LOW LATENCY QUERIES ON BIG GRAPH DATA

BY

RACHIT AGARWAL

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Assistant Professor Philip Godfrey, Chair
Assistant Professor Matthew Caesar
Professor Bruce Hajek
Professor Jennifer Rexford, Princeton University
Professor Nitin Vaidya

Abstract

The availability of large datasets and on-demand system capacity to analyze these datasets has led to exciting new applications in the context of big graph data. Many big graph data applications — social search and ranking, personalized and socially-sensitive search, social network analysis, online advertising, to name a few — require computing distances and paths between vertices in the graph. Systems for these applications need to meet three performance goals: (1) low memory footprint; (2) low latency; and (3) small stretch — the ratio of the cost of path returned by the system to the actual shortest path. The theory community has established that meeting these goals is impossible for extremely dense graphs. The central theme of this dissertation is to show that these goals can, in fact, be achieved by exploiting *graph sparsity*, a property almost always encountered in big graph data.

This dissertation formally establishes a separation between the sparse and the dense cases for the problem of computing distances on graphs. For the realistic case of sparse graphs, our algorithms exhibit a smooth three-way trade-off between space, stretch and query time — a phenomenon that does not occur in dense graphs. Specific operating points on this trade-off space give us linear-space data structures for computing paths of stretch 2, 3 and larger, and the first data structure for computing paths of stretch less than 2 on general weighted undirected graphs.

We then apply our techniques and algorithms to build systems that enable efficient path computations for various big graph data applications. We first present ASAP, a system that almost always computes the exact shortest distance in tens of microseconds on graphs with millions of vertices and edges. We then present ShapeShifter, a system that enables efficient computation of short paths on dynamic graphs; ShapeShifter can update, upon an edge insertion and/or deletion, the underlying data structure within tens of microseconds and answers each user query in less than a millisecond.

To my family

Acknowledgments

I feel very privileged to have worked with my thesis advisors, Matthew Caesar and Brighten Godfrey. To both of them, I owe a great debt of gratitude for their patience, support and friendship. For the last four years, Matt has been a brilliant force in guiding me towards the questions. He has an uncanny ability to challenge fundamental assumptions, and to synthesize new research problems surrounding these assumptions. He has instilled in me the taste for important research, and for that I thank him. When it comes to finding answers, Brighten is amazing. Be it an academic problem or a personal one, I always found him standing right beside me, knowing exactly what to do. Over the last four years, he has worked very hard to teach me the skills of understanding deep technical ideas in their most simplistic form, and explaining my own ideas concisely and precisely. Thank you Matt and Brighten, thank you.

I was also very fortunate to have a great set of people to advise me during my graduate studies. Two people who stand out in shaping my career are Ralf Kötter and Nitin Vaidya. I owe much of my academic career to Ralf — he was the first person to teach me the art of doing good research, writing papers and questioning the questions. I miss him. My first research project at UIUC was with Nitin. He supported me in my first year of grad school, and continued doing so until the end. Thank you Ralf and Nitin. I would also like to thank Jennifer Rexford and Bruce Hajek for serving on my dissertation committee. A part of my thesis work is theoretical in nature; at times when I was exploring these questions, I found it very useful to have Sariel Har-Peled, Chandra Chekuri and Jeff Erickson around me. They helped me better formulate the problem, provided me with directions, and welcomed me into the theory world. Thank you.

I was the first student of Brighten and Matt; most of the other students in the group were at least a couple of years younger. One could imagine this having both positive and negative aspects. On the positive front, I could ask stupid questions during the group meetings and still be perceived as smart! On the negative front, I had nobody but Brighten and Matt to guide me when I needed direction. Surprisingly, the other students in the group — Virajith Jalaparti, Ashish Vulimiri, Chia-Chi Lin, Ankit Singla, Chi-Yao Hong, Qingxi Li, Wenxuan Zhou — made both the positive and negative fade away within a short period of time. They were extremely smart to challenge my questions and surprisingly mature to even suggest directions. Thank you all for making my experience in Siebel Center so enjoyable.

My life would not be so enjoyable without having around so many friends. Many thanks to Riccardo, Virajith, Parikshit and Ravi for those wonderful evenings, chatting and drinking. Myungjin Lee, who is now a professor, has extended his time to me whenever I needed it; he is awesome. It was only during the lonely hours at Urbana-Champaign when I realized what jewels I had in the friends I made at IIT Kanpur. Gopal, Vibhor, Nidhi, Nisheet, Gunjan, my life would have been terrible without them!

There are not enough words to express my thanks to my family for their support and love and for having faith in me. My mother, Sadhna Agarwal, has made many sacrifices and I want to let her know that her sacrifices have not gone unnoticed. My father, Ghanshyam Das Agarwal, has always kept very high expectations but has never lost faith in me; this, in turn, has forced me to work harder and maintain my focus. Thanks Mom and Dad. My sister, Rachna Agarwal, has always extended her unconditional love and support to me, whenever I needed it the most. She has never asked anything but love in return; I love you sis and I thank you for making me a better person. Last, but not the least, I would like to thank my wife Gargi for standing by me all the time. Thank you for your love, support, encouragement, friendship, and for everything you have coloured me with. This work is dedicated to my family, remembering those moments when I shamefully neglected their presence.

Table of Contents

Chapter 1

Introduction

Big data refers to datasets that are *large* and *complex*. The sheer size of these datasets renders inefficient the existing approaches to storing, processing and networking. For instance, every day (on an average), Google generates and processes more than 20 petabytes of data [1]; Facebook loads 60 to 90 terabyte (TB) of uncompressed data to its servers [2]; the New York Stock Exchange generates about a TB of trade data [3]; the Large Hadron Collider produces about 41 TB of new data and transfers it over the Internet [4]; similar examples appear in biology [5] and chemoinformatics [6].

However, scale is only one aspect of big data; the fundamental challenges in big data arise due to an unprecedented complexity. Indeed, the current and emerging problems in big data require understanding the structure of the data, dealing with data redundancy and accuracy, formulating meaningful analysis metrics and managing storage and compute networks that manage big data; with humans in the loop, addressing the human factors of comprehending complex datasets also becomes a significant challenge.

A significant component of big data is modeled as graphs, and is referred to as *big graph data*. This includes big data relating to world wide web, social networks, the Internet, etc. Big data applications on these graphs encompass web graph analysis [7–9], social network analysis [10–12], designing efficient search engines [7, 13], analyzing information propagation [10, 14], network security [15–17], to name a few. A fundamental goal in many of these applications is to ascertain the user "interest" — web search engines often need to predict the content of interest to the user conducting the search; online advertisement industry needs to predict the products of interest to the user visiting the webpage; social networks need to predict other users and content of interest to the user, etc. A natural way to formalize the notion of interest is by using the proximity between the user and the content, where proximity is defined according to some distance measure on the underlying graph data.

This dissertation concerns building techniques, algorithms and systems for big graph data applications that require distance computations. We will discuss several concrete applications below; however, we note that since most of these applications compute distances in response to a user query, the goal is to minimize the query latency while maintaining feasible memory requirements; indeed, it is also desirable to compute the exact shortest distance or a distance estimate which is very close to the exact shortest distance. The theory community has established that meeting these goals is impossible for extremely dense graphs. The central theme of this dissertation is to show that these goals can, in fact, be achieved by exploiting *graph sparsity*, a property almost always encountered in big graph data.

We begin by developing techniques and algorithms that allow computing paths of small *stretch*, defined as the worst-case ratio of the distance returned by the algorithm to the actual shortest distance between the two vertices. Our first contribution is to formally establish a separation between the sparse and the dense cases for the problem of computing distances on graphs. For the realistic case of sparse graphs, our algorithms exhibit a smooth three-way trade-off between space, stretch and query time — a phenomenon that does not occur in dense graphs.

Specific operating points on this trade-off space give us linear-space data structures for computing paths of stretch 2, 3 and larger, and the first data structure for computing paths of stretch less than 2 on general weighted undirected graphs. Applying our techniques to the problem of routing in networks with limited memory, we get a distributed routing protocol that uses little router memory and yet routes along paths that are shorter than what was previously thought possible.

We then use our techniques and algorithms to build systems that enable efficient path computations for various big graph data applications. We first present ASAP, a system that almost always computes the exact shortest distance in tens of microseconds on graphs with millions of vertices and edges. Finally, we present ShapeShifter, a system that allows to efficiently compute short paths on dynamic graphs; ShapeShifter can update, upon an edge insertion and/or deletion, the underlying data structure within tens of microseconds and answers each user query in less than a millisecond.

1.1 Distance Queries and Applications

We start by informally defining exact and approximate distance queries on graphs. We then discuss a number of big graph data applications that perform exact and approximate distance queries.

1.1.1 Exact and Approximate Distance Queries

Consider a social network (Facebook, LinkedIn, Google+, etc.) and denote the set of users as V with each user having a set of "friends". To model the underlying data as a graph, one can imagine a "vertex" corresponding to each user in the network; the friendship between a user u and another user v can then be modeled as an "edge" between two vertices in the graph leading to set of edges E. Furthermore, each edge can be assigned a "weight" which could, for instance, be a measure of how frequently the two users constituting the edge interact. The data is then said to be modeled as a graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$. If modeled in such a manner, a path between two users u and v is simply an ordered set of users $(u \rightsquigarrow v_1 \rightsquigarrow v_2 \cdots \rightsquigarrow v)$ with edges between users u and v_1, users v_1 and v_2 and so on. The cost of the path is the sum of the weights of each edge along the path. The "distance" between two users s and t is then the cost of the least-cost path between s and t.

Given a graph $G = (V, E)$ and two vertices $s, t \in V$, a *distance query* asks for the distance between s and t; a closely related question is that of *path query*, which also requires to list one of the paths corresponding to the distance between s and t. A query is said to be a *shortest path query* if the desired output is in fact the shortest path between s and t. For many applications, however, an approximately shortest path suffices; the approximation is measured in terms of stretch — the worst-case ratio of the distance returned by the query to the actual shortest distance between the two vertices. A query is said to return distances of stretch c, if for any pair of vertices at distance d, the query returns a distance of at most $c \cdot d$.

1.1.2 Applications

In this section, we give a non-exhaustive list of contemporary applications where computing short paths is a key component.

Search and ranking of people on social networks

A common operation in social networks is to search for people; indeed, social networks are meant to connect old friends and make new ones. For instance, a study conducted by Google on their social network Orkut [18] suggests that more than 50% of the searches are for other users, with each query having less than 2 words per query. This makes it important for social network service providers to devise efficient techniques that allow searching people and ranking the search results.

However, devising a technique for search and ranking of people on social network is non-trivial for several reasons. First, the problem is fundamentally different from that of traditional problem of web search, that uses text-based ranking techniques and has no notion of "user preference". For instance, an experiment conducted in [18] suggests that the average number of answers per query when the retrieval algorithm is based on an exact match between the query and the user name (that is, for the query "Maria", only the users who declared their names exactly as "Maria" are retrieved) is 48. Furthermore, if partial matches between the user name and the query are allowed (that is, the query "Maria" provides a match to the user named "Maria A."), the average number of results per query is increased to 6034. In the absence of any user preference, there is no way to rank these results. This demonstrates the ineffectiveness of traditional text-based ranking techniques. Indeed, [18] proposed a distance-based ranking technique, leaving open the question of efficiently (that is, with low latency) computing distances on large graphs.

Currently used techniques for computing such distance-based rankings require preprocessing and storing the distance from each vertex u to vertices within certain number of *hops* from u. This approach is limited due to several factors. First, these techniques have high memory footprint and require dedicated servers and resources for managing extremely large datasets. Perhaps more fundamentally, such an approach is limited to rankings based on hop-distance and cannot handle graphs with weighted edges.

Search and ranking of paths on social networks

The previous application of search and ranking of people on social networks requires computing, from a user u, shortest distances to users in a set X (for instance, all users that match the query "Maria"). In professional social networks like LinkedIn and Microsoft Academic Search, a different kind of social query is initiated. Here, the goal of a user Alice is to connect to another user Bob. The goal of the social network service provider is to compute and rank a set of paths between users Alice and Bob.

This problem is significantly more challenging than the previous one. First, the social network has to devise techniques to compute these paths on the fly if paths between Alice and Bob were not stored during the preprocessing phase. Second, even if paths can be precomputed and stored, the memory requirements increase significantly — if 10 paths are stored between each pair of users, the memory requirements are expected to be an order of magnitude larger than storing a single shortest path. Finally, as in the previous application, the problem becomes significantly more challenging if one desires to take into the account the edge weights.

Socially-sensitive content search and ranking

Traditionally, web search results only reflect how important a particular piece of content is — here, the importance of a document is defined by computing a ranking function, PageRank for instance. Recently, there has been an increasingly intense discussion about personalized and socially-sensitive web search. For instance, it has been argued that the distance between the point where the query is initiated (here the initiation point may be the query context and not necessarily an user) and the relevant webpages is an important aspect in the ranking of the results [19]; similarly, [20] argues that a user may be more interested in finding contents from users that are close to her in the social graph. In this context, incorporating distance based ranking functions for search tasks has been proposed in several recent papers [18, 21–23].

Location-aware search is a more general form of socially-sensitive search that has relevance in information retrieval community [24] — it has been found that people who chat with each other are more likely to share interests; and this observation is used to retrieve information relevant to the users.

