorithm [43] also could be used. However, solving linear inequalities through this method is in $\mathcal{O}(n^6)$. These algorithms thus are not practical even for small number of objects.

3.3 Graph Theoretic Approach to Distance Cost Minimization

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 small increase in 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 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.

3.3.1 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 ($\mathcal{O}$), and every pair of nodes in the object set, (o_i, o_j) forms the edges in graph whose edge weights are induced by the distance function, $dist(o_i, o_j)$. As before, the distance between the objects satisfy 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 [6] 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 $dist(o_i, o_j)$ but which are not as tight as $TUB^{dist(o_i,o_j)}$). In the rest of the paper we refer to them as $UB^{dist(o_i,o_j)}$.

$$TUB^{d(o_i,o_j)} = sp(o_i, o_j) \tag{3.2}$$

$$UB^{d(o_i,o_j)} \geq sp(o_i, o_j) \tag{3.3}$$

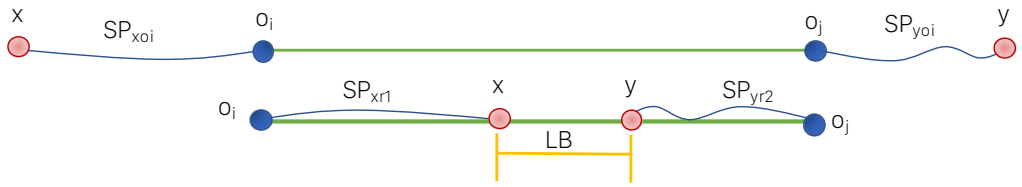


Figure 3.2: Geometric Interpretation of LB. [Top] Shortest Paths through Known Edge. [Bottom] Wrapping SP onto Known Edge.

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 [15] - 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 $LB^{dist(o_i,o_j)}$.

$$TLB^{dist(o_i,o_j)} = Maximum_{\forall p}\{dist(o_k, o_l) - path(o_i, o_k) - path(o_l, o_j)\} \tag{3.4}$$

$$LB^{dist(o_i,o_j)} \leq \forall p\{dist(o_k, o_l) - path(o_i, o_k) - path(o_l, o_j)\} \tag{3.5}$$

To explain further, we refer to Figure 3.2 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 3.5 in the light of the figure as shown by 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 3.1 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, only 8 out of 21 $\binom{7}{2}$ edges are known (the solid lines), while the remaining 13 edges are unknown. As given in figure, if the distance $dist(1,\ 3) = 0.8$ and $dist(3,\ 4) = 0.1$ then the 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.,

$$0.7 \leq dist(o_1, o_4) \leq 0.9$$

3.3.2 Problem Definitions

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

Problem 3 (BOUNDS PROBLEM) *: Given a partial graph, $\mathcal{G}(\mathcal{O}, E)$, and an unknown edge (o_i, o_j) in 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, $\mathcal{O}$.*

For instance, from the discussion following Example 2, 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 4 (UPDATE PROBLEM) *: Given a partial graph, $\mathcal{G}(\mathcal{O}, E)$, 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 section, we present multiple solutions that trade-off between tightness of produced bounds and running time to solve the 3 and 4 problems.

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

Discussion - Running Example - Let us take the example of same two unknown edges (o_2, o_6) and (o_3, o_5). Let us also assume an IF statements in proximity problems needs 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 $0.3 < 0.6$, 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 $dist(o_3, o_5)$.

3.4.1 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.

3.4.1.1 Algorithm Development:

Recall from Definition 1 that, in any given graph, the tightest upper bound of 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 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 from Dijkstra's Algorithm [7].

To find out the TUB between an unknown pair of edge $(o_i,\ o_j)$, we start a shortest path algorithm from one end point lets say from o_i to find the shortest path the to the other end point o_j. This in turn solves the problem of upper bound using the Equation 3.3.

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 complete two parallel shortest path algorithms from each end point 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 end points of the unknown edge to both the end points of the known edge. Since the tightest lower-bound is largest of the all available lower-bound distances, we only keep track the current largest value at each iteration.

Lemma 6 *The bound computed by the Lower Bound Algorithm in* `SPLUB` *, produces exact tightest lower bounds for the unknown edge.*

Proof 6 *Assume that we do no produce the tightest lower bound on 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 the all pairs shortest paths. In subsequent steps, 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 it length. From Equation 3.5 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 is 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 10 as `SPLUB` .

Algorithm 10 `SPLUB`

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

Output : $LB^{d(o_i,o_j)}$, $UB^{d(o_i,o_j)}$

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:

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

6: **end for**

7: $ub = min(ub, sp_{o_i}[o_j])$

8: $LB^{d(o_i,o_j)} = lb$

9: $UB^{d(o_i,o_j)} = ub$

10: **return** $LB^{d(o_i,o_j)}, UB^{d(o_i,o_j)}$

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 end points of the unknown edge. The Dijkstra's Algorithm [7] with its standard implementation taken $\mathcal{O}(m + n \log n)$ time to run. Thus we estimate the overall running time of the combined steps 2 and 3 as $\mathcal{O}(m + n \log n)$.

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 $\mathcal{O}(m + n \log n) + \mathcal{O}(m)$. The leading term is the first term and thus we claim the overall running time of the algorithm to be $\mathcal{O}(m + n \log n)$.

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 $\mathcal{O}(1)$.

3.4.2 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? Given a newly resolved edge, 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 3.5, it is evident that the bound obtained will not be tight. As an example, if we consider only the path $[o_1 \rightarrow o_3 \rightarrow o_2 \rightarrow o_4]$ to compute the lower bound on the

unknown edge (o_1, o_4), the lower bound will be 0.5. For all the unknown edges whose upper bounds are lesser than 0.5 can still be eliminated by this bound.

It is important point here to note 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. In the spirit of the above discussions, we propose a highly scalable yet effective heuristic `Tri Scheme` is designed with these two arguments in mind.

3.4.2.1 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 triangles 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 end points 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 $\mathcal{O}$*(size of the list)*. In adjacency list corresponding to each node in the graph, we use a balanced binary search tree[16] to make comparisons faster. However this scheme has increased the

new edge insertions to be in $\mathcal{O}(\log(n))$, which updates two binary search trees one corresponding to each end points of the resolved edge in the adjacency list with the edge value.

The pseudo-code of `Tri Scheme` is presented in Algorithm 11. As one can see that the computational bottle neck from `SPLUB` are entirely avoided to generate a simpler and practical algorithm. We present some theoretical properties of `Tri Scheme` next.

3.4.2.2 Expected Case Analysis for `Tri Scheme`

Comment 1 ***Theorem 1*** *Average-Cases Time Complexity of* ***`Tri Scheme`*** *is* $\mathcal{O}(m/n)$

Proof 7 *The average-case complexity is the amount of computational time used by the algorithm, averaged over all possible inputs. The graph contains of* m *edges with known values which represent the distance between the two connecting objects and the values for the rest* $n^2 - m$ *edges are not known. The numerator of the average case analysis consists of the number of edges accessed overall possible graphs. The optimization problem that represents the numerator is,*

$$maximize \sum_{i=1}^{n} d_i(n - d_i)$$
$$s.t. \sum_{i=1}^{n} d_i = 2m$$

The optimization term maximizes when the negative term is minimum. Hence, the numerator's optimization term can be written as,

$$maximize \sum_{i=1}^{n} nd_i - d_i^2 = n\sum_{i=1}^{n} d_i - \sum_{i=1}^{n} d_i^2$$

As, d_i^2 has to minimized, value of $d_i = (2m/n)$ can be used.

$$\begin{aligned} max\ n\sum_{i=n}^{n} d_i - \sum_{i=n}^{n} d_i^2 &\leq 2nm - \sum_{i=1}^{n} (\frac{2m}{n})^2 = 2nm - \frac{4m^2}{n^2}\sum_{i=1}^{n} 1 \\ &\leq 2nm - \frac{4m^2}{n} = \frac{4m}{n}(\frac{n^2}{2} - m) \\ &\leq \frac{4m}{n}(n^2 - m) \end{aligned}$$

The denominator consists of the number of edges that are not known in the graph($n^2 - m$), each of which is a potential input. Hence, the average case analysis is,

$$\frac{\frac{4m}{n}(n^2 - m)}{n^2 - m} = \frac{4m}{n} \in \mathcal{O}(\frac{m}{n})$$

Hence, proved.

Theorem 2 *Expected running time for* `Tri Scheme` *to lookup an edge is $\mathcal{O}(m/n)$*

Proof 8 *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 edge could be queried next with equal probability (uninformed prior) by the proximity algorithm there is 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,

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

In order to create an adversarial case, we would like to maximize the above formula to obtain a 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.

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

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, we get,

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

Thus, proved.