The growing interest in involving context and/or social connections in search tasks, suggests that ranking functions may soon incorporate (some form of) distance computations; see an experimental exploration of this socially-sensitive search in [25–27]. Indeed, the problem is pretty challenging — given the large number of possible search attributes and the large size of the underlying graph, it is practically impossible to precompute and store the results; on the other hand, computing such ranking functions entirely on the fly leads to high latency. It is, hence, desirable to design techniques that simultaneously achieve low memory footprint and low latency for socially-sensitive and location-aware search and ranking.

Social network analysis

Distance queries over social networks have also been used to analyze information dissemination [28, 29], to detect communities [27, 30] to estimate structural similarity between two given networks [31], to compute graph separation metrics [25, 26, 32, 33], to compute centrality measures [25, 26], etc. In addition, algorithms for detecting Sybil attacks rely on detecting communities [16] and hence, can benefit from efficiently answering distance queries.

Routing in networks with limited memory

The distance query problem in big graph data is related to the problem of scalable routing on large networks. In the latter problem, it is desirable that routers require limited memory to store forwarding tables [34–37] and yet route along short paths. Traditionally, the technique used to route using limited memory is hierarchy — the network is partitioned into multiple domains and separate routing protocols are used across and within domains; across domains, routing is performed on higher level aggregates (e.g., IP prefixes), while a separate protocol that typically implements shortest path routing is used within a single domain. While traditional approaches have been able to sustain the network growth, they can be inefficient: in the worst-case, the routes used by traditional schemes can be arbitrarily longer than the shortest path, leading to high network latency.

1.2 Goals and State-of-the-art

In this section, we outline a number of performance goals for systems and algorithms for answering distance queries based on the applications discussed above. We then briefly discuss the state of the art techniques for answering distance queries and outline the limitations of these techniques.

1.2.1 Goals

We argue that systems and algorithms for answering distance queries must meet the following three goals: (1) low latency; (2) an extremely small constant stretch as close to 1 as possible; and (3) feasible memory requirements. We discuss these goals in more depth below.

Low Latency

Most of the applications discussed in the previous section compute distances and paths in response to a user query. This imposes stringent latency requirements on systems and algorithms for answering user queries. For instance, a study conducted by Google [38] suggests that increasing the query latency from 100 to 400 milliseconds led to significant reduction in the number of user queries, leading to revenue loss. The problem is further exacerbated since many applications (for instance, the application of search and ranking of people on social networks discussed above) initiate multiple sub-queries for a single query. Consequently, one of the most important goals is to design techniques, algorithms and systems that answer each query extremely quickly, with typical latency requirement being less than a few milliseconds.

Low Stretch

Most of the applications above require or could benefit from computing shortest paths. However, if all the desired distances cannot be precomputed and stored, this may be infeasible. In such a case, the structure of social networks makes it necessary that the returned distance estimate be of low stretch. For instance, consider a pair of users at distance 2 (that is, one user is the friend of friend of another user). Then, if the algorithm returns a distance estimate

of stretch 3, the returned distance estimate will be 6. However, for all practical purposes, this estimate is useless since any pair of users in real-world social network is less than 6 hops away due to small-world property of social networks. Hence, our goal is to design systems and algorithms that return a *distance estimate of stretch less than* 3.

Feasible Memory

It is rather trivial to achieve the first two goals if one has access to machines with extremely large memory (by precomputing and storing all-pair shortest paths). However, memory limitations and the size of networks in question mean that simple solutions, like precomputing and storing all-pair shortest paths, are infeasible; even for a social network with 3 million users, this would require roughly 4.5 trillion entries. Social networks of interest can in fact be much larger in size — Facebook (1+ billion users [39]), Twitter (500 million users) and LinkedIn (200 million users [40]). Hence, it is desirable to minimize the memory requirements while meeting the above two goals.

1.2.2 State-of-the-art

We briefly discuss the state-of-the-art for answering distance queries on big graph data. A distance oracle is a compact representation of all-pair shortest path matrix of a graph. A stretch-c oracle for a weighted undirected graph $G = (V, E)$ returns, for any pair of vertices $s, t \in V$ at distance $d(s, t)$, a distance estimate $\delta(s, t)$ that satisfies $d(s, t) \leq \delta(s, t) \leq c \cdot d(s, t)$. Let $n = |V|$ be the number of vertices and $m = |E|$ be the number of edges in the graph.

For general weighted undirected graphs, Thorup and Zwick [41] showed a fundamental space-stretch trade-off — for any integer $k \geq 2$, they designed an oracle of size $O(kn^{1+1/k})$ that returned distances of stretch $(2k-1)$ in $O(k)$ time; the construction time of their oracle was $\widetilde{O}(kmn^{1/k})$, in expectation. Thorup-Zwick oracle was a significant improvement over previous constructions that had much higher stretch and/or query time [42–44].

The space-stretch trade-off of Thorup-Zwick oracle (TZ-oracle) is essentially optimal, assuming the girth conjecture of Erdős. In particular, Thorup and Zwick [41] showed that any oracle for undirected graphs that re-

turns distances of stretch less than $(2k+1)$ must have size $\Omega(n^{1+1/k})$. Their lower bound proof is information-theoretic, essentially showing the existence of dense-enough graphs that are incompressible: if a certain stretch is desired, then the size of the data structure is lower bounded by the number of edges in the specially-constructed graph. For example, proving that stretch less than 3 requires $\Omega(n^2)$ space uses a graph with $\Theta(n^2)$ edges. Hence, the space-stretch trade-off of their oracles is optimal only for the obscure case of extremely dense graphs.

1.3 Dissertation Outline

Can lower stretch be achieved using sub-quadratic space for the realistic case of sparse graphs? This question is both interesting and important for two reasons. First, far from being a narrow special case of the problem, sparse graphs are the most relevant case. Nearly all large real-world networks are sparse, including road networks [45], social networks [11], the router-level Internet graph [46] and the Autonomous System-level Internet graph [46], as well as networks like expander graphs that are important in many settings. For instance, letting $\mu = c \log_2 n$, empirically, $c \approx 0.6$ for an AS-level map of the Internet [46], $c \approx 0.4$ for a router-level map of the Internet [46], and $c \approx 1.34, 0.65, 1.21, 5.10, 29.9$ for social networks Cyworld, Testimonial, Orkut, MySpace, and Facebook, respectively [11, 47]. There is no hope of the proof technique of TZ lower bound being helpful for these graphs, that is, graphs with much less than n^2 edges since this technique will only show that achieving any constant stretch value requires $\Omega(m)$ bits.

The second reason sparse graphs are interesting is that the mathematical structure of the question changes dramatically in the case of sparse graphs. Indeed, if $\Omega(m)$ space is allowed, one can trivially construct stretch-1 oracles (that is, oracles that return the shortest path) by storing the original graph and running a shortest path algorithm for each query; this, however, takes time $O(m)$ per query. Thus, in the context of distance oracles, the cases of dense and sparse graphs are quite different. In the dense case the key is to *compress* the graph while ensuring that sufficient information remains to return low-stretch distances. In the sparse case the graph need not be compressed, but the trade-off with *query time* becomes critical.

The first part of the dissertation builds an understanding of distance oracles beyond the Thorup-Zwick bound. In particular, we explore several questions:

> Is it possible to design oracles of size $o(n^{3/2})$ that return distances of stretch 3 in time $o(m)$ for sparse graphs?
>
> Do constant-stretch oracles with sub-linear query time and linear space exist or do we necessarily require super-linear space, that is space $\Omega(m^{1+\delta})$ for some $\delta > 0$?
>
> Is it possible to design oracles of size $o(n^2)$ that return distances of stretch less than 3 in time $o(m)$ for sparse graphs?
>
> Is there in fact a smooth trade-off between space and query time for any fixed stretch?
>
> Do the space and query time reduce smoothly as the graph gets sparser?

The second part of the dissertation builds systems using techniques developed in the first part. We show that the new techniques not only improve the worst-case stretch of Thorup-Zwick oracles (using the same space) but empirically, lead to schemes with average stretch extremely close to 1. Next, we summarize the results in the dissertation and outline our contributions.

1.3.1 Oracles with Linear Space

Chapter 3 explores these questions for stretch 3 and larger. Let S denote the size of the oracle and let T denote the query time. Our main result is design of stretch-3 oracles for each point on the space-time curve $S \times T^2 = O(n^2)$ for sparse graphs; we get similar results for stretch larger than 3 (see Figure 1.1 for a simple visualization of this space-time trade-off).

This answers all our questions above: for any graph, there is indeed a smooth space-time trade-off — for any fixed stretch, it is possible to reduce the space requirements (of the corresponding constant-time oracle) at the cost of higher query time. Moreover, the space-time trade-off improves as the graph gets sparser. Finally, and perhaps most interestingly, there exist oracles of size

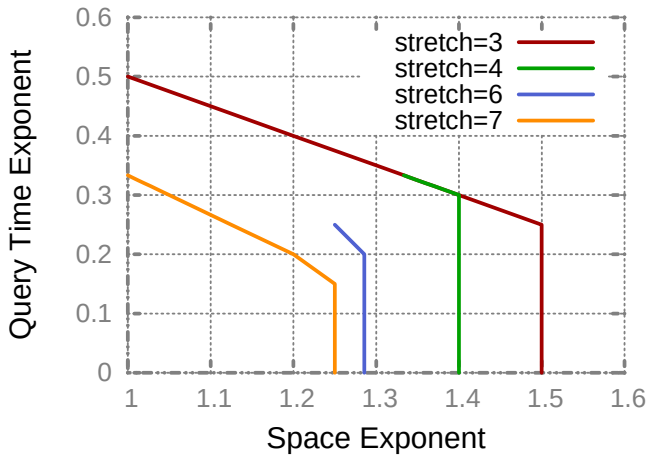


Figure 1.1: Space-time trade-off for our oracles for stretch 3 and larger for graphs with $m = \widetilde{O}(n)$ edges. Let S be the size of the oracle and let T be the query time; then, the space and the query time exponents are defined as $\log_n(S)$ and $\log_n(T)$, respectively.

linear in the input size that can compute distances of constant stretch in sub-linear query time; for instance, it is possible to design oracles of size $\widetilde{O}(m)$ that return stretch-3 distances in time $O(\sqrt{m})$. For computing distances of stretch $4k-1$, for any integer $k \geq 1$, our linear-space oracles require query time $O(m^{1/(k+1)})$.

Our second contribution is an extremely simple construction of constant-time oracles that return distances of stretch $2k$, for any $k \geq 2$. These oracles have size $\widetilde{O}(m^{1-\frac{2}{2k+1}} n^{\frac{4}{2k+1}})$. These are the points with zero query exponent in Figure 1.1. For unweighted graphs, the space can be reduced to $\widetilde{O}(n^{1+\frac{2}{2k+1}})$ at the expense of an additive stretch of 1. The results in this chapter appeared in [48, 49].

1.3.2 Oracles for stretch 2

Chapter 3 explores one direction in which TZ-oracles can be improved for the case of sparse graphs — for any fixed stretch, it is possible to design oracles that require less space compared to the TZ-oracle at the expense of higher query time. Chapter 4 explores improvement in TZ-oracles in the other dimension — reducing the stretch at the expense of higher query time. In fact, we show that for sparse graphs, it is possible to reduce both the stretch and the space of TZ-oracle at the expense of sub-linear query time.

We give several results for stretch 2. Our first result is construction of stretch-2 oracles for each point on the space-time trade-off of $S \times T = O(n^2)$ for sparse graphs; we get similar results for denser graphs with space-time trade-off dependent on graph density. Hence, for $S = O(n^{3/2})$ as in TZ-oracle, it is possible to compute stretch-2 distances in sparse graphs using time $O(\sqrt{n})$. As with stretch-3 oracles, we get a smooth space-time trade-off that improves as the graph gets sparser. Furthermore, for graphs with $m = \Omega(n^{1+\varepsilon})$ edges for any $\varepsilon > 0$, we also get linear-space oracles that require sub-linear query time. Figure 1.2 shows the space-time trade-off of our oracles.

The stretch-2 oracle above leads to a $\widetilde{O}(m^{1/2}n^{3/2})$ time algorithm for computing all-pair stretch-2 distances, matching the run time of a decade-old result due to Cohen and Zwick, albeit using significantly different and simpler techniques. A way to interpret this result is that the trade-off between the query time and the construction time of our oracle is optimal unless there exists a faster algorithm for computing all-pair stretch-2 distances; in other words, improving either the query time or the construction time of this oracle will lead to an asymptotically faster combinatorial algorithm for computing all-pair stretch-2 distances.

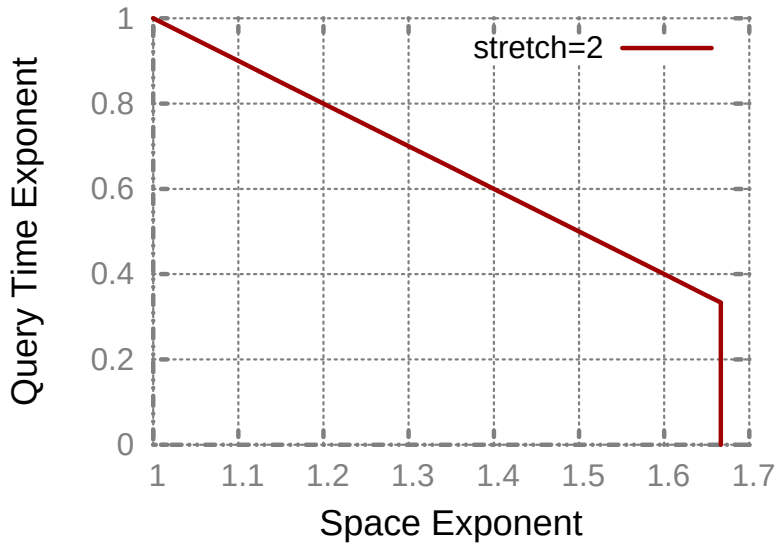


Figure 1.2: Space-time trade-off for our oracles for stretch 2 for graphs with $m = \widetilde{O}(n)$ edges. Let S be the size of the oracle and let T be the query time; then, the space and the query time exponents are defined as $\log_n(S)$ and $\log_n(T)$, respectively.

We present two more results in this section. First, we give an extremely simple construction of constant-time stretch-2 oracle of size $\widetilde{O}(n^{4/3}m^{1/3})$ (the point with zero query exponent on the S-T curve of Figure 1.2). Second, we apply our results on stretch-2 oracles (with super-constant query time) to the problem of routing in networks with limited memory; we get a distributed routing protocol that uses little router memory and yet routes along paths that are shorter than what was previously thought possible. The results in this chapter appeared in [48–50].

1.3.3 Oracles for Stretch Less Than 2

Chapter 5 presents the first oracles that compute distances of stretch less than 2 on general weighted undirected graphs. As with oracles for stretch 2 and larger, our oracles achieve a three-way trade-off between space stretch and query time. For sparse graphs, our oracles achieve a space-stretch-time trade-off of $S \times T^{1/k} = O(n^2)$ for computing distances of stretch $1+1/k$; the trade-off can be further improved for certain values of stretch. For instance, Figure 1.3 shows the space-time trade-off for our stretch-1.67 oracles.

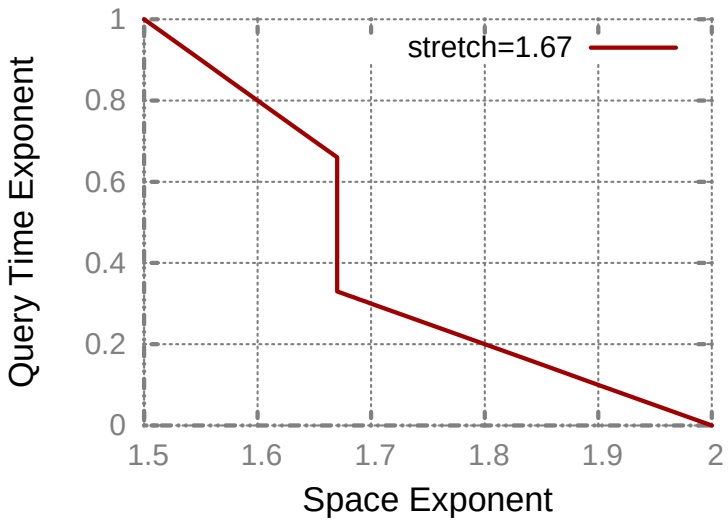


Figure 1.3: Space-time trade-off for our oracles for stretch 1.67 for graphs with $m = \widetilde{O}(n)$ edges. Let S be the size of the oracle and let T be the query time; then, the space and the query time exponents are defined as $\log_n(S)$ and $\log_n(T)$, respectively.

As with our stretch-2 oracle, we argue that our oracles for stretch less than 2 may achieve an optimal trade-off between query time and construction time, unless there exists a faster algorithm for Boolean Matrix Multiplication (BMM). Specifically, the problem of computing all-pair stretch-less-than-2 distances in undirected graphs is equivalent to combinatorial BMM over the (OR, AND) semiring. Let T denote the query and let T' denote the construction time of our stretch-1.667 oracles. If we can reduce the query time to $T^{1-\varepsilon}$, for any $\varepsilon > 0$, without increasing the construction time (or vice versa), it would be possible to multiply two Boolean matrices in time $o(mn)$. This would lead to a purely $o(mn)$ time combinatorial algorithm for BMM, a long standing open problem. The results in this chapter appeared in [51, 52].

1.3.4 Shortest Paths in Microseconds

Motivated by the application discussed in §1.1, we apply our techniques from Chapter 3, Chapter 4 and Chapter 5 to build systems for computing short paths for big graph data applications. Chapter 6 presents ASAP, a system that quickly computes shortest paths by exploiting the structure of big graph data.

ASAP preprocesses the network to compute a *partial shortest path tree* (PSPT) for each vertex. PSPTs have the property that for any pair of vertices, each edge along the shortest path is very highly likely to be contained in the PSPT of either the source or the destination. Hence, a shortest path can be computed by simply exploring the PSPT of the source and the destination. ASAP demonstrates and exploits the observation that the structure of big graph data enables the PSPT of each vertex to be an extremely small fraction of the entire network; hence, PSPTs can be stored efficiently and each shortest path can be computed extremely quickly.

ASAP, even on networks with millions of vertices and edges, computes shortest paths in *tens of microseconds* using a single machine. Furthermore, unlike most previous works, ASAP admits efficient distributed implementation and can be easily mapped on distributed programming frameworks like MapReduce. Finally, unlike any previous technique, ASAP can compute multiple paths between any given pair of vertices using the same data structure as the one used for single path computation and will minimal latency increase.

Table 1.1: Summary of results for ASAP on several datasets (see Chapter 6). "Accuracy" refers to the fraction of the vertex pairs (approximated to two decimal places) for which ASAP returns the shortest path.

Dataset	ASAP				Speed-up
	#Paths = 1		#Paths > 1		(compared to
	Time (in μs)	Accuracy	Time (in μs)	#Paths	State-of-the-art)
DBLP	20.3	1.00	31.6	173	916×
Flickr	26.5	1.00	52.0	523	3171×
Orkut	31.8	0.94	65.0	237	23963×
LiveJournal	48.9	1.00	99.2	453	3197×

ASAP, even on network with millions of vertices and edges, computes the shortest path between most vertex pairs in less than 50 μs; see Table 1.1. ASAP also allows computing hundreds of paths and corresponding distances between most vertex pairs in less than 100 μs without any change in the data structure for single shortest path computation. The results in this chapter appeared in [53, 54].

1.3.5 Shortest Paths on Dynamic Graphs

A particularly challenging problem in big graph data is to handle graph dynamics — insertions and deletions of edges and vertices over time. Chapter 7 presents ShapeShifter, an extension of ASAP from Chapter 6 that enables quick computation of distances on dynamic graphs. This extension uses our techniques from Chapter 5 along with some new ideas for quickly updating vertex partial shortest path trees (PSPT).

The main idea is to compute and store PSPTs that are smaller in size than the PSPTs stored in ASAP. Smaller PSPTs not only reduce the space requirements of ShapeShifter (compared to ASAP), but also enable quick updates since the time taken to compute a PSPT is proportional to the size of the PSPT; in addition, ShapeShifter uses new techniques to quickly identify and update the set of PSPTs that are affected by an update. The challenge, however, is that the PSPTs of any vertex pair may no more intersect. To resolve this, ShapeShifter computes larger PSPTs on the fly leading to almost perfect accuracy.

Table 1.2: Summary of results for ShapeShifter on several datasets (see Chapter 7). For vertex pairs whose PSPT intersect along the shortest path, ShapeShifter returns the shortest path; otherwise, ShapeShifter returns a low stretch path.

Dataset	Fraction of intersecting PSPTs		Query time (in μs)	Average Update time (in ms/update)
	Total	Shortest Path		
DBLP	1	0.98	414.2	1.5
Flickr	1	0.93	521.0	1.7
Orkut	1.00	0.91	922.1	2.1
LiveJournal	1.00	0.96	997.1	2.4

ShapeShifter, on networks with millions of vertices and edges, computes shortest paths for a large fraction of vertex pairs in less than a millisecond (see Table 1.2); in addition, ShapeShifter returns a low stretch path for all other vertex pairs. ShapeShifter can handle thousands of edge insertions and deletions in a second on a single machine; furthermore, the update time of ShapeShifter decreases almost linearly by using multiple cores and multiple machines. The results in this chapter appeared in [55].

Chapter 2

Preliminaries

This chapter builds up the basic foundation for the rest of the dissertation. We begin with some formal definitions related to graphs in §2.1. We then define balls, vicinities, inverse-balls and inverse-vicinities of graph vertices in §2.2; these are certain neighborhoods of vertices in the graph that are used in our results. Finally, in §2.3, we briefly review the distance oracle of Thorup and Zwick (TZ) [41], and follow-up research on improving the original TZ-oracle. We also discuss, in §2.3, the known lower bounds on distance oracles.

2.1 Graphs

We start with some basic definitions and terminologies related to graphs.

Definition 1 (**Graphs**). *A graph G is a pair $G = (V, E)$, where V is the set of vertices in the graph and $E \subseteq \binom{V}{2}$ is the set of edges in the graph. The graph is said to be undirected if the edges have no orientation, that is, for any pair of vertices $u, v \in V$, the edge (u, v) is identical to (v, u). The graph is said to be directed otherwise; in such a case, the edges are defined as an ordered pair $E \subseteq V \times V$.*

Social networks and information networks are often modeled as graphs. For instance, each user in a social network is modeled as a vertex; two users have an edge connecting between them if they are "friends". Most of the early social networks (Facebook, Orkut, LinkedIn) have bidirectional edges — if Alice and Bob are friends, the edge between Alice and Bob is symmetric. These networks are modeled as undirected networks. Many recent social networks (Google+ and Twitter, for instance) have unidirectional edges and are modeled as directed networks.

Definition 2 (**Neighbors of a vertex**). *Two vertices $u, v \in V$ of a graph $G = (V, E)$ are said to be adjacent if there is an edge between u and v, that is, if $(u, v) \in E$. For a graph $G = (V, E)$, the set of neighbors of any vertex $v \in V$, denoted by $N(v)$, is the set of all vertices that are adjacent to v, that is, $N(v) := \{u : (u, v) \in E\}$. For a subset of vertices $U \subseteq V$, the set of neighbors of vertices in U, denoted by $N(U)$, is defined as: $N(U) = \bigcup_{u \in U} N(u)$.*

Definition 3 (**Degree of a vertex**). *For a graph $G = (V, E)$, the degree of a vertex $v \in V$, denoted by $\deg(v)$, is defined as the number of its neighbors, that is, $\deg(v) := |N(v)|$.*

Definition 4 (Δ-**maximum degree bounded graphs**). *A graph $G = (V, E)$ is said to be Δ-maximum degree bounded graph (or equivalently, Δ-degree bounded graph) if the degree of each vertex in G is at most Δ, that is, for each vertex $v \in V$, $\deg(v) \leq \Delta$.*

Definition 5 (μ-**average degree bounded graphs**). *A graph $G = (V, E)$ is said to be μ-average degree bounded graph (or equivalently, has average degree μ) if $\mu = 2m/n$.*

The notion of μ-average degree bounded graphs and Δ-maximum degree bounded graphs will play a crucial role in our construction. Note that a Δ-maximum degree bounded graph is also a Δ-average degree bounded graph; however, the reverse in not true since some vertices in the graph may have degree higher than Δ.

Definition 6 (**Edge Weight**). *A graph $G = (V, E)$ is said to be weighted if it is associated with a weight function $w : E \to \mathbb{R}$ that assigns a weight to each edge in G. The graph is said to be unweighted if each edge is assigned the same weight.*

The weight of an edge is perhaps the most natural way to differentiate between any two edges in the graph. Edge weights in social networks can signify how "strong" the relationship is or how frequently the two user constituting the link interact. Edge weights in information networks may signify the delay of sending a packet from one router to another router. We now define the shortest paths and the notion of stretch.

Definition 7 (**Paths**). *A path in G from a vertex $s = u_0$ to another vertex $t = u_k$ is a sequence of edges $\{(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, (u_k))\}$. Alternatively, a path is denoted as an ordered sequence of adjacent vertices $(u_0, u_1, \ldots, u_k)$. The length of a path P is the sum of its edge weights, that is, $length(P) := \sum_{i=0}^{k-1} w(u_i, u_{i+1})$. The hop-length of a path P is the number of edges in P.*

Definition 8 (**Shortest Paths**). *Let $\mathcal{P}$ denote the set of paths between s and t in G. The shortest distance (or equivalently, the exact distance or simply the distance) between s and t in G, denoted by $d(s,t)$, is defined as the length of the shortest path between s and t. More formally $d(s,t) := \min_{P \in \mathcal{P}} length(P)$. If $\mathcal{P} = \emptyset$, we let $d(s,t) := \infty$.*

Definition 9 (**Connected graphs**). *An undirected graph $G = (V, E)$ is said to be connected if $d(s,t)$ is finite for all pairs of vertices $s, t \in V$.*

Definition 10 (**Stretch**). *Let P be a path between a pair of vertices s, t in G. Then, P is said to be a path of stretch-k if $d(s,t) \leq length(P) \leq k \cdot d(s,t)$.*

In this dissertation, unless stated otherwise, we consider connected weighted undirected graphs with each edge assigned a non-negative weight. Assuming connectedness is not fundamental to our results but simplifies the exposition of our techniques and results; all our results hold for graphs with multiple disconnected components. Assuming non-negative edge weights and undirected graphs is, however, fundamental. In particular, we will require that the paths on the input graph constitute a metric space [56]. That is, they have the following three properties: (1) for any pair of vertices $u, v \in V$, we have that $d(u,v) \geq 0$; (2) for any pair of vertices $u, v \in V$, we have that $d(u,v) = d(v,u)$; and (3) for any triplet of vertices u, v, w, we have that $d(u,v) \leq d(u,w) + d(w,v)$. The last of the above three properties is known as **triangle inequality**.