Bootstrapping `Tri Scheme` through Landmarks: In section 3.4.2, we have developed `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` [35] 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 bounds on the rest. Recall our problem setting described in section 3.3.1 that assumes m edges are resolved at the beginning of the algorithm. We basically use an initialization of the graph $\mathcal{G}$ by priming it with `LAESA` inside every proximity algorithm, for different values of m. Later in experiment section 3.5 we shall show the effectiveness of our schemes due to this initialization.

3.5 Experimental Evaluation

Algorithms are developed in Python 3.6 and the experiments are conducted on a Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz running a Linux distribution, Ubuntu 18.04.5 LTS using 64 GB.

3.5.1 Experimental Setup

In this section, we describe used datasets, implemented baselines, studied proximity problems, and our performance measures.

3.5.1.1 Datasets

We use 3 real datasets and vary different parameters considering various proximity problems and summarized in Table 3.1. For each dataset, the actual pairwise distances (i.e., ground truth) are known.

Dataset	Algorithms	# Objs	# Edges	Dim	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 3.1: Dataset Description

Comment 2 *(I)* California Road Network and Points of Interest *[44]: The dataset contains the longitude and latitudes of 21048 POIs in California. The data is collected and curated by the University of Utah and is available for public access [45]. Each of the POI has the following attributes in the database, Node ID,* `Longitude` *and* `Latitude`*.*

(II) The UrbanGB Dataset *[46]: The dataset contains the longitudes and latitudes of the road accidents within Great Britain urban areas. The data is obtained from UCI Machine Learning repository and is accessible for public download [47]. The dataset has the longitudes and latitudes of 360,177 accidents. Each record in the database has two attributes* `Longitude` *and* `Latitude`*.*

(III) Flicker 1M Dataset *[48]: This repository is a collection of thumbnail images downloaded from the MIRFLICKR database. The repository [49] has a collection of 1M thumbnail images which were used in establishing the effectiveness of our methods. Each of the object is an image represented, in* `.jpg` *image compression representation.*

3.5.1.2 Implemented Baselines

(1) We implement `ADM` [15] algorithm which provides exact upper, lower bounds and updates induced as a solution scheme. Throughout the experiment section, we refer to the baseline algorithm as, `ADM` .

Even though not a direct competitor, we also implement landmark based algorithm `LAESA` [35].

As described in Section 3.6, to the best of our knowledge, these are the only existing solutions that could be adapted to solve our problem.

These baselines are compared with (i) `SPLUB` in section 3.3 and, (ii) `Tri Scheme` in section 3.4.2. In cases where we use landmarks towards the graph initialization, we use $k = \log(n)$ landmarks unless otherwise mentioned.

3.5.1.3 Proximity Algorithms

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

ⓘ k Nearest Neighbor Graph (k-NNG) Construction: We implement `KNNrp`, a popular and recent k-NNG proposed in [17] that computes the k-NNG of a given set of objects.

ⓘⓘ MST: We implement the popular $Prim$'s [7] and $Kruskal$'s [18] algorithm for evaluation.

ⓘⓘⓘ Clustering: We implement two popular centroid based swapping algorithms, PAM [50] and $CLARANS$ [20].

3.5.1.4 Experimentation Goals

The goal of our experimental analysis is to provide insights to the following questions:

(Subsection 3.5.2) Comparison between proposed graph theoretic techniques (`SPLUB` and `Tri Scheme`), and compare with `ADM` and `LAESA` , on the following parameters. (i) Quality of bounds and, (ii) In computation time.

(Subsection 3.5.3) Comparison between `Tri Scheme` and `LAESA` , in saving distance calls for various proximity algorithms.

(Subsection 3.5.4) Comparison between `Tri Scheme` , `LAESA` , and the original algorithm in overall running time by varying the cost of distance oracle.

(Subsection 3.5.5) Varying proximity algorithms parameters l and k and its effect on CPU overhead and Distance Calls.

3.5.1.5 Evaluation Measures

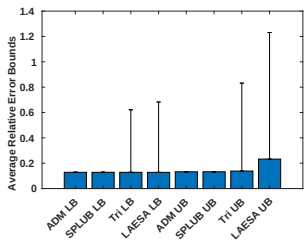
Our main investigation here is to study how `Exact SP` and `Tri Scheme` compare with `ADM` and `LAESA` in producing distance bounds, as well as their effectiveness in saving distance calls and overall running time inside different proximity algorithms. 1 *Relative Error & CPU Overhead* We present relative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

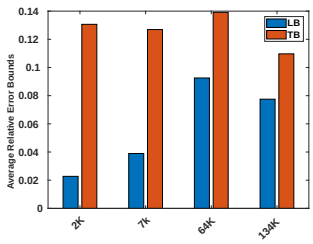
Additionally, we present some deeper analysis that compares `Tri Scheme` with `LAESA` and `ADM` qualitatively and running time-wise.

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

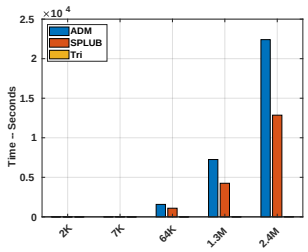
Summary of results: Our experiment results indicate that `SPLUB` produces *tightest and exact* bounds as `ADM`, yet significantly faster than `ADM`. While `ADM` is only useful in smaller graphs, `SPLUB` is useful for moderate sized graphs. `Tri Scheme` is significantly faster in running time compared to `SPLUB` with comparable quality of bounds and is practical. Compared to `LAESA` , we see that `Tri Scheme` produces tighter bounds at the expense of marginal increase in CPU time. Our results indicate that `Tri Scheme` is indeed a realistic approach to be used inside proximity algorithms to reduce the distance calls without incurring much overhead.



(a) [SF] Bounds $135K$ Edges SF Graph. Relative Error is 0 for `SPLUB` with `ADM` and for `Tri Scheme` much lesser than `LAESA` , especially UB.



(b) [SF] `Tri Scheme` Bounds Varing #of Edges. With scale `Tri Scheme` LB-UB gap reduces drastically (3.3× across 2k and 134k)

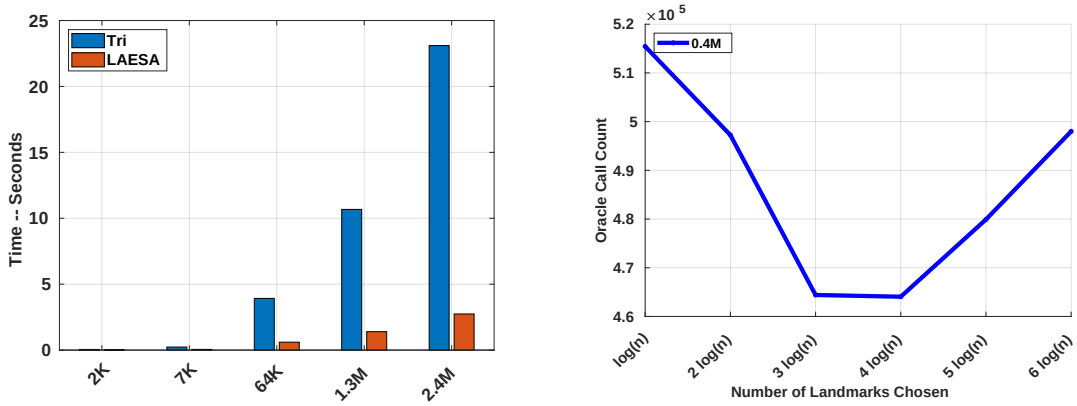


(c) [SF] Time: `ADM` is not scalable. `SPLUB` produces same bound with 50% of time. `Tri Scheme` improves time by 99% (highest: 23s)

Figure 3.3: Bounds Comparison for UB and LB (a) Relative Error is 0 for `SPLUB` with `ADM` and for `Tri Scheme` much lesser than `LAESA`

3.5.2 Tightness of Bounds and Running Time

Figure 3.3a demonstrates `Exact SP` produces exact same bounds (upper and lower) as `ADM` and the error bar is virtually collapsed. Bounds produced by `Tri Scheme` (Figure 3.3b) are looser than `ADM` , however, however much tighter compared to `LAESA` . From these figures it is evident that `Tri Scheme` is a practical yet viable solution for our studied problems. Figure 3.3c and Figure 3.4a provide additional insights: First and foremost, `ADM` , even though produces the best bounds, is not scalable due to its cubic running time. In fact, neither `Exact SP` nor `ADM` is suitable for larger graphs in terms of CPU overhead. Figure 3.4a shows even though `LAESA` is the fastest among all the algorithms, the relative error is much higher.



(a) [SF] fast but loose bounds of `LAESA` (consequence increased distance calls)

(b) [SF] ideal # of landmarks selection problem in `LAESA`

Figure 3.4: Examination of limitations of `LAESA`

3.5.2.1 Limitation of `LAESA`

We discuss the limitations of `LAESA` in section 3.4.2 for choosing the right value of the number of landmarks and its overall effect on the number of distance calls. This

argument has been experimentally corroborated in figure 3.4b. We experimentally observe that the optimal value of number of landmarks for the *chosen dataset* is $4 \times \log(n)$. But this varies largely across datasets and proximity algorithms and there is no obvious ways to determine this parameter.

3.5.3 `Tri Scheme` for Distance Counts

In earlier section we have established impracticality of `ADM` and `Exact SP` for large graphs. Thus we turn our attention to the practical approach, `Tri Scheme` and study how it saves distance calls inside various proximity algorithms wrt `LAESA` .

These experiments confirm our previous findings. Proximity algorithms augmented with ***`Tri Scheme`*** *shows significant improvement in saving the distance calls when compared with* `LAESA` *. 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` .

UrbanGB Dataset [Oracle Call Count]								
Prims Algorithm [$k = \log_2(n)$]								
# of Edges	LandMK `Tri Scheme`	Without Plug	Prime Calls	`Tri Scheme`	LAESA	**Savings (%)**	Best k (LAESA)	Best k LAESA
2016	6	2016	363	999	1097	**8.93**	$\log(n)$	6
8128	7	8128	868	2980	3343	**10.86**	$\log(n)$	7
32640	8	32640	2012	10017	13011	**23.01**	$2\times\log(n)$	16
130816	9	130816	4563	30045	43259	**30.55**	$3\times\log(n)$	27
499500	10	499500	9945	82630	142572	**42.04**	$3\times\log(n)$	30
1999000	11	1999000	21934	260191	529904	**50.90**	$2\times\log(n)$	22
7998000	12	7998000	47922	774466	2096444	**63.06**	$2\times\log(n)$	24

Table 3.2: # of expensive Oracle Calls comparison by `Prim's` Algorithm with `Tri Scheme` and `LAESA` along with parameters

UrbanGB Dataset [Oracle Call Count]								
Prims Algorithm [$k = \log_2(n)$]								
# of Edges	LandMK `Tri Scheme`	Without Plug	Prime Calls	`Tri Scheme`	LAESA	**Savings (%)**	Best k (LAESA)	Best k LAESA
2016	6	2016	363	1230	1254	**1.91**	$\log(n)$	6
8128	7	8128	868	3670	3813	**3.75**	$\log(n)$	7
32640	8	32640	2012	12081	13212	**8.56**	$\log(n)$	8
130816	9	130816	4563	40547	48317	**16.08**	$2\times\log(n)$	18
499500	10	499500	9945	138184	178852	**22.74**	$4\times\log(n)$	40
1999000	11	1999000	21934	369324	523728	**29.48**	$3\times\log(n)$	33
7998000	12	7998000	47922	1312757	2123068	**38.17**	$4\times\log(n)$	48
31996000	13	31996000	103909	4213538	7334143	**42.55**	$4\times\log(n)$	48

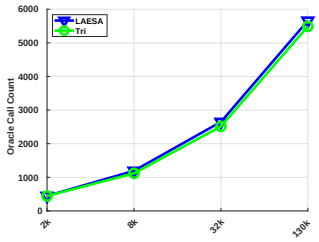
Table 3.3: # of expensive Oracle Calls comparison by `Prim's` Algorithm with `Tri Scheme` and `LAESA` along with parameters

3.5.3.1 Evaluation of MST Algorithms

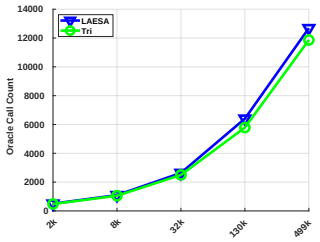
We compare the classical `Prim's` and `Kruskal's` algorithms for the MST problem with their augmented versions through `Tri Scheme` varying number of objects. We present the save-ups.

Table 3.2 and Table 3.3 presents comprehensive results that demonstrate that the original algorithms need to exhaustively calculate all the distances to build the graph in order to finish the minimum spanning tree. Save-ups is increased with increasing size of the datasets, shown in bold as percentage of distance calls saved in `Tri Scheme` compared to `LAESA` , demonstrating the efficacy of `Tri Scheme` .

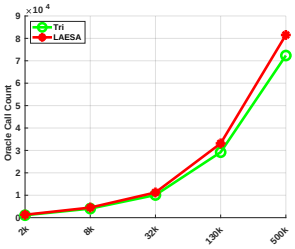
Figures 3.5a and Figure 3.5b represent the distance save up. It is interesting to note that, proximity algorithms, in general, are sensitive to the total number of pairwise distances. The efficacy of `Tri Scheme` in saving the distance calls is evident in both the figures.



(a) [Flicker1M]`Kruskal's` algorithm distance save varying dataset size



(b) [UrbanGB] `Kruskal's` algorithm showing distance save varying dataset size.



(c) [UrbanGB] `KNNrp` algorithm showing distance save. `Tri Scheme` bounds matching with `SPLUB` (theory exact bound)

Figure 3.5: Number of Expensive Oracle Calls for completion of Algorithms `Kruskal's` algorithm and `KNNrp`

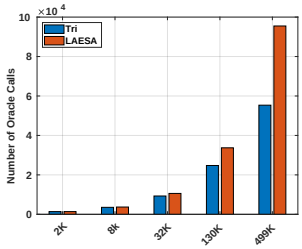
3.5.3.2 Evaluation of Clustering

We compare the two l-medoid ($l = 10$) algorithms `PAM` and `CLARANS` with their augmented versions with `Tri Scheme` . Overall, algorithms augmented with `Tri Scheme` use on average one third the number of distance calculations. Here we vary the size of the graph while conducting the experiments.

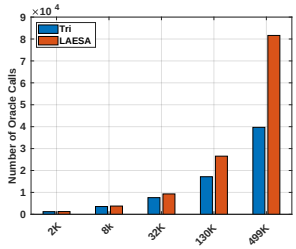
Figure 3.6a, Figure 3.6b, Figure 3.6c, Figure 3.7a and, Figure 3.7b clearly show the results. For both algorithms, as the number of objects grows, the number of distance calculations also increases. We observe the maximum saving up to 36% for SF and a save up of 44% for the UrbanGB datasets. We also note that the perceived large running time of `PAM` is due to its inherent nature, and not due to `Tri Scheme` .

3.5.3.3 Evaluation of k-NNG

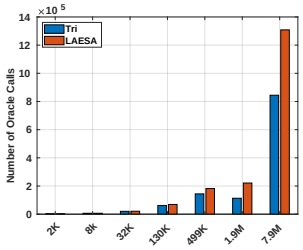
The objective of these set of experiments is to compare the vanilla `KNNrp` [27] ($k = 5$) with the `KNNrp` augmented by the algorithmic scheme, `Tri Scheme` developed



(a) SF: PAM Algorithm Varying Size. With increasing dataset size, the distance save % increases for `Tri Scheme`



(b) UrbanGB: PAM Algorithm Varying Size. With increasing dataset size, the distance save % increases for `Tri Scheme`



(c) SF: `CLARANS` Algorithm Varying Size. Increased Scalability to larger settings ($7.9M$) with `CLARANS` without compromising saves.

Figure 3.6: Number of Expensive Oracle Calls For Algorithms `PAM` and `CLARANS`

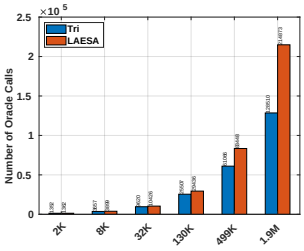
in this work in saving distance calls. Figure 3.5c describes the number of distance calls made by the algorithm. The findings are similar to other proximity algorihtms.

3.5.4 `Tri Scheme` for Running Time

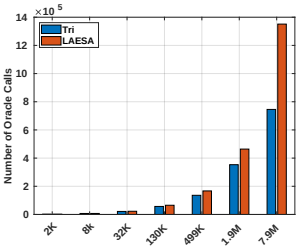
We present the the end-to-end completion of the proximity algorithms, when augmented by `Tri Scheme` and `LAESA` by varying the cost of distance computation. *We observe that the overhead induced by our algorithms, are nominal when compared with the results from* `LAESA` *. However, when induced with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

3.5.4.1 Evaluation of MST Algorithms - Time

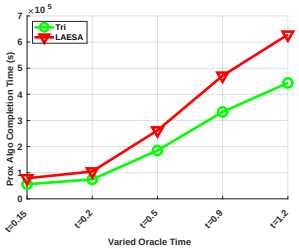
Here we present the timing details for the `Prim's` algorithm, for `Tri Scheme` and `LAESA` for completing the algorithms. Figure 3.7c give an idea about the overall time taken for the proximity algorithm to complete when different distance oracle time is considered. We vary the time taken for a distance call to be as $0.15s, 0.2s, 0.5s, 0.9sand1.2s$



(a) Flicker1M: PAM Algorithm Varying Size. 40% save up in largest setting.