The notation used in the above definitions is summarized in Table 2.1.

2.2 Balls and Vicinities

In this section, we start with formally defining the vertex balls and vertex vicinities. We then discuss efficient algorithms to construct vertex balls and vicinities.

Table 2.1: Notation used throughout the dissertation.

G	A connected weighted undirected graph
V	Set of vertices in the graph
E	Set of edges in the graph
n	Number of vertices in the graph
m	Number of edges in the graph
$N(u)$	Neighbors of vertex u
$N(U)$	Neighbors of vertices in U
$\deg(u)$	Degree of vertex u
$d(s,t)$	Shortest distance between vertices s and t
Δ	Maximum degree in the graph
μ	Average degree of the graph

2.2.1 Definitions and notation

Definition 11 (Landmark vertex). *Let $G = (V, E)$ be a weighted undirected graph and let $L \subset V$ be a subset of vertices. The landmark vertex of any vertex v, denoted by $\ell(v)$, is the vertex $\ell \in L$ that minimizes $d(\ell, v)$, ties broken arbitrarily.*

The set L in the above definition will be referred to as the set of "landmarks". The notion of landmarks is used to define certain neighborhood of vertices in the graph. Of particular interest are the notion of balls and vicinities:

Definition 12 (Ball and ball radius of a vertex). *Let $G = (V, E)$ be a connected weighted undirected graph and let $L \subset V$ be a subset of vertices. The ball of a vertex $v \in V$, denoted by $B(v)$, is the set of vertices $w \in V$ for which $d(v, w) < d(v, \ell(v))$. The ball radius of v, denoted by r_v, is the distance from v to its landmark vertex, that is, $r_v := d(v, \ell(v))$.*

In other words, the ball of a vertex v is the set of all vertices w that are strictly closer to v than its landmark vertex $\ell(v)$.

Observe the following interesting property of the ball of any vertex v. Let w and w' be two vertices such that $d(v, w) \leq d(v, w')$; then, if $w' \in B(v)$, we have that $w \in B(v)$. That is, if any vertex w' is contained in $B(v)$, then all vertices at distance less than or equal to $d(v, w')$ are contained in the ball of v. Next, we define the vicinity of a vertex; this definition is closely related to the definition of the balls but has a dramatically different structure.

Definition 13 (**Vicinity of a vertex**). *Let $G = (V, E)$ be a connected weighted undirected graph and let $L \subset V$ be a subset of vertices. The vicinity of a vertex $v \in V$, denoted by $B^\star(v)$, is the set of vertices in $B(v) \cup N(B(v))$.*

We make several important observations. First, for unweighted graphs, the vicinity of a vertex v is simply a larger ball of radius $r_v + 1$; hence, vicinities have the same properties as that of balls. For weighted graphs, however, this does not hold. In particular, consider two vertices w, w' such that $d(v, w) \leq d(v, w')$; then, it may be the case that the vicinity of v may contain vertex w' but not w. To see this, let v' be some vertex in $B(v)$ such that the edge (v', w') is contained in the edge set. Then, by definition $w' \in B^\star(v)$. However, if no neighbor of w is contained in the ball of v and if $d(v, w) \geq d(v, \ell(v))$, the vertex w is not contained in the vicinity of v. Finally, note that the vicinity of a vertex v may contain an arbitrarily larger number of vertices than the ball of v. However, if the graph is μ-degree bounded, we can bound the size of vertex vicinities as: $|B^\star(v)| = \mu \cdot |B(v)|$.

It follows from the discussion above that the vicinity of a vertex u may contain vertices w without necessarily containing all vertices along the shortest path between u and w. To account for this distance "asymmetry", we will need the following notion of distance:

Definition 14 (**Candidate distance**). *Let $G = (V, E)$ be a weighted undirected graph. The candidate distance from a vertex v to another vertex $w \in B^\star(v)$, denoted as $d'_v(w)$, is defined as the cost of the least-cost path from v to w such that all intermediate vertices on this path are contained in $B(v)$; that is:*

$$d'_v(w) = \min_{x \in N(w) \cap B(v)} \{d(v, x) + \textit{weight of edge}(x, w)\}$$

Note that the candidate distance from v to w may be arbitrarily larger than the shortest distance between v and w. However, as we will show later, there are certain vertices in the vicinity of v for which the candidate distance is equal to the shortest distance.

Definition 15 (**Intersection of balls and vicinities**). *Let $G = (V, E)$ be a weighted undirected graph. The balls of a pair of vertices $s, t \in V$ are said to have a non-empty intersection if $B(s) \cap B(t) \neq \emptyset$, that is, there is a vertex $w \in V$ such that $w \in B(s)$ and $w \in B(t)$. The ball-vicinity and vicinity-vicinity intersection are defined identically.*

2.2.2 Constructing balls and vicinities of bounded size.

We now describe efficient algorithms that, given a weighted undirected graph, construct vertex balls and vicinities of bounded worst-case sizes. We will also give algorithms for efficiently computing candidate distances to vertices in the vicinity of each vertex.

Lemma 1 ([41]). *Let $G = (V, E)$ be a weighted undirected graph with n vertices and m edges. For any fixed $1 \leq \alpha \leq n$, there exists a subset of vertices L of size $\widetilde{O}(n/\alpha)$ such that for each vertex $v \in V$, we have that $|B(v)| = O(\alpha)$ with high probability. Moreover, such a set L can be computed in time $\widetilde{O}(n)$.*

We outline the proof of the above lemma. Let us first describe an algorithm to construct such a set L such that the bound on the size of set L and on the size of the ball of each vertex is bounded in expectation.

The algorithm starts by sampling each vertex in V (for inclusion in set L) independently with probability $1/\alpha$. Hence, the expected size of the set of sampled vertices is $O(n/\alpha)$. To bound the size of the ball of a vertex v, let $u_0, u_1, \ldots, u_n$ be the vertices in G sorted in non-decreasing order of distance from v; then, if u_j is the first sampled vertex in this sorted order, the size of the ball of v is $j - 1$. Since each u_i is sampled independently with probability $1/\alpha$, the size of the ball is a geometric random variable with parameter $1/\alpha$. Consequently, the expected size of the ball of v is α.

By sampling each vertex independently at random with probability $\widetilde{O}(1/\alpha)$, it follows using an argument as above and using Chernoff's bounds, that the size of the landmark set is bounded by $\widetilde{O}(n/\alpha)$ with high probability and the size of each ball is bounded by $O(\alpha)$ with high probability. It is, in fact, possible to derandomize the above algorithm such that the size of the set L and the size of the ball of each vertex is bounded deterministically:

Lemma 2 ([41, 57]). *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = 2m/n$. For any fixed $1 \leq \alpha \leq n$, there exists a subset of vertices L of size $\widetilde{O}(n/\alpha)$ such that for each vertex $v \in V$, we have that $|B(v)| = O(\alpha)$. Moreover, such a set L and the distance from each vertex v to each vertex $w \in B(v)$ can be computed in time $\widetilde{O}(m\alpha)$.*

A deterministic algorithm for constructing such a set L is as follows [41,57]. The algorithm first lets $\mathcal{N}_v$, for each vertex $v \in V$, to be the set of $O(\alpha)$ vertices of V closest to v ties broken arbitrarily. The algorithm then chooses a set L of size $O(n \log n/\alpha)$ that hits all the sets $\mathcal{N}_v$, that is, L contains at least one element from each set $\mathcal{N}_v$. To construct such a set L, the algorithm repeatedly adds vertices from V to L that hit as many unhit sets as possible until n/α sets $\mathcal{N}_v$ are unhit. The construction of set L is then completed by adding an element from each of the unhit set $\mathcal{N}_v$. Thorup and Zwick [41], using a result of Alon and Spencer [57], show that such a set L has size at most $O(n \log n/\alpha)$ and can be constructed in time $\widetilde{O}(n + n\alpha)$, given sets $\mathcal{N}_v$. For a $\mu = 2m/n$-degree bounded graph, sets $\mathcal{N}_v$ can be constructed in time $O(\alpha\mu)$ using a modified shortest path algorithm that stops once the closest $O(\alpha)$ vertices have been explored. Hence, the total construction time of the algorithm is $\widetilde{O}(m\alpha)$.

Recall that for $\mu = 2m/n$-degree bounded graphs, the size of the vicinity of any vertex is at most a factor μ larger than the size of ball of the vertex. Using this fact along with the definition of vertex vicinities and candidate distances, we get the following lemma:

Lemma 3. *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = 2m/n$. For any fixed $1 \leq \alpha \leq n$, there exists a subset of vertices L of size $\widetilde{O}(n/\alpha)$ such that for each vertex $v \in V$, we have that $|B(v)| = O(\alpha)$ and $|B^\star(v)| = O(\alpha\mu)$. It is possible to compute, in time $\widetilde{O}(m\alpha)$, such a set L, the shortest distance from each vertex v to each vertex $w \in B(v)$ and the candidate distance from each vertex v to each vertex $w \in B^\star(v)$.*

2.3 Inverse-balls and Inverse-vicinities

In this section, we extend the idea of vertex balls and vicinities to inverse-balls and inverse-vicinities. We then give efficient algorithms to construct inverse-ball and inverse-vicinities of vertices in weighted undirected graphs.

Definition 16 (Inverse-ball of a vertex). *Let $G = (V, E)$ be a connected weighted undirected graph and let $L \subset V$ be a subset of vertices. The inverse-ball of a vertex $v \in V$, denoted by $\overline{B}(v)$, is the set of vertices $w \in V$ that contain v in their ball, that is, the set of vertices $w \in V$ for which $d(w, v) < d(w, \ell(w))$.*

Definition 17 (Inverse-vicinity of a vertex). *Let $G = (V, E)$ be a connected weighted undirected graph and let $L \subset V$ be a subset of vertices. The inverse-vicinity of a vertex $v \in V$, denoted by $\overline{B}^{\star}(v)$, is the set of vertices $w \in V$ that contain v in their vicinity, that is, the set of vertices $w \in V$ for which $v \in B^{\star}(w)$.*

Constructing inverse-balls and inverse-vicinities of bounded size. The result of Lemma 3 bounds the size of vertex balls and vicinities; while this leads to bounds on *average* size of inverse-balls and inverse-vicinities, we would like a bound on the worst-case size. We now discuss how to efficiently construct inverse-balls and inverse-vicinities of bounded worst-case size. We will need the following result:

Lemma 4 ([58]). *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = 2m/n$. For any fixed $1 \leq \alpha \leq n$, there exists a subset of vertices L of expected size $8n \log n/\alpha$ such that for each vertex $v \in V$, we have that $|\overline{B}(v)| = \alpha$. Moreover, such a set L and the distance from each vertex v to each vertex $w \in \overline{B}(v)$ can be computed in time $\widetilde{O}(m\alpha)$.*

For sake of completeness, we informally describe the algorithm for constructing such a set L. Fix some $1 \leq \alpha \leq n$. The algorithm maintains two set of vertices — a set L that constitutes the final output of the algorithm and another set W that contains all vertices that have inverse-ball of size more than α. The set L is initialized to an empty set and W is initialized to the vertex set V. The algorithm runs in multiple iterations; in each iteration, it uniform randomly samples $4n/\alpha$ vertices from W, inserts them to set L; re-computes the inverse-ball of each vertex and updates W to all vertices that still contains more than α vertices in their inverse-ball. The algorithm terminates when W contains $4n/\alpha$ or fewer vertices; in this case, all vertices in W are inserted in set L. The main idea behind the proof of correctness is as follows. Clearly, by construction, each vertex has inverse-ball of size at most α. The main challenge is to bound the size of set L. It is shown in [58] that the expected number of iterations performed by the algorithm before termination is at most $2 \log n$; since $4n/\alpha$ vertices are added to L in each iteration, the size of the set L output by the algorithm is at most $8n \log n/\alpha$.

It is easy to verify that the set of vertices in the inverse-vicinity of any vertex v is given by $\overline{B}^{\star}(v) = \bigcup_{w \in N(v)} \overline{B}(w)$. Hence, once the inverse-ball for each vertex has been computed, the inverse-vicinity of any vertex v can be computed easily by iterating through each vertex $w \in N(v)$, and letting each vertex in $\overline{B}(w)$ to be in the inverse-vicinity of v. Hence, we get:

Lemma 5. *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = 2m/n$. For any fixed $1 \leq \alpha \leq n$, there exists a subset of vertices L of expected size $8n \log n/\alpha$ such that for each vertex $v \in V$, we have that $|\overline{B}(v)| = \alpha$ and $|\overline{B}^{\star}(v)| \leq \mu \cdot \alpha$. It is possible to compute, in time $\widetilde{O}(m\alpha)$, such a set L, the distance from each vertex v to each vertex $w \in \overline{B}(v)$ and the candidate distance from each vertex v to each vertex $w \in \overline{B}^{\star}(v)$.*

Lemma 5 gives an efficient way to sample a set of vertices of size $\widetilde{O}(n/\alpha)$ such that the size of the *inverse-ball* of each vertex is bounded by $O(\alpha)$; compare this with the sampling technique of Lemma 3 that gives an efficient way to sample a set of vertices of the same size such that the *ball* of each vertex is bounded by $O(\alpha)$. We emphasize that the above lemma bounds the size of set L in expectation, while the size of inverse-ball and inverse-vicinity for any vertex is bounded deterministically.