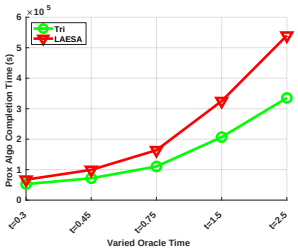


(b) UrbanGB: `CLARANS` Algorithm Varying Size

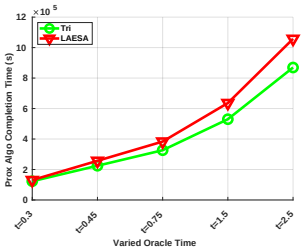


(c) [SF $1.99M$][Prox Completion Time] 30% save for 1.2s oracle on completion of `Prim's`

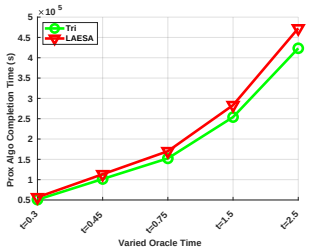
Figure 3.7: (a, b) Number of Expensive Oracle Calls For Algorithms `PAM` and `CLARANS`) (c) Time for completion of Proximity algorithms for `PAM`



(a) [Flicker1M $1.99M$][Prox Time] 37% save for 0.3s oracle on completion of `PAM`



(b) [UrbanGB $1.99M$][Prox Time] 40% save for 1.2s oracle on completion of `CLARANS`



(c) [Urban $1.99M$][Prox Time] 10% save for 2.5s oracle on completion of `KNNrp`

Figure 3.8: Actual Proximity Algorithm Completion Time for different Proximity Algorithms in the light of varying cost of distance oracles

As one can notice that the time for completing the proximity algorithm for `LAESA` in every case is higher than `Tri Scheme` . The difference is in the order of 30% even if the distance computations on an average consume 1.2s.

3.5.4.2 Evaluation of Clustering - Time

We study `PAM` and `CLARANS` ($l = 10$) the overall completion times in conjunction with `Tri Scheme` and `LAESA` .

Figures 3.8a and 3.8b show the results by varying oracle cost as 0.3, 0.45, 0.75, 1.5, and 2.5 seconds. The overall time saveup is 37% when `Tri Scheme` is used for augmentation and outperforms `LAESA` significantly.

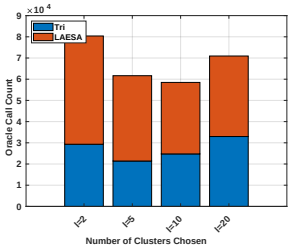
3.5.4.3 Evaluation of k-NNG - Time

Finally, we take the k-NNG ($k = 5$) problem using Urban dataset with $1.99M$ settings.

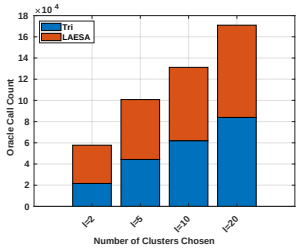
These results indicate that by leveraging triangle inequality `Tri Scheme` outperforms `LAESA` .

3.5.5 Varying Proximity Parameters

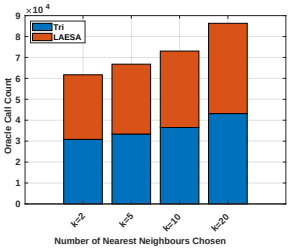
Proximity algorithms are sensitive to its parameters of choice. Clustering algorithms like `PAM` and `CLARANS` are required to accept the number clusters (l) as a part of their inputs. Similarly, kNNG needs k, the number of neighbours k.



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

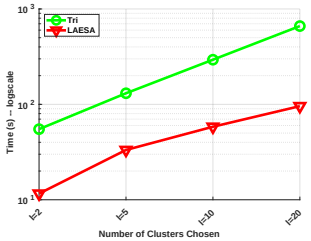


(b) [SF $130K$][Oracle Count] Stability of `Tri Scheme` to variance of l in `CLARANS`

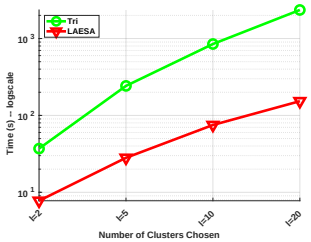


(c) [SF $130K$][Oracle Count] Stability of `Tri Scheme` to variance of l in `KNNrp`

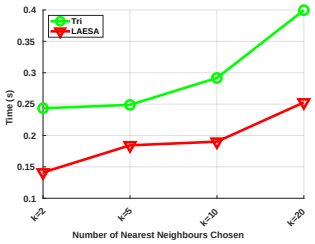
Figure 3.9: Effects of varying l (k) on the number of distance calls for `PAM` & `CLARANS` (`KNNrp`)



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



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



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

Figure 3.10: Effects of varying l (k) on local CPU computation (disconnecting the problems of distance compute and CPU compute and reducing distance compute ($\downarrow$) at the expense of CPU compute ($\uparrow$)) for `PAM` & `CLARANS` (`KNNrp`)

3.5.5.1 Clustering (varying l) - Count, CPU overhead

Here we vary l and compare the number of distance calls of `Tri Scheme` against `LAESA` . Figure 3.9a presents the results from the `PAM` algorithm. As the number of objects is fixed, increasing the number of clusters will results in more local minima for the `PAM` algorithm which in turn makes the algorithm converge faster. Figure 3.9b presents the results from the `CLARANS` algorithm. As l increases, CLARANS on the other hand, does not exhibit this phenomena and as the number of clusters increases the number of distance calls also increases.

Figure 3.10a and Figure 3.10b show that the CPU overhead for `PAM` and `CLARANS` algorithms respectively. As expected, when l increases, we see increase in the CPU overhead in response to the number of additional upper and lower bound comparisons.

3.5.5.2 k-NNG (varying k)- Count and CPU overhead

We present the results by varying k for `KNNrp` algorithm here. Figure 3.9c 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 3.10c shows the same effect in CPU overhead, as described in section 3.5.5.1.

3.6 Related Work

Existing works could broadly be divided in two ways - ones that deal with metric space directly and those that transform metric space into vector space [22], [23], [24]. These latter techniques do not produce exact answers while used inside proximity problems and is not our focus here.

3.6.1 Pivot based indexes

Pivot based indexed data structures select a set of nodes (pivots) in order to divide the space into smaller sub-spaces, essentially grouping similar objects together. They are described in depth next.

Tree based pivots. BKT [25], a pivot based data structure designed for similarity search which recursively builds a tree based on the distance to other objects. FQT, FHQT [26], and Fixed Queries Array (FQA)[30] are follow up works that offer improvement to this. Any object stores a sorted list of (integer) distances from the object to the k pivots in an array. Yianilos [51] proposed Vantage Point Tree(VPT) index structure to solve the problem of nearest neighbor queries in general metric spaces.

However, these works are designed for specifically for discrete-valued distance functions and/or do not generalize to provide effective bounds, such as ours.

Landmark based pivots. A different trend of algorithms based on pivots, stores partial information of nodes in an array form to answer nearest neighbour queries in metric space. The representative algorithms in this type are AESA and LAESA [34, 35] which take the idea of pivot based methods to the extreme. In AESA,

all pairwise distance are precomputed and stored in matrix. During query, a random object from the database is selected to find the distance between the query object and the selected object to establish the boundary of the search space eliminating a large part of the search space. In LAESA, an extension to AESA, a set of objects known as Base Prototype/Landmark/Pivot are chosen and all the pairwise distances between those are stored. This matrix is later used during the query time.

There are two main issues with the landmark based solutions. First, these works are not suitable for our update operations. Second, heuristics adopted for finding the landmarks requires to know the number of landmarks, which often is dependent of the underlying data distribution and thus is hard to generalize.

3.6.2 Voronoi-*like* indexes

Voronoi diagrams, commonly used in proximity queries in vector spaces, have inspired data structures in metric spaces, namely GNAT [29] and M-tree [32]. GNAT [29] introduces an indexing structure for nearest neighbour queries in large metric spaces. The tree construction starts with a selection of k objects in random from the dataset and associate remaining objects (partition) to the k objects selected based each objects proximity to one of them. M-Tree [32] is a balanced tree indexing structure for metric space similarity search and *k-NN* problems. M-Tree, which works by partitioning the space, is built in a bottom up manner, and has fixed number of objects in each node. When each level is filled, the tree expands upwards, relaying the partition information upwards and also maintaining a balanced structure. *These body of works have a fundamental limitation that make them inapplicable to our problem: These are specific to nearest neighbor search and does not generalize to proximity problems in metric space.*

3.6.3 Matrix Based Indexes

We are aware of the work by Sasha and Wang [6] that proposes `ADM` to estimate tightest upper and lower bounds of distances that are then demonstrated to save up different querying cost. The bounds are stored in a quadratic data structure $\mathcal{O}(n^2)$ and could be used for proximity calculations. *Even though they produce exact bounds on missing edge values, the computational cost in bounding the distances for* `ADM` *is in the* $\mathcal{O}(n^3)$*, and thus is not practical to be used inside proximity algorithms. Additionally, this method does not take advantage of the sparsity of the underlying graph.*

Comment 3 *3.6.4 Transforming Metric Space into Vector Space:*

The second kind *transforms the metric space into a vector space by transforming each object to a finite-dimensional vector. The distance function is then defined based on the distance between these vectors. Some of the most useful distance functions are Euclidean distance, Manhattan distance, or* L_∞*. Popular solutions in this space include* kd-tree *[22],*R-tree *[23], or more recent* X-tree *[24]. These approaches usually use the coordinate information of the objects.* However, there are two fundamental issues to use these algorithms for solving general metric space problems. First, these class of solutions, when applied in proximity problems, do not produce identical results, with the original algorithm. Second, we do not investigate this genre of work, as they do not directly optimize distance calls.

3.7 Conclusion

In this paper, we revisit a suite of popular proximity problems that repeatedly perform distance computations by making calls to a third party distance oracle (such as, google maps) to compare a pair of distances during their execution. We present

Direct Feasibility Test that studies how distance comparisons between two different pairs of objects could be modeled as a system of linear inequalities that assists in saving distance computations. We furthermore present an alternative formalism with the goal of computing distance bounds and present a suite of graph based algorithms that are computationally "light weight". We compare our designed solutions conceptually and empirically wrt a broad range of existing works through comprehensive experimentation. As an immediate extension, we are exploring whether it is beneficial to update the unknown distance bounds after every update in the underlying graph or better to defer it and process a set of updates as a batch. If the latter is more appropriate, we shall investigate how to determine batch size inside different proximity algorithms that suitably trade-off between running time and tightness of produced bounds.

Reconstructing a high dimensional unknown signal, using lower dimensional observations is a challenging problem, known as *signal reconstruction problem* (SRP), with diverse applications including network traffic engineering, medical image reconstruction, and astronomy. Recently the database community has shown significant advancements in solving the SRP problem efficiently, effectively, and in scale by leveraging database techniques such as similarity joins. In this demo, we demonstrate Orca-SRthat highlights the benefits of signal reconstruction in scale by demonstrating real-time network traffic flow analysis on large networks that were not possible before. Orca-SRis a web application that enables a user to generate network flow and load the network for interactive analysis of the impact of different traffic patterns on signal reconstruction.

Algorithm 11 `Tri Scheme`

Input : graph $\mathcal{G}$ = (O,E), unknown edge (o_i, o_j)

Output : $LB^{d(o_i,o_j)}$, $UB^{d(o_i,o_j)}$

$adj_i = AdjacencyList(o_i)$

$adj_j = AdjacencyList(o_j)$

$lb = 0$

$ub = 1$

while $i \leq len(adj_i) and j \leq len(adj_j)$ **do**

    if $adj_i[i] == adj_j[j]$ **then**

        $lb = max(lb, |E[o_i, adj_i[i]] - E[o_j, adj_j[j]]|)$

        $ub = min(lb, E[o_i, adj_i[i]] + E[o_j, adj_j[j]])$

    else

        if $adj_i[i] > adj_j[j]$ **then**

            $j = j + 1$

        else

            $i = i + 1$

        end if

    end if

end while

$LB^{d(o_i,o_j)} = lb$

$UB^{d(o_i,o_j)} = ub$

return $LB^{d(o_i,o_j)}, UB^{d(o_i,o_j)}$

CHAPTER 4

Orca-SR: A Real-Time Traffic Engineering Framework leveraging Similarity Joins

4.1 Introduction

Developing scalable algorithms for processing and analyzing massive data systems have been the mainstays of database research for many decades. Database optimization techniques, in particular, have been applied beyond the realm of the database community to yield scalable algorithms for solving problems in the areas of astronomy, computer networks, machine learning, etc.

Signal Reconstruction Problem (SRP) is a challenging problem with diverse applications where the objective is to reconstruct a high dimensional unknown signal using lower dimensional observations. Prominent database techniques such as similarity joins and selectivity estimation for set similarity queries, can yield significant performance improvements on SRP by optimizing for the computational bottlenecks. Several algorithms were proposed in [8] and its extended version [9] for a scalable solution for SRP problem. One popular application of SRP is in Computer Networks, where the observation of a system is limited[1] to the edge level traffic due to infrastructural limitations. We demonstrate Orca-SR, a system for traffic flow estimation, which can estimate source-destination traffic flows in *real-time*.

Signal Reconstruction Problem (SRP): The aim of SRP is to solve a linear system of the form $AX = b$, where $X \in \mathbb{R}^m$ is a high-dimensional unknown *signal*, $b \in \mathbb{R}^n$ $(n \ll m)$ is a low-dimensional projection of X that can be observed in practice

[1]SRP has numerous applications in diverse domains that are elaborated in [52], the report for the SIGMOD Research Highlight 2019.

and A is a $n \times m$ matrix that captures the linear relationship between X and b.

Traffic Matrix Computation as SRP: Consider an IP network with r ingress/egress points where network traffic may enter or leave with any of the $m = r^2$ source-destination flows (SD flow). Knowing the traffic flow between any source-destination pair is crucial in traffic engineering. Considering the infrastructural limitations, one usually cannot directly measure SD flows. Protocols such as SNMP, collect the link-level flow information, however, falls short in estimating the end-to-end traffic flow. Hence, SRP seeks to estimate the flow between every SD pairs from link-level flow information and routing policy available through routing protocols like BGP. With the growing network size (a few billion SD pairs), prior solutions [53, 54] for SRP simply do not scale. Hence, we proposed methods that support *large scale signal reconstruction* [8, 9]. ORCA-SRleverages these methods and demonstrates a real-time system for interactive network traffic analysis for helping the network analysts to visualize a selected network along with the reconstructed traffic flows.

Problem Formulation: Computer networks are often represented as graph where nodes corresponds to endpoints while the edges correspond to network links. The interconnection between different SD pairs is defined by a routing policy and is commonly represented as a routing matrix, $\mathcal{A}$. It is a binary matrix with edges as rows and SD pairs as columns where for the edges involved in the route between SD pairs is set to 1. The observable properties of the network include the flow at an edge, known as edge traffic vector, b and $\mathcal{A}$. The traffic reconstruction problem is defined as resolving individual SD traffic, $\mathcal{X}$ - represented as a vector, when routing matrix $\mathcal{A}$ and edge traffic vector b are available. This linear system of equations is represented as,

$$\mathcal{A} \cdot \mathcal{X} = b$$

where $\mathcal{A} \in \mathbb{Z}_2^{n \times m}$ is a sparse binary matrix, where n is number of edges in the network and m the number of SD pairs($n \ll m$).

SRP could not be solved using system of linear equation techniques as the number of variables m is larger than the number of equations, n, rendering the system underdetermined with no unique solution. Hence, an additional criterion is needed to choose a distinct $\mathcal{X}$ from the infinitely many possible solutions. A common secondary criterion used in signal reconstruction is the use of a prior point, $\mathcal{X}'$, provided by experts [54]. The final traffic flow values are obtained by taking the point in the solution space that is closest to the given prior. In [8], we provide a closed form formula

$$\mathcal{X} = \mathcal{X}' - \mathcal{A}^T(\mathcal{A}\mathcal{A}^T)^{-1}(\mathcal{A}\mathcal{X}' - b) \tag{4.1}$$

As evident from Equation 4.1, $\mathcal{A}\mathcal{A}^T$, is the computational bottleneck for the algorithm which runs in $\mathcal{O}(n^2m)$. It turns out that each diagonal element $(\mathcal{A}\mathcal{A}^T)[i][i]$ is the upper bound for values in its i^{th} row and i^{th} column of $\mathcal{A}\mathcal{A}^T$. Using techniques from set similarity joins [55] and selectivity estimation for similarity queries [56], we can improve the performance of the algorithm. We now briefly describe the three key ideas. Please refer to [8, 9] for comprehensive details.

- *Thresholding*: We use the diagonal values as the threshold for the values on the same row/column and prune the ones that are below a threshold τ.
- *Conversion to Set Operations*: We transform the problem of computing the inner product between the rows $\mathcal{A}[i]$ and $\mathcal{A}[j]$ (for computing $\mathcal{A}\mathcal{A}^T[i,j]$) into a set similarity instance.
- *Similarity Joins*: We design a hybrid algorithm combining the threshold-based and sketch-based similarity estimation to reduce the complexity of computing a cell in $\mathcal{A}\mathcal{A}^T$ from $O(m)$ to $O(\log m)$.

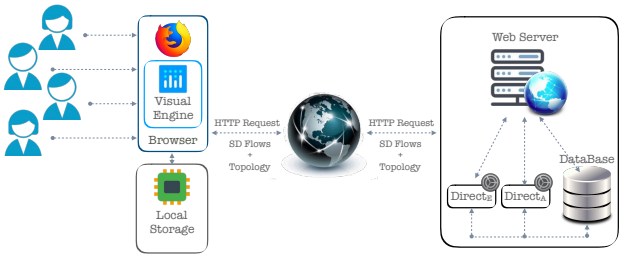


Figure 4.1: Architecture of Orca-SR

The exact version of DIRECT algorithm based on Eqn 4.1 is referred to as DIRECT-E and the approximate version leveraging database optimizations proposed above forms DIRECT-A.