It is, in fact, possible to combine the sampling technique of Lemma 3 and Lemma 5 to construct a set L of size $\widetilde{O}(n/\alpha)$ such that the ball, the vicinity, the inverse-ball and inverse-vicinity of each vertex is of bounded size. Specifically, fix some $1 \leq \alpha \leq n$. Then, first the algorithm samples a set of vertices L_1 of size $\widetilde{O}(n/\alpha)$ using the algorithm of Lemma 3. The set L_1 is used as a seed set for the algorithm of Lemma 5. Then, another set of vertices L_2 of size $\widetilde{O}(n/\alpha)$ using the algorithm of Lemma 5. This gives us the final set of sampled vertices $L = L_1 \cup L_2$ with the following property:

Lemma 6. *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = 2m/n$. For any fixed $1 \leq \alpha \leq n$, there exists a subset of vertices L of expected size $\widetilde{O}(n/\alpha)$ such that for each vertex $v \in V$, we have that $|B(v)| = O(\alpha)$, $|\overline{B}(v)| = O(\alpha)$, $|B^{\star}(v)| = O(\alpha\mu)$ and $|\overline{B}^{\star}(v)| = O(\alpha\mu)$. It is possible to compute, in time $\widetilde{O}(m\alpha)$, such a set L, the distance from each vertex v to each vertex $w \in B(v)$ and to each vertex $w \in \overline{B}(v)$ and the candidate distance from each vertex v to each vertex $w \in B^{\star}(v)$ and to each vertex $w \in \overline{B}^{\star}(v)$.*

The notation used in the last two sections in summarized in Table 2.2.

Table 2.2: Notation on balls and vicinities used throughout the dissertation.

$\ell(v)$	Landmark of vertex v
$B(v)$	Ball of vertex v
r_v	Ball radius of vertex v
$B^\star(v)$	Vicinity of vertex v
$\overline{B}(v)$	Inverse-ball of vertex v
$\overline{B}^\star(v)$	Inverse-vicinity of vertex v

2.4 Thorup-Zwick Oracle: Upper and Lower Bounds

For general weighted undirected graphs, Thorup and Zwick [41] showed a fundamental space-stretch trade-off — for any integer $k \geq 2$, they designed an oracle of size $O(kn^{1+1/k})$ that returns distances of stretch $(2k-1)$ in $O(k)$ time; the construction time of their oracle was $\widetilde{O}(kmn^{1/k})$, in expectation. In this section, we briefly describe the construction of stretch-3 and stretch-5 distance oracles of Thorup and Zwick. We then review the follow-up research on improving the original construction of Thorup-Zwick (TZ) oracle.

2.4.1 Thorup-Zwick oracles: Upper Bounds

We start with the stretch-3 oracle and then describe the stretch-5 construction.

Stretch-3 oracle

The construction of the stretch-3 oracle starts by sampling a set L of landmark vertices using Lemma 1 for $\alpha = \sqrt{n}$. The oracle stores, for each $v \in V$:

- a hash table storing the exact distance to each vertex in L;
- the nearest vertex $\ell(v)$ and the ball radius r_v; and
- a hash table storing the exact distance to each vertex in the ball of v, that is $B(v)$.

When queried for the distance between vertices s and t, the exact distance is returned if $s \in B(t)$ or if $t \in B(s)$; else, the algorithm returns the distance $d(s,\ell(s)) + d(t,\ell(s))$.

The algorithm clearly returns a distance estimate using three hash table lookups; hence, the query time is $O(1)$. We bound the construction time, size and the stretch. Using Lemma 2, constructing the set L and computing distances from each vertex v to each vertex $w \in B(v)$ takes time $\widetilde{O}(m\sqrt{n})$, leading to a total construction time of $\widetilde{O}(m\sqrt{n})$. To bound the size, note that the size of set L is $\widetilde{O}(\sqrt{n})$ for $\alpha = \sqrt{n}$; furthermore, the size of each ball is bounded by $O(\sqrt{n})$. Hence, the size of the oracle is $\widetilde{O}(n\sqrt{n})$.

To bound the stretch, note that the exact distance is returned if $s \in B(t)$ or if $t \in B(s)$. Otherwise, the returned distance is $\delta(s,t) = d(s,\ell(s)) + d(t,\ell(s))$. Using triangle inequality, we have that $d(t,\ell(s)) \leq d(s,t) + d(s,\ell(s))$. Hence, the returned distance is $\delta(s,t) \leq 2d(s,\ell(s)) + d(s,t)$. Finally, since $t \notin B(s)$, we have that $d(s,t) \geq d(s,\ell(s))$, leading to the fact that $\delta(s,t) \leq 3d(s,t)$.

Stretch-5 oracle

The construction of the stretch-5 oracle starts by sampling a set L_1 of landmark vertices using Lemma 1 for $\alpha = n^{1/3}$; in the second step, a set L_2 of landmark vertices are sampled from the vertex set L_1, again using Lemma 1 for $\alpha = n^{1/3}$. Let $\ell_1(v)$ and $\ell_2(v)$ be the vertices in L_1 and L_2, respectively, that are closest to v. The oracle stores, for each $v \in V$:

- a hash table storing the exact distance to each vertex in L_2;
- the nearest vertices $\ell_1(v)$ and $\ell_2(v)$ and the corresponding distances;
- a hash table storing the exact distance to each vertex in the ball of v defined with respect to set L_1, that is to each vertex w such that $d(v,w) < d(v,\ell_1(v))$; and
- a hash table storing the exact distance to each vertex in set $S_v = \{w \in L_1 : d(v,w) < d(v,\ell_2(v))\}$.

When queried for the distance between vertices s and t, the exact distance is returned if $s \in B(t)$ or if $t \in B(s)$. Else, the algorithm checks if $\ell_1(t) \in S_s$; if such is the case, the algorithm returns the distance $d(s,\ell_1(t)) + d(t,\ell_1(t))$; this is easily proved to be a stretch-3 distance using arguments similar to the stretch-3 oracle. If neither of the above two conditions is satisfied, the algorithm returns the distance $d(s,\ell_2(s)) + d(\ell_2(s),t)$.

The algorithm returns a distance estimate using five hash table lookups; hence, the query time is $O(1)$. We bound the size and the stretch. To bound the size, note that the size of set L_1 is $\widetilde{O}(n^{2/3})$ and the size of set L_2 is $\widetilde{O}(n^{1/3})$ for $\alpha = n^{1/3}$. It is rather straightforward to prove that for each vertex v, $|S_v| = O(n/\alpha^2)$; hence, for the above construction, we have that $|S_v| = \widetilde{O}(n^{1/3})$ for each vertex v. Furthermore, the size of each ball is bounded by $O(n^{1/3})$. Hence, the size of the oracle is $\widetilde{O}(n^{4/3})$.

We bound the stretch for the cases when the distance is returned in the last step of the query algorithm. Note that we return the distance in the third step only if $\ell_1(t) \notin S_s$; hence, we have that $d(s,\ell_2(s)) \leq d(s,\ell_1(t))$, which by triangle inequality, gives us $d(s,\ell_2(s)) \leq d(s,t) + d(\ell_1(t),t)$. Furthermore, since $s \notin B(t)$, we get that $d(s,\ell_2(s)) \leq d(s,t) + d(s,t) = 2 \cdot d(s,t)$. The algorithm returns a distance estimate of $\delta(s,t) = d(s,\ell_2(s)) + d(\ell_2(s),t)$, which using triangle inequality gives us $\delta(s,t) \leq 2 \cdot d(s,\ell_2(s)) + d(s,t) \leq 2 \cdot 2 \cdot d(s,t) + d(s,t) = 5 \cdot d(s,t)$, as desired.

Follow-up research

Much of the early research following Thorup-Zwick result focused on improving the construction time. Roditty, Thorup and Zwick [59] derandomized the construction of Thorup and Zwick. Baswana and Sen [60] improved the construction time to $O(n^2)$ for unweighted graphs. Their result was extended to weighted graphs by Baswana and Kavitha [61]. Baswana, Gaur, Sen and Upadhyay [62] showed that it is possible to achieve subquadratic construction time for unweighted graphs at the expense of a constant additive stretch. Recently, Nilsen [63] achieved subquadratic construction time for weighted graphs with $m = o(n^2)$ edges.

The query time of the TZ oracle is not constant for super-constant stretch. Mendel and Naor [64] reduced the query time to $O(1)$ at the expense of increasing the stretch to $O(k)$ and the construction time to $\widetilde{O}(n^{2+1/k})$. It is possible to reduce the stretch (by a constant factor) [65, 66] and/or construction time [67] of their construction. Recently, Nilsen [65] reduced the query time of the TZ oracle to $O(\log k)$ using a new query algorithm that incorporates binary search within TZ oracle. Interestingly, Chechik [68] showed that it is possible to reduce the query time to an absolute constant independent of the stretch while keeping the same space-stretch trade-off.

The space-stretch trade-off of Thorup-Zwick oracle is essentially optimal, assuming the girth conjecture of Erdős. In particular, Thorup and Zwick [41] showed that any oracle for undirected graphs that returns distances of stretch less than $(2k+1)$ must have size $\Omega(n^{1+1/k})$. Their lower bound proof is information theoretic showing the existence of a dense enough graph that is incompressible. For instance, for stretch less than 3, their result implies a trivial space lower bound of $\Omega(m)$, that is, compression is impossible. Hence, a priori, it is conceivable that oracles of subquadratic size that return distances of stretch less than 3 may exist for graphs with $m = o(n^2)$ edges.

However, improving the space-stretch trade-off turned out to be a much harder problem than improving the query time and/or construction time of the TZ-oracle. Until 2010, a better trade-off was known only for special graph classes such as planar graphs [69, 70], bounded-genus and minor-free graphs [71], power-law graphs [72], random graphs [73], etc.

2.4.2 Thorup-Zwick Oracles: Lower Bounds

For general weighted undirected graphs, Thorup and Zwick [41] showed (subject to a conjecture of Erdős) that achieving (integer) stretch $(2k-1)$ requires $\Omega(kn^{1+1/k})$ space. Their proof is information-theoretic, essentially showing that for any constant stretch, there exist graphs that require storing as many bits as the number of edges in the graph. For example, proving that stretch 2 requires $\Omega(n^2)$ space uses a graph with $\Theta(n^2)$ edges; proving $\Omega(n^{3/2})$ space requirements for stretch 3 uses a graph with $\Theta(n^{3/2})$ edges.

There is no hope of this proof technique being helpful in the sparse case. In particular, for graphs with m edges, this technique will only show that achieving any constant stretch value requires $\Omega(m)$ bits, that is, compression is impossible. However, a space linear in the input size is entirely acceptable for sparse graphs, and in fact, *can* permit retrieval of shortest paths, simply by storing the original graph and running a shortest path algorithm for each query. Of course, this takes time $\widetilde{O}(m)$ per query. Thus, in the context of distance oracles, the cases of dense and sparse graphs are quite different. In the dense case the key is to *compress* the graph while ensuring that sufficient information remains to return low-stretch distances. In the sparse case, the graph need not be compressed but the trade-off with *query time* becomes critical.

Very little is known about this trade-off space for sparse graphs. First, Sommer *et al.* [74] proved in the cell-probe model that the size of stretch-s time-t distance oracles is lower bounded by $n^{1+\Omega(1/st)}$. That is, for graphs with $m = \widetilde{O}(n)$ edges, computing distances of constant stretch in constant time requires super-linear space. However, if we allow $\Omega(\log n)$ query time, their result implies a trivial lower bound of $\Omega(m)$ for any constant stretch. Pătraşcu, Roditty and Thorup [75] strengthened their result for stretch-2 oracles by proving a conditional lower bound of $\Omega(m^{5/3})$ on the size of oracles with constant query time. There are reasons to believe that it may be hard to improve the above lower bounds unconditionally [75, 76], and realistically, upper bounds seem to be the only way to make progress on the problem. A particularly compelling scenario is of $\Omega(\log n)$ query time, like ours, for which no non-trivial lower bounds are known and it is conceivable that distance oracles with constant stretch *and* linear size exist.

For stretch 3 and larger, the lower bounds for distance oracles also hold for compact routing schemes [58]; consequently, these are tight only for dense graphs. It is shown in [77–79] that any compact routing scheme with stretch less than 2 must require $\Omega(n \log n)$ memory at some vertices in the network – this bound holds even for extremely sparse graphs [78].