4.2 Orca-SR

In this section we describe ORCA-SR, its architecture and implementation details. We describe various technical challenges and user interaction with ORCA-SR.

Comment 4 *Both the computational efficiency and effects of approximation by choosing a* user specified threshold *is presented to a user through the $\mathcal{A}\mathcal{A}^T$ matrix over which a user can get direct feedback on the choice of threshold over the network infrastructure. Often with large networks, complete information will burden a network engineer and subsetting the flows will be essential. Thus to simplify,* ORCA-SR*, provides an interactive sort and filter feature where a user can focus on parts of the network to analyze the details of the flows.*

4.2.1 Architecture

ORCA-SRis a web application where users submit a query for generating the network, along with the parameters for generating the traffic and analyzing it. Architecture of ORCA-SRis shown in Figure 4.1. Users submit their requests through a web browser, to ORCA-SRWeb Server. These include network type, edge level

traffic distribution, a prior probability distribution for traffic and, a threshold. The Orca-SRweb Server generates the routing matrix and traffic vector and subsequently applies the `SRP` algorithm (described in [8]), either Direct-e or Direct-a based on users requirement to generate exact or approximate answers. Once the algorithm finds SD pair traffic flows, the server sends the results back to the client's visual component which processes and renders the flows. Local storage at client browser caches the SD pairs flow information to answer interactive queries by users. With Orca-SR, a user can learn more about the applied optimizations by choosing a small subset of the network and subsequently focusing on the sub-network and, changing its input parameters. One can also fine-tune the parameters interactively through our visual interface.

Our application uses the exact algorithm Direct-e and the approximated one Direct-a for scalability, both implemented in C++ . For orchestrating the Web App, we use Shiny from R[2]. For visual rendering at client browser we use Plotly[3] which internally uses the visualization library D3.js.

4.2.2 Technical challenges

In this subsection, we discuss major challenges encountered, their resolutions, and the reasoning for their choices.

Scalability through Parallelization: The first challenge entails from the large scale of the network that we simulate. A general assumption in [8, 9] is the availability of $\mathcal{A}$, a matrix at the scale of up to hundreds of millions SD pairs. Materializing $\mathcal{A}$ at this point is beyond the capabilities of a Web application. Even the sparse representation of $\mathcal{A}$ is in the order of a few 100 GBs. Hence, we designed optimized binary

[2]https://www.rstudio.com/products/shiny/shiny-server/

[3]https://plot.ly/

matrix multiplication libraries that operate on the sparse matrix for the implementation of DIRECT-E in C++, which utilizes the underlying processor parallelism. These modules take advantage of the underlying parallelism in processors. And, each of the multiplication will have a depth of the length of path for each source destination pair in $\mathcal{A}\mathcal{A}^T$ multiplication. If the mean length of the path in an un-directed path will be $l = \frac{1}{n\times(n-1)} \sum_{i\neq j} d_{ij}$ where n is the number of distance nodes in the network and this approach provides for a space and performance advantages to algorithms proposed in [8, 9].

During the simulation process, a sequential path generator is the bottleneck for the application. Hence, we designed a parallelized version of the route generator considering the performance issues. Our code can be found at `https://github.com/jeesaugustine/orca_demo`.

Comment 5 ***Scalability*** *: The first challenge entails from the large scale of the network that we simulate. A general assumption in [8, 9] is the availability of $\mathcal{A}$, a matrix at the scale of up to hundreds of millions SD pairs. However, the routing matrix, $\mathcal{A}$ in practice could range in the order of a few hundred million rows and columns, proportionally, in the order of a few billion. Materializing $\mathcal{A}$ at this point is beyond the capabilities of a Web application. Even the sparse representation of $\mathcal{A}$ is in the order of a few 100 GBs. Hence, we designed optimized binary matrix multiplication libraries that operate on sparse matrix for the implementation of* DIRECT-E *in C++. These modules take advantage of the underlying parallelism in processors. If the mean length of the path in an un-directed path will be $l = \frac{1}{n\times(n-1)} \sum_{i\neq j} d_{ij}$ where n is the number of distance nodes in the network and This approach provides for a space and performance advantages to algorithms proposed in [8, 9].*

Routing: Another major challenge is obtaining the routing matrix $\mathcal{A}$ for large networks. Algorithms proposed in [8, 9] require a user to furnish $\mathcal{A}$, b and $\mathcal{X}'$ for com-

puting DIRECT-E and DIRECT-A. Even with parallelization, assigning the shortest path to billions of SD pairs is computationally expensive. We use a clustering-based approach in addition to the shortest paths to generate and thus $\mathcal{A}$. This is analogous to a hierarchical routing approach [57] practiced in the networking community. Please refer to [8] for further details.

Multi-User Interaction: Algorithms described in [8] were designed for solving a single instance of SRP. To facilitate multiple users interacting with the system seamlessly, we added a string of optimizations such as caching that avoids redundant computation of expensive results and sandboxing that separates the requests of different users.

4.2.3 User interfaces

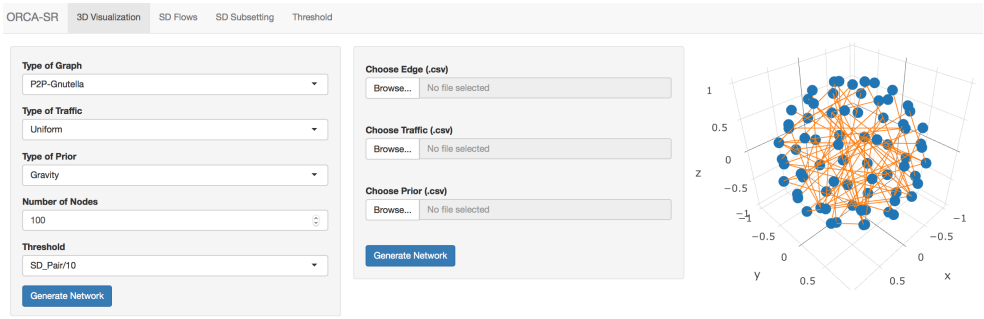


Figure 4.2: Orca-SR Interactive Network Simulation and Visualization interface

ORCA-SRhas extensive functionalities and has multiple interfaces. Due to the space limitations, we limit ourselves to the key functionalities of ORCA-SRin the write up.

Network Graph Simulation - Figure 4.2 shows the primary input interface. We provide support for various forms of input graphs (i) real networks (from SNAP

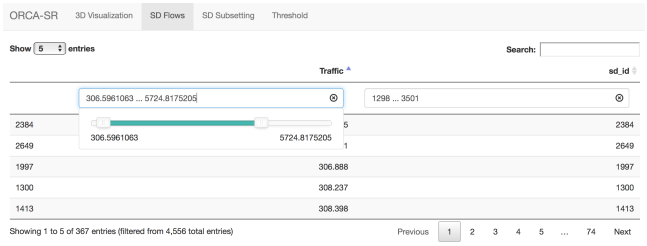


Figure 4.3: Interactive selection of top-k SD Flows

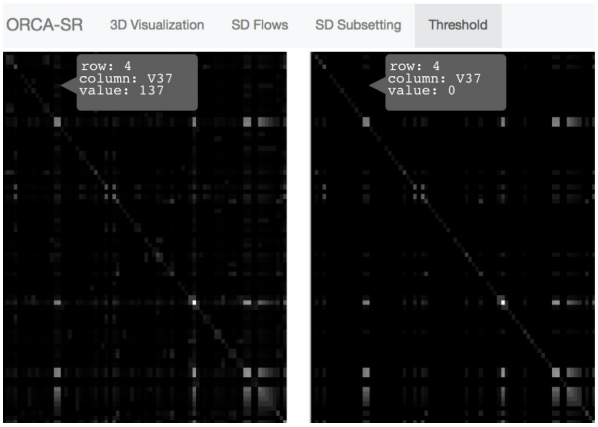


Figure 4.4: Visualizing $\mathcal{A}\mathcal{A}^T$ without and with thresholding

repository) (ii) a random graph generator by Erdős–Rényi random graph generation[4] (based on user given parameters) and, (iii) a user provided network. Users may choose different approaches for modeling *prior* probability distribution for traffic and various traffic generation methods according to the choice and fitting to the application domain requirements. A user will also get a choice of different thresholds for the DIRECT-a algorithm as fractions of the number of SD pairs in the network. On selecting a node, ORCA-SRwill process the request and display the traffic flowing from the selected node to all other nodes in the network. A user may download all the artifacts of the network and flow (as *.csv* files) for further investigation.