Chapter 3

Distance Oracles With Linear Space

For general graphs, Thorup and Zwick [41] constructed a distance oracle that, for any graph with n vertices and for any integer $k \geq 2$, is of size $O(kn^{1+1/k})$ and returns paths of stretch $2k-1$ in time $O(k)$. They also showed that their result is essentially optimal in that any oracle for stretch less than $2k+1$ must require space $\Omega(n^{1+1/k})$. However, the hard instances for the matching lower bound are rather dense graphs, with average degree $\Omega(n^{1/k})$. For instance, to prove a space lower bound of $\Omega(n^{3/2})$ for stretch 3, the proof uses a graph with $\Omega(n^{3/2})$ edges. The lower bound essentially states that there exist graphs that are incompressible: if a certain stretch is desired, then the size of the oracle is lower bounded by the number of edges in the specially-constructed dense graph. For sparse graphs, however, their result can be far from optimal since the proof only implies that any constant-time oracle must have size $\Omega(m)$.

This chapter presents oracles for stretch 3 and larger that, for sparse graphs, substantially break the space-stretch trade-off of Thorup and Zwick [41]. Our oracles, for any fixed stretch, exhibit a space-time trade-off that is not possible for the case of dense graphs: one can smoothly trade off query time to reduce the space requirements (and vice versa) for a given stretch.

3.1 Contributions and Techniques

This chapter makes three contributions. First, we show a space-time trade-off for all even stretch values greater than 3. The result is as follows:

Theorem 1. *Let G be a weighted undirected graph with n vertices, m edges with non-negative edge weights and average degree $\mu = 2m/n$. For any $1 \leq \alpha \leq n$ and for any integer $k \geq 2$, there exists a distance oracle of size $\widetilde{O}\left(m + n\alpha + n^2/\alpha^k\right)$ that returns stretch-$2k$ distances in time $O(\alpha\mu)$. The query time can be reduced to $O(\alpha)$ using an additional $\widetilde{O}(m\alpha)$ space.*

Using the same space as that of Thorup-Zwick oracle, our oracles reduce the stretch from $2k+1$ to $2k$ using query time $O(n^{1/(k+1)})$. For instance, in dense graphs, retrieving distances of stretch 5 and 7 requires space $\Theta(n^{4/3})$ and $\Theta(n^{5/4})$ respectively [41]; queries require constant time, and larger time cannot help reduce space or stretch. For the realistic case of graphs with $m = \widetilde{O}(n)$ edges, special cases of our oracles from Theorem 1 yield schemes for retrieving stretch 4 distances using space $\widetilde{O}(n^{4/3})$, and stretch 6 distances using space $\widetilde{O}(n^{5/4})$, at the expense of $\widetilde{O}(n^{1/3})$ and $\widetilde{O}(n^{1/4})$ query time, respectively. Furthermore, the query time can be reduced at the expense of larger space, providing a space-time trade-off for any even stretch $k \geq 4$.

Our second contribution is an extremely simple construction of constant-time stretch-$2k$ oracles for any positive integer k:

Theorem 2. *Let G be a weighted undirected graph with n vertices and m edges with non-negative edge weights. For any integer $k \geq 2$, there exists a distance oracle of size $\widetilde{O}\left(m^{1-\frac{2}{2k+1}} n^{\frac{4}{2k+1}}\right)$ that returns stretch-$2k$ distances in $O(k)$ time.*

For instance, the above theorem shows the existence of constant-time stretch-4 and stretch-6 oracles of size $\widetilde{O}(n^{4/5}m^{3/5})$ and $\widetilde{O}(n^{4/7}m^{5/7})$, respectively. The above theorem generalizes and simplifies the result of [75], who constructed similar oracles but using significantly more complicated techniques. Our technique, on the other hand, is extremely simple and is a natural generalization of the techniques used for the result of Theorem 1.

The oracles of Theorem 1 and of Theorem 2 can return paths corresponding to stretch-$2k$ distance estimate in constant-time per hop. In many applications, however, distance computations suffice and shortest paths are not desired. For such applications, we show the existence of distance oracles with significantly better space-time trade-off for certain stretch values. These oracles allow us to compute constant stretch distances using space linear in the size of the input graph. The result is precisely summarized in the following theorem:

Theorem 3. *Let G be a weighted undirected graph with n vertices, m edges with non-negative edge weights and average degree $\mu = 2m/n$. For any $1 \leq \alpha \leq n$ and for any integer $k \geq 2$, there exists a distance oracle of size $\widetilde{O}\left(m + (n/\alpha)^{1+1/k}\right)$ that returns stretch-$(4k-1)$ distances in time $O(\alpha\mu)$. The query time can be reduced to $O(\alpha)$ using an additional $\widetilde{O}(m\alpha)$ space.*

For instance, for graphs with $m = \widetilde{O}(n)$ edges, our last oracle is a stretch-3 oracle of size $\widetilde{O}(n)$ that answers each distance query in time $O(\sqrt{n})$.

Table 3.1: Summary of distance oracles presented in this chapter. Here, $\mu = 2m/n$ is the average degree of the graph, $k \geq 1$ is a constant and $1 \leq \alpha \leq n$ is a parameter providing trade-off between space and query time.

Stretch	Space	Query time	Construction time
$2k$	$\widetilde{O}\left(m + n\alpha + \frac{n^2}{\alpha^k}\right)$	$O(\alpha\mu)$	$\widetilde{O}(n^2)$
$2k$	$\widetilde{O}\left(m\alpha + \frac{n^2}{\alpha^k}\right)$	$O(\alpha)$	$\widetilde{O}(n^2 + m\alpha)$
$2k$	$\widetilde{O}\left(m^{1-\frac{2}{2k+1}} n^{\frac{4}{2k+1}}\right)$	$O(k)$	$\widetilde{O}(m^{1-\frac{2}{2k+1}} n^{\frac{4}{2k+1}})$
$4k-1$	$\widetilde{O}\left(m + (n/\alpha)^{1+\frac{1}{k}}\right)$	$O(\alpha\mu)$	$\widetilde{O}(mn/\alpha)$
$4k-1$	$\widetilde{O}\left(m\alpha + (n/\alpha)^{1+\frac{1}{k}}\right)$	$O(\alpha)$	$\widetilde{O}(mn/\alpha)$

A summary of our distance oracles from Theorem 1, Theorem 2 and Theorem 3 is presented in Table 3.1. Figure 3.1 provides a visual understanding of the space-time trade-off for small values of k for the special case of graphs with $m = \widetilde{O}(n)$ edges. We complement our theoretical results with extensive simulations on synthetic and real-world networks; our results suggest that our oracles not only improve the worst-case stretch of Thorup-Zwick oracles but significantly improve the average-case stretch as well. In particular, we show that our oracles are able to retrieve the exact shortest path for most source-destination pairs.

Techniques. Our stretch-$2k$ oracles of Theorem 1 are conceptually similar to the stretch-$(2k+1)$ oracles of Thorup and Zwick (TZ) [41] (see §2.4.1) with the difference that our construction is parameterized with a parameter α that provides the space-time trade-off in our oracle. That is, rather than sampling each vertex with probability $1/n^{1/k}$ for inclusion in set L as in TZ oracles, our oracle samples each vertex with probability $1/\alpha$. Our main contribution here is a query algorithm that reduces the stretch from $2k+1$ to $2k$.

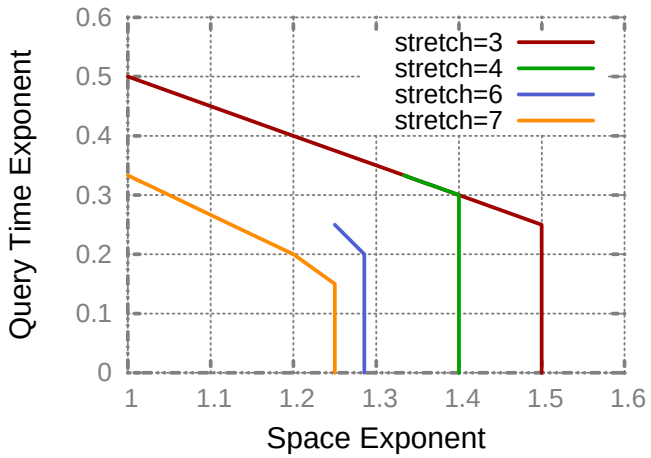


Figure 3.1: Space-time trade-off of oracles in Table 3.1 for graphs with $m = \widetilde{O}(n)$ edges. The space and the query time exponents are defined as $\log_n(S)$ and $\log_n(T)$, where S is the size and T is the query time of the oracle.

Recall from §2.4.1, that when queried for the distance between a pair of vertices s and t, TZ oracle returns a distance estimate of $(2k+1) \cdot d(s,t)$. A closer examination of the stretch proof reveals that the distance returned by the oracle is precisely $(2k-1) \cdot d(s,t) + 2 \cdot d(s,\ell(s))$; since $d(s,\ell(s)) \leq d(s,t)$, we get that the returned distance is indeed of stretch $2k+1$. Intuitively, the cases that attain worst-case stretch in their oracle are the ones for which the destination t is just outside the ball of the source s (that is, $d(s,\ell(s)) \approx d(s,t)$).

For such source-destination pairs, we exploit the idea of *ball-vicinity intersection* — upon receiving a query, we search for vertices in $B^\star(s) \cap B(t)$. Finding such vertices takes some time; but among all vertices $w \in B^\star(s) \cap B(t)$, the least-cost path $s \rightsquigarrow w \rightsquigarrow t$ is a candidate stretch-$2k$ path. We show that if this candidate path is not a stretch-$2k$ path (in fact, we show something stronger — if this path is not the shortest path between s and t), the vertices s and t must be relatively distant, giving us a lower bound of $2 \cdot d(s,\ell(s))$ on the exact distance between s and t. Using this lower bound, we get that the distance returned by the TZ-oracle is in fact $(2k-1) \cdot d(s,t) + 2 \cdot d(s,\ell(s)) \leq (2k-1) \cdot d(s,t) + d(s,t) = 2k \cdot d(s,t)$, as desired.

To further reduce the space for stretch 3 oracle, our last oracle stores the exact distances only between all pairs of landmarks. This uses significantly less space; for instance, in a graph with n nodes, storing shortest distances between every pair of n/α landmarks requires only $O(n^2/\alpha^2)$ space. We are, however, no more able to retrieve the corresponding paths. For larger stretch,

this scheme can be generalized to achieve reduced space at the expense of increased stretch: rather than storing shortest paths between landmarks, we approximate these distances using TZ oracles [41].

Related Work. A detailed comparison of our results with previously known upper bounds on distance oracles for general graphs is presented in Table 3.2. Independent of our work, Abraham and Gavoille [80] constructed constant-time stretch $(2k, 1)$ oracles for the special case of unweighted graphs; our constant-time oracles generalize their construction to weighted graphs.

Table 3.2: Upper bounds for distance oracles for general undirected graphs. The (α, β) in column 3 denotes multiplicative α and additive β stretch. $\mu = 2m/n$ denotes the *average* degree of the graph. Distance oracles with additive stretch are for unweighted graphs.

Reference	Space	Stretch	Query Time
[41]	$O(n^{3/2})$	3	$O(1)$
§3.5, [48]	$O(m + n^2/\alpha^2)$	3	$O(\alpha\mu)$
§3.5, Unpublished	$O(m\alpha + n^2/\alpha^2)$	3	$O(\alpha)$
[41]	$O(kn^{1+1/k})$	$2k-1$	$O(k)$
§3.5, [48]	$O(m + (n/\alpha)^{(1+1/k)})$	$4k-1$	$O(\alpha\mu)$
§3.5, [48]	$O(m\alpha + (n/\alpha)^{(1+1/k)})$	$4k-1$	$O(\alpha)$
§3.3, Unpublished	$O(m + n\alpha + n^2/\alpha^k)$	$2k$	$O(\alpha\mu)$
§3.3, Unpublished	$O(m\alpha + n^2/\alpha^k)$	$2k$	$O(\alpha)$
§3.4, Unpublished	$O(m^{1-\frac{2}{2k+1}} n^{\frac{4}{2k+1}})$	$2k$	$O(k)$
[76]	$O(n^{5/3})$	$(2,1)$	$O(1)$
[80]	$O(n^{1+\frac{2}{2k+1}})$	$(2k,1)$	$O(1)$
[48]	$O(n\alpha + n^2/\alpha)$	$(2,1)$	$O(\alpha)$
[48]	$O(n\alpha + (n/\alpha)^{(1+1/k)})$	$(4k-1, 2k)$	$O(\alpha)$

3.2 Reduction to Degree-bounded Graphs

We show that in the context of distance oracles, average-degree-bounded graphs are no harder than maximum-degree-bounded graphs.

Lemma 7. *Let $G = (V, E)$ be a weighted undirected graph with n vertices and m edges. For any integer $\Delta \geq 3$, it is possible to convert G into a graph H with maximum degree Δ with $m' = (1+2/(\Delta-2))m$ edges and $n' = n+2m/(\Delta-2)$ vertices such that, for any pair of vertices $u, v \in V$, the distance between any copy of u to any copy of v in H is equal to the distance of u and v in G. The conversion can be done in $O(m+n)$ time.*

Proof: The algorithm starts with a copy H of G and scan the vertices of H one by one. If a vertex $v \in V(H)$ has degree $\leq \Delta$, then we leave it as it is. Otherwise, the algorithm replaces v by a list of $\lceil \deg(v)/(\Delta-2) \rceil$ vertices, all of them connected by a new path of edges having weight 0. Each of these new vertices is assigned at most $\Delta - 2$ edges that were adjacent to v (these reassigned edges keep their original weight). The algorithm terminates when each vertex in the graph has a degree bounded by Δ.

We bound the number of vertices and edges in H. The number of vertices created by the algorithm to replace a single vertex v is $\lceil \deg(v)/(\Delta-2) \rceil$. Hence, the number of vertices in H is

$$\sum_{v \in V(G)} \lceil \deg(v)/(\Delta-2) \rceil \leq n + \sum_{v \in V(G)} \deg(v)/(\Delta-2) \leq n + 2m/(\Delta-2)$$

The number of new edges created for any vertex v is $\lceil \deg(v)/(\Delta-2) \rceil - 1$. Hence, the total number of new edges added to H are at most $2m/(\Delta-2)$.

The reduction only introduces new edges of weight 0; hence, the distance between any pair of vertices in the modified graph is the same as the distance between the corresponding vertices in the original graph. Clearly, the conversion requires time linear in the size of H, or equivalently, $O(m+n)$. □

Corollary 1. *Let $G = (V, E)$ be an undirected weighted graph with n vertices, m edges, and average degree $\mu = 2m/n$. Then one construct an equivalent graph with maximum degree $\Delta = \lceil \mu + 2 \rceil$, such that the new graph has $2n$ vertices, $m+n$ edges, and has the same distances between any pair of vertices as the distance in the original graph between the corresponding vertices. The new graph can be computed in $O(n+m)$ time.*

3.3 Space-time Trade-off for Stretch $2k$

In this section, we present the first two oracles from Table 3.1. Our oracles are similar to TZ-oracle [41] with three differences. First, the sampling probability for vertices in the landmark set is parameterized by $1 \leq \alpha \leq n$; this allows us to achieve the space-time trade-off for any fixed stretch. The second difference is that our oracle stores the input graph; this is required to compute the balls and the vicinities on the fly. Finally, the third difference is a new query algorithm that allows to reduce the stretch from $2k+1$ to $2k$.

3.3.1 Proof of Theorem 1

Constructing the distance oracle

We begin by first constructing a $\mu = 2m/n$-degree bounded graph using the result of Lemma 7 and Corollary 1. Let $G = (V, E)$ be the resulting μ-degree bounded graph. Fix some $1 \leq \alpha \leq n$ and integer $k \geq 2$. Let $L_0 = V$. We then construct a hierarchy of landmark sets $V \supseteq L_0 \supseteq L_1 \supseteq \cdots \supseteq L_k$, as in [41]; each L_i is constructed by sampling from L_{i-1} each vertex independently at random with probability $1/\alpha$. The oracle stores, for each vertex v:

- a hash table containing its neighbors $N(v)$ in G.
- for each $0 \leq i \leq k$, the vertex $\ell_i(v) \in L_i$ that is closest to v.
- a hash table containing the distance to each vertex in L_k.
- for each $0 \leq i \leq k-1$, a hash table containing the shortest distance to each vertex $w \in L_i$ that is at distance less than $d(v, \ell_{i+1}(v))$.

Query algorithm

Suppose the query asks for the distance between vertices $s, t \in V$. The algorithm (see Algorithm 1) starts by constructing a hash table containing the candidate distance from s to each vertex in $B^\star(s)$; this can be done since the oracle stores the ball of each vertex v with respect to $\ell_1(v)$ (since $L_0 = V$, all vertices at distance less than $d(v, \ell_1(v))$ from v are stored in the oracle using the last piece of information) and the edges incident on each vertex.

Algorithm 1 Query $(s, t, 2k)$: the query algorithm for computing stretch-$2k$ paths.

1: Compute candidate distance from s to each vertex in $B^\star(s)$
2: $\gamma_1 \leftarrow \infty$, $\gamma_2 \leftarrow \infty$, $\gamma_3 \leftarrow \infty$
3: $\gamma_1 \leftarrow \min_{w \in B^\star(s) \cap B(t)} \left\{ d'_s(w) + d(t, w) \right\}$
4: $\gamma_2 \leftarrow$ Query-TZ$(s, t, k+1)$
5: $\gamma_3 \leftarrow$ Query-TZ$(t, s, k+1)$
6: return $\min\{\gamma_1, \gamma_2, \gamma_3\}$

The algorithm then computes three sets of paths. First, it computes the least-cost path among paths of the form $s \rightsquigarrow w \rightsquigarrow t$ via vertices w that lie in $B^\star(s) \cap B(t)$. The second set of paths is a stretch-$(2k+1)$ path returned by the query algorithm of the TZ-oracle, when queried for distance between vertices s and t. The third set of paths is a stretch-$(2k+1)$ path returned by the query algorithm of the TZ-oracle, when queried for distance between vertices t and s (note that the order of vertices is reversed here). Finally, the algorithm returns the least-cost path among all the three sets of paths.

Analysis

We start by proving the size and query time of the distance oracle.

Claim 1. *The size of the distance oracle is* $O\left(m + n\alpha + \frac{n^2}{\alpha^k}\right)$ *and it takes time* $O(\alpha\mu)$ *to answer each distance query.*

Proof: The size of the TZ-oracle with $k+1$ levels and a sampling probability of $1/\alpha$ for each level is given by $O\left(n\alpha + \frac{n^2}{\alpha^k}\right)$; the bound on the size follows using the fact that our oracle stores the input graph in addition to the TZ-oracle. To bound the query time, we note that it takes time $O(\alpha\mu)$ to compute the candidate distance from s to each vertex in $B^\star(s)$. In addition, checking for ball-vicinity intersection takes time $O(\alpha)$; and, lines (4) and (5) clearly take time $O(k)$, leading to the desired bound on the query time. □

Lemma 8 (Ball-vicinity Intersection Lemma). *Let $G = (V, E)$ be a weighted undirected graph and let $P(s,t) = (s, x_1, x_2, \ldots, t)$ be the shortest path between vertices $s, t \in V$. Let $w = x_{i_0}$, where $i_0 = \max\{i \ : \ x_j \in B(s), \forall j < i\}$. Then, if $w \notin B(t)$, then $d(s,t) \geq r_s + r_t$.*

Proof: By definition, w is the first vertex along $P(s,t)$ that does not belong to the ball of s; hence, we have that $d(s,w) \geq d(s, \ell_1(s))$. Furthermore, since $w \notin B(t)$, we have that $d(t,w) \geq d(t, \ell_1(t))$. Finally, since w lies along the shortest path $P(s,t)$, we have that $d(s,t) = d(s,w) + d(t,w) \geq d(s, \ell_1(s)) + d(t, \ell_1(t)) = r_s + r_t$. □

Claim 2. *Let $G = (V, E)$ be a connected weighted undirected graph and let $P(s,t) = (s, x_1, x_2, \ldots, t)$ be the shortest path between vertices $s, t \in V$. Let $w = x_{i_0}$, where $i_0 = \max\{i \ : \ x_j \in B(s), \forall j < i\}$. Then, $d'_s(w) = d(s,w)$.*

Proof: The proof follows using definition of candidate distances and using the fact that $x_{i_0 - 1} \in B(s)$. □

Proof of Theorem 1. The bound on the size and query time follows from Claim 3. We prove the bound on stretch. Let $P(s,t) = (s, x_1, x_2, \ldots, t)$ be the shortest path between vertices $s, t \in V$. Let $w = x_{i_0}$, where $i_0 = \max\{i \ : \ x_j \in B(s), \forall j < i\}$. First, consider the case when $w \in B(t)$. Then, $\gamma_1 \leq d'_s(w) + d(t,w)$, which using Claim 2, gives us $\gamma_1 \leq d(s,w) + d(t,w) = d(s,t)$. Hence, the algorithm returns the shortest path via γ_1.

Consider the case when $w \notin B(t)$. From Lemma 8, we get that $d(s,t) \geq r_s + r_t$. Without loss of generality, assume that $r_s \leq r_t$. Hence, we have that $d(s,t) \geq 2r_s$; or equivalently, $r_s \leq d(s,t)/2$. Furthermore, we have using an argument similar to [41] that $\gamma_2 \leq (2k-2)d(s,t) + d(s, \ell_1(s)) + d(\ell_1(s), t)$, which using triangle inequality gives us $\gamma_2 \leq (2k-2)d(s,t) + d(s,t) + 2d(s, \ell_1(s))$. Using the fact that $r_s = d(s, \ell_1(s)) \leq d(s,t)/2$, we get that $\gamma_2 \leq (2k-2)d(s,t) + 2d(s,t) = 2k \cdot d(s,t)$. Since the algorithm returns a distance of at most $\min\{\gamma_2, \gamma_3\}$, we get that the returned distance is at most $2k \cdot d(s,t)$, as desired. □

Note that the above oracle has query time $O(\alpha\mu)$ because one needs to compute candidate distances to each vertex in the vicinity of s. However, we can precompute and store these distances within the oracle. Since each vertex has vicinity of size $O(\alpha\mu)$, this requires additional space $O(n\alpha\mu) = O(m\alpha)$. However, the query time is now reduced to $O(\alpha)$. This modification in the construction and the query algorithm leads to the second oracle of Table 3.1.

3.3.2 Implications of the average-to-max-degree-bound reduction

The construction algorithm for our oracle of Theorem 1 uses the reduction of Lemma 7 to reduce the input graph with $m = O(n\mu)$ edges into a $\mu = O(m/n)$-degree bounded graph. We show how to incorporate the reduction into the algorithm in a way that yields intuition and eases implementation.

Let G be the graph with average degree μ. The reduction implies that each node v in G which has degree $\deg(v) > \mu$ effectively "emulates" $\lceil \deg(v)/\mu \rceil$ nodes in G_μ. Now consider constructing the oracle presented in this section. While sampling nodes for the landmark set L_i, the node v is now sampled with probability $1/\alpha \cdot \lceil \deg(v)/\mu \rceil$, that is, with probability that is proportional to the degree of v. It follows from Lemma 7 that the size of $B(v)$ remains unchanged asymptotically (vertices contained in $B(v)$ and hence $B^*(v)$ may change, but not the size of these sets).

Thus, the implications of the reduction are simple: just sample each node v in the graph with probability $1/\alpha \cdot \lceil \deg(v)/\mu \rceil$ rather than probability $1/\alpha$. *In other words, rather than sampling nodes uniform-randomly, they are sampled with probability proportional to their degree.*

3.4 Constant-time Oracles for Stretch $2k$

In this section, we present the third distance oracle from Table 3.1; this facilitates the proof for Theorem 2. Our constant-time oracle for stretch-$2k$ is similar to the oracle from previous section with two differences. First, since we no longer desire a space-time trade-off (the focus is on achieving a constant-time construction), we fix a specific value for parameter α. Moreover, we use a slightly different sampling algorithm with a different sampling probability. Second, to facilitate constant-time queries, our oracle now stores the distance not only to vertices in the ball of any vertex v but to all the vertices w whose ball intersects with the vicinity of v. The main technique is to bound the size of the total number of ball-vicinity intersections by exploiting the graph sparsity.

3.4.1 Proof of Theorem 2

Constructing the distance oracle

Fix some integer $k \geq 2$. Let $L_0 = V$. Our construction first constructs a hierarchy of landmark sets $V \supseteq L_0 \supseteq L_1 \supseteq \cdots \supseteq L_k$, as in [41]. Let $L_0 = V$; L_1 is constructed by sampling from L_0 using the result of Lemma 6 from Chapter 2 (the value of α will be described soon); then, for each $2 \leq i \leq k-1$, L_i is constructed by setting L_{i-1} to be the vertex set and using the sampling technique of Lemma 6 from Chapter 2 with sampling probability $1/\alpha^2$. The oracle stores, for each vertex v:

- for each $0 \leq i \leq k$, the vertex $\ell_i(v) \in L_i$ that is closest to v.
- a hash table containing the distance to each vertex in L_k.
- for each $0 \leq i \leq k-1$, a hash table containing the shortest distance to each vertex $w \in L_i$ that is at distance less than $d(v, \ell_{i+1}(v))$.
- a hash table storing the exact distance to each vertex in the set $S_v = \{w : B(v) \cap B^*(w) \neq \emptyset\}$, that is, to each vertex w whose vicinity intersects with the ball of v.

This completes the construction of the oracle.

Query algorithm

Suppose the query asks for the distance between vertices $s, t \in V$. The algorithm (see Algorithm 2) returns the exact distance if $s \in S_t$ or if $t \in S_s$. If neither of these conditions satisfy, the algorithm computes two set of paths. The first set of paths is a stretch-$(2k+1)$ path returned by the query algorithm of the TZ-oracle, when queried for distance between vertices s and t. The second set of paths is a stretch-$(2k+1)$ path returned by the query algorithm of the TZ-oracle, when queried for distance between vertices t and s (note that the order of vertices is reversed here). Finally, the algorithm returns the least-cost path among the two sets of paths.

Algorithm 2 Query (s, t, k): the query algorithm for computing stretch-$2k$ paths in constant time.