SD Flow Analysis - The chosen parameter values are used to calculate the SD traffic for each SD pair. A user can interact with the flow values to selectively view

[4] `https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model`

parts of the output, focusing on flows the user is most interested in analyzing. The interface offers subsetting capabilities, which allows a user to dynamically choose a traffic volume and sort the flows (ascending or descending) to obtain top-k (one can also vary k) flows in an order preferential to the user using sliders. Users can search for a particular SD pairs or a traffic volume to locate the flow. In Figure 4.3, one can observe the top-5 SD flows filtered by the flow volumes (306-5725) and SD Pair ids (1298-3501) sorted in ascending order.

Source Destination Subsetting - We allow a user to closely analyze network flow over a small subset of the generated network. The flow values that we obtain from the algorithm are represented as a scatter plot for analysis. Once selected, a subgraph (of selected flows) is obtained, and users can interactively visualize the effect of a change in parameters (prior, path, change in route etc.) on flow values.

Visualizing $\mathcal{A}\mathcal{A}^T$ - Database optimization for SRP is based on the observation that, bulk of the values of $\mathcal{A}\mathcal{A}^T$ is concentrated in a small number of values(along the major diagonal). We allow the user to interactively view the effect of using a threshold on $\mathcal{A}$ and $\mathcal{A}\mathcal{A}^T$ for its processing. The heat map on the left panel of Figure 4.4 shows the sparseness of the $\mathcal{A}\mathcal{A}^T$ matrix as well as the domination of the diagonal values over the corresponding row and column. Far fewer non zero values on right panel shows effect of thresholding in computation of $\mathcal{A}\mathcal{A}^T$. We highlighted the same cell (row-4, column-37) in both panels to show the optimization through thresholding (left calculated as 137 and right was never calculated, as 137 is lesser than the threshold, 455). One can clearly notice the pixel density(as a gradient of value associated with it) of the cell to turning from a light shade(on left) of white to complete dark (on right).

4.3 Demonstration plan

This section details the implementation details of ORCA-SR, it's demonstration plan and comparison portfolios.

4.3.1 Demonstration scenarios

Datasets. We experimented ORCA-SRwith all datasets mentioned in the works by Asudeh *et al.* [9] and reproduced the results. For this demonstration, we highlight three different datasets – synthetic Erdős-Rényi graph N_1 and two real network datasets $p2p_1$ and $p2p_2$ from SNAP[5] repository.

Comment 6 Synthetic Erdős-Rényi graphs*: The synthetic dataset* N_1*, of* 274 *Nodes with* 281 *Edges and* 827 *SD Pairs.* **Real Dataset 1***:* $p2p_1$ *dataset is a* subnetwork *of snapshot of* **Gnutella Peer-to-Peer** *network(August* 2002*) with* 10876 *Nodes and* 39994 *Edges published in the SNAP dataset(https://snap.stanford.edu/).* $p2p_1$ *has* 369 *Nodes with* 4549 *Edges and* 136*K SD Pairs.* **Real Dataset 2***:* $p2p_2$ *dataset is also derived from the* Gnutella *dataset.* $p2p_2$ *is a medium sized network with* 612 *Nodes with* 3486 *Edges and* 373*K SD Pairs.*

User Interaction Model. On load, the web app introduces a user to network, traffic and threshold selection page. A user can choose a dataset (one of N_1, $p2p_1$ and $p2p_2$), traffic generation model (*Pareto [58]*, *normal*, *exponential*), choice of traffic prior (*preferred: gravity [54]*, *normal*, *random*). The prior is a critical measure as our final solution is a point in the solution space that is closer to the chosen prior. As the final input, a user may provide the choice of the *threshold*(*preferred: fractions of 10 on # of Nodes*) value, the one which determines the accuracy of the results and computational time for the results. Besides, users can provide a new dataset to the system as *.csv* file.

[5]https://snap.stanford.edu/

Comment 7 *Once inputs are chosen,* ORCA-SR*simulates the network and, SD traffic is generated. Interactive visualization of the network is presented to the user, where one can review the topology of the network. Users can sort and filter the traffic based on the flow identifier or the traffic volume. Descriptions and details of various demonstration use cases are as follows.*

Active Real-Time SRP. While prior state-of-the-art methods computed the traffic matrix in a couple of hours, ORCA-SRcompletes the SD pairs traffic estimation in a few minutes, demonstrating orders of magnitude improvement. We demonstrate the performance improvements based on large real-world networks. Traffic volume generated by SD pairs change instantaneously and previously, network administrator had to wait for a couple of hours for re-estimation with changes in b. With ORCA-SRand its optimizations, we demonstrate to track these changes and reflect the changes in SD pairs traffic over the network in real-time. Users can select the frequency of change in traffic from the drop-down menu in the *visualization* tab and the system automatically updates the traffic instantaneously.

Dynamic Network Adaptation. Real networks evolve dynamically and a network administrator would need to get a macroscopic view of the network adapting to these changes for planning resources. Often changes in the network requires a complete recomputation in previous solution frameworks and were impractical for real-time system monitoring. ORCA-SRcan dynamically capture these network changes and adapt without recomputing the solution through selective reuse of signature matrix [9]. Users can choose to update the network from the *SD Flows* tab with frequency of addition(deletion) or the number of new(deleted) nodes. ORCA-SRdynamically adapts and adjusts to these network changes in real-time.

Top-k Flows Detection and Real-Time Network Sub-setting: Network administers are also interested in microscopic view of the network for resource allocation including the top-k flows. ORCA-SRenables an administrator to identify the top flows [9] and further facilitates a user to focus on a subset of the network with top-k flows for analysis. In this demonstration, once a sub-network is chosen the system automatically adapts to the sub-network, allowing a user to update traffic flows in the sub-network.

REFERENCES

[1] H. Rahman, S. B. Roy, and G. Das, "A probabilistic framework for estimating pairwise distances through crowdsourcing," in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß, Eds. OpenProceedings.org, 2017, pp. 258–269. [Online]. Available: https://doi.org/10.5441/002/edbt.2017.24

[2] J. Yi, R. Jin, A. K. Jain, S. Jain, and T. Yang, "Semi-crowdsourced clustering: Generalizing crowd labeling by robust distance metric learning," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., 2012, pp. 1781–1789.

[3] V. Polychronopoulos, L. de Alfaro, J. Davis, H. Garcia-Molina, and N. Polyzotis, "Human-powered top-k lists," in *Proceedings of the 16th International Workshop on the Web and Databases 2013, WebDB 2013, New York, NY, USA, June 23, 2013*, A. Bonifati and C. Yu, Eds., 2013, pp. 25–30. [Online]. Available: http://webdb2013.lille.inria.fr/Paper%2035.pdf

[4] S. E. Whang, P. Lofgren, and H. Garcia-Molina, "Question selection for crowd entity resolution," *Proc. VLDB Endow.*, vol. 6, no. 6, pp. 349–360, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p349-whang.pdf

[5] R. C. Prim, "Shortest connection networks and some generalizations," *Bell system technical journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[6] J. T.-L. Wang and D. E. Shasha, "Query processing for distance metrics." in *VLDB*, vol. 90, 1990, pp. 602–613.

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[8] A. Asudeh, A. Nazi, J. Augustine, S. Thirumuruganathan, N. Zhang, G. Das, and D. Srivastava, "Leveraging similarity joins for signal reconstruction," *PVLDB*, vol. 11, no. 10, pp. 1276–1288, 2018.

[9] A. Asudeh, J. Augustine, A. Nazi, S. Thirumuruganathan, N. Zhang, G. Das, and D. Srivastava, "Scalable algorithms for signal reconstruction by leveraging similarity joins," *The VLDB Journal, Special Issue on best of VLDB'18*, pp. 1–27, 2019.

[10] ——, "Efficient signal reconstruction for a broad range of applications," *SIGMOD Rec.*, vol. 48, no. 1, pp. 42–49, 2019. [Online]. Available: https://doi.org/10.1145/3371316.3371327

[11] J. Augustine, S. Shetiya, A. Asudeh, S. Thirumuruganathan, A. Nazi, N. Zhang, G. Das, and D. Srivastava, "Orca-sr: A real-time traffic engineering framework leveraging similarity joins," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2977–2980, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2977-augustine.pdf

[12] J. Augustine, S. Shetiya, M. Esfandiari, S. B. Roy, and G. Das, "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 Conference 2021*. ACM, submitted.

[13] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das, "Deep learning models for selectivity estimation of multi-attribute queries," in *Proceedings of the 2020 International Conference on Management of*

Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 1035–1050. [Online]. Available: https://doi.org/10.1145/3318464.3389741

[14] ——, "Multi-attribute selectivity estimation using deep learning," *CoRR*, vol. abs/1903.09999, 2019. [Online]. Available: http://arxiv.org/abs/1903.09999

[15] D. Shasha and T.-L. Wang, "New techniques for best-match retrieval," *ACM Transactions on Information Systems (TOIS)*, vol. 8, no. 2, pp. 140–158, 1990.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[17] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro, "Practical construction of k-nearest neighbor graphs in metric spaces," in *WEA*, vol. 6. Springer, 2006, pp. 85–97.

[18] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[19] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990. [Online]. Available: https://doi.org/10.1002/9780470316801

[20] R. T. Ng and J. Han, "Clarans: A method for clustering objects for spatial data mining," *IEEE transactions on knowledge and data engineering*, vol. 14, no. 5, pp. 1003–1016, 2002.

[21] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM computing surveys (CSUR)*, vol. 33, no. 3, pp. 273–321, 2001.

[22] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[23] A. Guttman, *R-trees: A dynamic index structure for spatial searching.* ACM, 1984, vol. 14, no. 2.

[24] S. Berchtold, D. Keim, and H. Kriegel, "The x-tree: An efficient and robust access method for points and rectangles," in *Proc. 1996 Int. Conf. Very Large Data Bases*, 1996, pp. 28–39.

[25] W. A. Burkhard and R. M. Keller, "Some approaches to best-match file searching," *Communications of the ACM*, vol. 16, no. 4, pp. 230–236, 1973.

[26] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu, "Proximity matching using fixed-queries trees," in *Annual Symposium on Combinatorial Pattern Matching*. Springer, 1994, pp. 198–212.

[27] G. Navarro and R. Baeza-Yates, "Proximal nodes: A model to query document databases by content and structure," *ACM Transactions on Information Systems (TOIS)*, vol. 15, no. 4, pp. 400–435, 1997.

[28] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA*, vol. 93, no. 194, 1993, pp. 311–321.

[29] S. Brin, "Near neighbor search in large metric spaces," in *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, U. Dayal, P. M. D. Gray, and S. Nishio, Eds. Morgan Kaufmann, 1995, pp. 574–584. [Online]. Available: http://www.vldb.org/conf/1995/P574.PDF

[30] E. Chávez, J. L. Marroquín, and G. Navarro, "Fixed queries array: A fast and economical data structure for proximity searching," *Multimedia Tools and Applications*, vol. 14, no. 2, pp. 113–135, 2001.

[31] I. Kalantari and G. McDonald, "A data structure and an algorithm for the nearest point problem," *IEEE Transactions on Software Engineering*, no. 5, pp. 631–634, 1983.

[32] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An e cient access method for similarity search in metric spaces," in *Proceedings of the 23rd VLDB conference, Athens, Greece*. Citeseer, 1997, pp. 426–435.

[33] F. Dehne and H. Noltemeier, "Voronoi trees and clustering problems," *Information Systems*, vol. 12, no. 2, pp. 171–175, 1987.

[34] E. V. Ruiz, "An algorithm for finding nearest neighbours in (approximately) constant average time," *Pattern Recognition Letters*, vol. 4, no. 3, pp. 145–157, 1986.

[35] M. L. Micó, J. Oncina, and E. Vidal, "A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements," *Pattern Recognition Letters*, vol. 15, no. 1, pp. 9–17, 1994.

[36] H. Rahman, S. B. Roy, and G. Das, "A probabilistic framework for estimating pairwise distances through crowdsourcing." in *EDBT*, 2017, pp. 258–269.

[37] A. Beygelzimer, S. M. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006*, ser. ACM International Conference Proceeding Series, W. W. Cohen and A. W. Moore, Eds., vol. 148. ACM, 2006, pp. 97–104. [Online]. Available: https://doi.org/10.1145/1143844.1143857

[38] W. Wu, H. Xiong, and S. Shekhar, Eds., *Clustering and Information Retrieval*. Kluwer, 2003.

[39] Y. Bartal, M. Charikar, and D. Raz, "Approximating min-sum k-clustering in metric spaces," in *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, J. S. Vitter, P. G. Spirakis, and M. Yannakakis, Eds. ACM, 2001, pp. 11–20. [Online]. Available: https://doi.org/10.1145/380752.380754

[40] R. Caruana and A. Niculescu-Mizil, "Data mining in metric space: an empirical analysis of supervised learning performance criteria," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22-25, 2004*, W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, Eds. ACM, 2004, pp. 69–78. [Online]. Available: https://doi.org/10.1145/1014052.1014063

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

[42] V. Klee and G. J. Minty, "How good is the simplex algorithm," *Inequalities*, vol. 3, no. 3, pp. 159–175, 1972.

[43] L. G. Khachiyan, "Rounding of polytopes in the real number model of computation," *Mathematics of Operations Research*, vol. 21, no. 2, pp. 307–320, 1996.

[44] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S. Teng, "On trip planning queries in spatial databases," in *Advances in Spatial and Temporal Databases, 9th International Symposium, SSTD 2005, Angra dos Reis, Brazil, August 22-24, 2005, Proceedings*, ser. Lecture Notes in Computer Science, C. B. Medeiros, M. J. Egenhofer, and E. Bertino, Eds., vol. 3633. Springer, 2005, pp. 273–290. [Online]. Available: https://doi.org/10.1007/11535331_16

[45] "Real Datasets for Spatial Databases: Road Networks and Points of Interest, howpublished = https://www.cs.utah.edu/~lifeifei/spatialdataset.htm, note = Accessed: 2020-09-10."

[46] "UrbanGB, urban road accidents coordinates labelled by the urban center Data Set, howpublished = https://archive.ics.uci.edu/ml/datasets/urbangb%2c+urban+road+accidents+coordinates+labelled+by+the+urban+center, note = Accessed: 2020-09-10."

[47] C. Baldassi, "Recombinator-k-means: A population based algorithm that exploits k-means++ for recombination," *arXiv preprint arXiv:1905.00531*, 2019.

[48] B. T. Mark J. Huiskes and M. S. Lew, "New trends and ideas in visual concept detection: The mir flickr retrieval evaluation initiative," in *MIR '10: Proceedings of the 2010 ACM International Conference on Multimedia Information Retrieval*. New York, NY, USA: ACM, 2010, pp. 527–536.

[49] "MIRFLICKR Dataset Download, howpublished = http://press.liacs.nl/mirflickr/mirdownload.html, note = Accessed: 2020-09-10."

[50] L. Kaufman and P. Rousseeuw, *Clustering by means of medoids*. North-Holland, 1987.

[51] P. N. Yianilos, "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*, V. Ramachandran, Ed. ACM/SIAM, 1993, pp. 311–321. [Online]. Available: http://dl.acm.org/citation.cfm?id=313559.313789

[52] A. Asudeh, J. Augustine, A. Nazi, S. Thirumuruganathan, N. Zhang, G. Das, and D. Srivastava, "Efficient signal reconstruction for a broad range of applications," *SIGMOD Records, Special Issue on 2018 ACM SIGMOD Research Highlights*, vol. 48, no. 1, p. 42–49, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3371316.3371327

[53] P. Tune and M. Roughan, "Maximum entropy traffic matrix synthesis," *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 2, pp. 43–45, 2014.

[54] Y. Zhang, M. Roughan, N. Duffield, and A. Greenberg, "Fast accurate computation of large-scale ip traffic matrices from link loads," in *SIGMETRICS*, vol. 31. ACM, 2003, pp. 206–217.

[55] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*. IEEE, 2006.

[56] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava, "Hashed samples: selectivity estimators for set similarity selection queries," *PVLDB*, vol. 1, no. 1, pp. 201–212, 2008.

[57] L. Kleinrock and F. Kamoun, "Hierarchical routing for large networks performance evaluation and optimization," *Computer Networks*, vol. 1, no. 3, pp. 155–174, 1977.

[58] J. Gordon, "Pareto process as a model of self-similar packet traffic," in *Global Telecommunications Conference, 1995. GLOBECOM'95., IEEE*, vol. 3. IEEE, 1995, pp. 2232–2236.

BIOGRAPHICAL STATEMENT

Jees Augustine was born in Kerala, India. He received his Bachelor of Technology in Computer Science & Engineering from Mahatma Gandhi University, Kerala, India, in 2010, his Master of Engineering, in Computer Science from The Birla Institute of Technology & Science, Pilani, in 2013 respectively. After his Master program, he worked at Cisco Systems (India) Private Limited before joining Ph.D. program at The University of Texas at Arlington. He is a recipient of various fellowships, scholarships and awards, including Kelcy Warren Graduate Fellowship for Engineering in 2018 & in 2019, at The University of Texas at Arlington, Computer Science Engineering IAB Endowed Scholarship 2019, NSF - VLDB 2018 Travel Award, Cyneta Networks - Outstanding Computer Science and Engineering Teaching Assistant (Graduate) Award 2019. His work titled "Leveraging similarity joins for signal reconstruction" received ACM SIGMOD Research Highlights award in 2019, was invited to the VLDB Journal Special Issue on 'Best of VLDB 2018' and the first paper to be featured at Reproducibility Highlights of VLDB 2019. His work titled, "Efficient Signal Reconstruction for a Broad Range of Applications" is accepted to the Communications of the ACM's Research Highlight.