If $t \in S_s$
    return $d(s,t)$
If $s \in S_t$
    return $d(s,t)$
$\gamma_1 \leftarrow \infty$, $\gamma_2 \leftarrow \infty$
$\gamma_1 \leftarrow$ Query-TZ$(s, t, k+1)$
$\gamma_2 \leftarrow$ Query-TZ$(t, s, k+1)$
return $\min\{\gamma_1, \gamma_2\}$

Analysis

The central idea used in the proof of Theorem 2 is to bound the size of the oracle — intuitively, if each vertex has a small size inverse-ball (or equivalently, is contained in a few balls) as guaranteed by Lemma 5, then the number of vertex pairs with ball-vicinity intersection is also small, thereby bounding $\sum_v |S_v|$. This is summarized in the following lemma:

Lemma 9. *Let $G = (V, E)$ be a weighted undirected graph with n vertices and m edges. For any fixed $1 \leq \alpha \leq n$, if the oracle is constructed as above, then:*

$$\sum_{v \in V} |S_v| \leq 2\alpha^2 m$$

Proof: For any vertex $w \in V$, let $\gamma(w)$ be the number of vertex pairs whose ball-vicinity intersection contains w; that is, $\gamma(w) = |\{(u, v) \ : \ w \in B(u) \cap B^\star(v)\}|$. Then, by definition, we get that $\sum_{v \in V} |S_v| \leq \sum_{w \in V} \gamma(w)$. Recall, using Lemma 5, each vertex w (deterministically) belongs to at most α balls and at most $\alpha \deg(w)$ vicinities. Hence, the number of ball-vicinity intersections that can occur at w is bounded by $\gamma(w) \leq \alpha^2 \deg(w)$. Hence, $\sum_{v \in V} |S_v| \leq \sum_{w \in V} \gamma(w) \leq 2\alpha^2 m$. □

Proof of Theorem 2. We first bound the size of the oracle. Storing the vertex $\ell_i(v)$ for each $1 \leq i \leq k$ and for each vertex v takes $O(nk)$ space. Also, note that $|L_1| = O(n/\alpha)$ and the ball and the inverse-ball of each vertex with respect to L_1 is bounded by size $O(\alpha)$. Furthermore, since L_{i+1} is sampled from L_i with probability $1/\alpha^2$, we get that $|L_{i+1}| = O(n/\alpha^{2i+1})$, which also gives us that the size of the ball of each vertex with respect to L_{i+1} is $O(\alpha^{2i+1})$.

Since L_i is sampled independently at random with probability $1/\alpha^{2i-1}$, this means that the number of vertices in L_i that are at distance less than $d(u, \ell_{i+1})$ from any vertex u is bounded by $O(\alpha^2)$; hence, storing the third piece of information requires space at most $O(n\alpha^2)$. Using Lemma 10, the size of the oracle is bounded by $8n^2 \log n/\alpha^{2k-1} + 2\alpha^2 m$; this expression is minimized for a specific value of the parameter used in sampling: $\alpha = 2n^{2/(2k+1)} m^{-1/(2k+1)} \log^{1/3}(n)$, leading to the desired bound.

Next, we show that the query algorithm returns a distance of at most $2k \cdot d(s,t)$. Let $P(s,t) = (s, x_1, x_2, \ldots, t)$ be the shortest path between vertices $s, t \in V$. Let $w = x_{i_0}$, where $i_0 = \max\{i \ : \ x_j \in B(s), \forall j < i\}$. First, consider the case when $w \in B(t)$. Then, it follows from the definition of S_t that s is contained in S_t and hence, the exact distance is returned.

Consider the case when $w \notin B(t)$. From Lemma 8, we get that $d(s,t) \geq r_s + r_t$. Without loss of generality, assume that $r_s \leq r_t$. Hence, we have that $d(s,t) \geq 2r_s$; or equivalently, $r_s \leq d(s,t)/2$. Furthermore, we have using an argument similar to [41] that $\gamma_2 \leq (2k-2)d(s,t) + d(s, \ell_1(s)) + d(\ell_1(s), t)$, which using triangle inequality gives us $\gamma_2 \leq (2k-2)d(s,t) + d(s,t) + 2d(s, \ell_1(s))$. Using the fact that $r_s = d(s, \ell_1(s)) \leq d(s,t)/2$, we get that $\gamma_2 \leq (2k-2)d(s,t) + 2d(s,t) = 2k \cdot d(s,t)$. Since the algorithm returns a distance of at most $\min\{\gamma_2, \gamma_3\}$, we get that the returned distance is at most $2k \cdot d(s,t)$. □

For the special case of unweighted graphs, it is possible to reduce the space requirements at the cost of a small additive stretch. In particular, Abraham and Gavoille [51] designed a constant time oracle of size $O(n^{1+2/(2k+1)})$ for unweighted graphs that, for any pair of vertices at distance d, returns a path of length at most $2k \cdot d + 1$.

3.5 Space-time Trade-off for Stretch $4k - 1$

In this section, we prove Theorem 3 by constructing distance oracles that return paths of worst-case stretch $(4k-1)$, for any positive integer k. The main technique used in the construction of this oracle is to avoid storing distances from each vertex to each landmark vertex; instead, we store distances between each pair of landmark vertices. This leads to reduced space requirements at the cost of higher stretch.

3.5.1 Proof of Theorem 3

Constructing the distance oracle

We begin by first constructing a $\mu = 2m/n$-degree bounded graph using the result of Lemma 7 and Corollary 1. Let $G = (V, E)$ be the resulting μ-degree bounded graph. Fix some $1 \leq \alpha \leq n$ and some integer $k > 0$. Our construction begins by sampling each node independently at random with probability $1/\alpha$, creating a set L of sampled nodes. We now create a complete graph G' with nodes in L as the node set and for each pair $l_1, l_2 \in L$, the weight of the edge (l_1, l_2) being the shortest path between l_1 and l_2 in G. We then construct a stretch-$(2k-1)$ TZ-oracle $\mathcal{D}'$ on G'. The oracle $\mathcal{D}$ stores $\mathcal{D}'$ as a sub-data structure. Furthermore, $\mathcal{D}$ also stores, for each node $v \in V$, its set of neighbors $N(v)$, its closest landmark node $\ell(v)$ and the ball radius r_v.

Query algorithm

Let QUERYTZ(u, v) be the query algorithm for the Thorup-Zwick scheme [41] that returns stretch-$(2k-1)$ distances between nodes s and t. The query algorithm for our distance oracle is shown in Algorithm 3.

Suppose the query asks for distance between nodes $s, t \in V$. The algorithm starts by running a shortest path algorithm that stops when the two nodes s and t have computed their vicinities and candidate distances to nodes in their vicinities. This can be done since the graph is stored in the distance oracle and requires $O(\alpha\mu)$ time using a modified version of the algorithm presented in [41]. Both s and t temporarily store this information in a hash table.

If $t \in B(s)$ or $s \in B(t)$, the algorithm returns the exact distance $d(s, t)$ from the hash table at s or t, respectively. If $t \notin B(s)$ and $s \notin B(t)$, the algorithm checks for *ball-vicinity intersection*, that is, for each node $w \in B^*(s)$, the algorithm checks if $w \in B(t)$; the least-cost path via such vertices w is a candidate stretch-$(4k-1)$ path. Another candidate stretch-$(4k-1)$ path is computed using the TZ-oracle: the algorithm queries the TZ-oracle for the distance between $\ell(s)$ and $\ell(t)$ and lets $d(s, \ell(s)) + \text{Query-TZ}(\ell(s), \ell(t), k) + d(\ell(t), t)$ be the second candidate distance. The minimum of the two candidate distances is returned. Finally, the hash tables are deleted from nodes s and t.

Algorithm 3 Query $(s, t, 4k-1)$: the query algorithm for computing stretch-$(4k-1)$ paths.

1: Compute candidate distance from s to each vertex in $B^\star(s)$
2: $\gamma_1 \leftarrow \infty$, $\gamma_2 \leftarrow \infty$
3: $\gamma_1 \leftarrow \min_{w \in B^\star(s) \cap B(t)} \left\{ d(s,w) + d'_t(w) \right\}$
4: $\gamma_2 \leftarrow d(s, \ell(s)) + \text{Query-TZ}(\ell(s), \ell(t), k) + d(\ell(t), t)$
5: return $\min\{\gamma_1, \gamma_2\}$

Analysis

Claim 3. *The size of the distance oracle is* $\widetilde{O}\left(m + (n/\alpha)^{(1+1/k)}\right)$ *and the query time is* $O(\alpha\mu)$.

Proof: Note that $E[|L|] = O(n/\alpha)$ and hence, using the results in [41], the size of the oracle $\mathcal{D}'$ is $O((n/\alpha)^{(1+1/k)})$. Storing $N(v)$ for each node v requires an additional $O(n\mu)$ space; storing $\ell(v)$ and r_v require an additional $O(1)$ space. Hence, the size of the oracle is $O(n\mu + (n/\alpha)^{(1+1/k)})$.

Regarding the query time, note that the query algorithms is very similar to the query algorithm for our oracles of §3.3 with the only difference in lines (4) and (5). Indeed, checking ball-vicinity intersection is still the bottleneck in terms of query time; hence, using arguments similar to those in §3.3, we get that the query time for the query algorithm is $O(\alpha\mu)$. □

Proof of Theorem 3. The bound on the size and the query time follows from Claim 3. We prove the bound on stretch. Let $P(s,t) = (s, x_1, x_2, \ldots, t)$ be the shortest path between vertices $s, t \in V$. Let $w = x_{i_0}$, where $i_0 = \max\{i \ : \ x_j \in B(s), \forall j < i\}$. Then, if $w \in B(t)$, we get that $\gamma_1 = d'(s,w) + d(w,t)$, which using Claim 2, gives us $\gamma_1 = d(s,w) + d(w,t)$. Since w lies along the shortest path between s and t, we get that $\gamma_1 = d(s,t)$.

Consider the case when $w \notin B(t)$. Then, using Lemma 8, we have that $d(s,t) \geq r_s + r_t$. Furthermore, $\gamma_2 = d(s,\ell(s)) + \textsc{QueryTZ}(s,t) + d(\ell(t),t)$. Since QueryTZ(s, t) returns a stretch-$(2k-1)$ distance, we have that $\gamma_2 \leq d(s,\ell(s)) + (2k-1)d(\ell(s),\ell(t)) + d(\ell(t),t)$. By the triangle inequality, $d(\ell(s),\ell(t)) \leq d(\ell(s),s) + d(s,t) + d(t,\ell(t))$. Hence, $\gamma_2 \leq 2k \cdot d(s,\ell(s)) + (2k-1)d(s,t) + 2k \cdot d(t,\ell(t))$. Since $d(s,\ell(s)) = r_s$ and $d(t,\ell(t)) = r_t$, we get $\gamma_2 \leq 2k \cdot r_s + (2k-1)d(s,t) + 2k \cdot r_t = 2k(r_s + r_t) + (2k-1)d(s,t)$, which using the lower bound on the distance gives us $\gamma_2 \leq 2k \cdot d(s,t) + (2k-1)d(s,t) = (4k-1) \cdot d(s,t)$, which we set out to prove. □

3.6 Evaluation

In this section, we evaluate the performance of our stretch 3 scheme of Theorem 3 on large-scale synthetic and real-world network topologies. We first present our methodology, followed by a summary of the evaluation results and conclude with a detailed discussion on the results.

3.6.1 An optimization

Although the worst-case stretch for our distance oracle is $4k-1$, we can apply simple heuristics to improve the stretch in practice. Recall that the worst-case stretch in our oracles occurs for vertex pairs s, t for which $B^\star(s) \cap B(t) = \emptyset$; the query may return a path, for instance, $s \rightsquigarrow \ell(s) \rightsquigarrow \ell(t) \rightsquigarrow t$ that is of stretch 3. The main observation is that for such vertex pairs, there may exist a $w \in B^\star(s)$ for which the length of the path $s \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow \ell(t) \rightsquigarrow t$ is less than the path $s \rightsquigarrow \ell(s) \rightsquigarrow \ell(t) \rightsquigarrow t$. The query can then be answered by the oracle as the minimum of the distances retrieved by checking all $w \in B^\star(s)$ (see §3.6 for implementation details). Since checking the length of the paths $s \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow \ell(t) \rightsquigarrow t$ for all $w \in B^\star(s)$ takes (asymptotically) the same time as checking the ball-vicinity intersection, the heuristic does not increase the query time, with potential improvements in stretch of retrieved paths. Indeed, this optimization not only improves the average stretch but also increases the number of vertex pairs for which our oracle returns the exact shortest paths.

3.6.2 Methodology

We evaluate four schemes: the stretch-3 TZ scheme with landmarks selected uniform randomly, the stretch-3 TZ scheme with landmarks selected using our scheme, and two version of our stretch 3 scheme: the stretch-3 scheme (for $k = 1$) from the last oracle with $\alpha = \sqrt{n}$ with and without the optimization discussed above. For the TZ scheme, we sampled each vertex (for set L) with probability $\sqrt{\log n/n}$. For our stretch 3 scheme, each vertex was sampled with probability $\sqrt{n \log n} \times \deg(v)/\log^2 n$. All the constants in the big-O notation were set to be 1. All these schemes were evaluated using static simulator, assuming static graph topologies, which we describe next.

We present evaluation results for three topologies. (1) $G(n,m)$ random graphs, *i.e.*, $n = 16384$ nodes with m uniform-random edges, with m set so that the average degree is 6, (2) geometric random graphs with $n = 16384$ nodes with average degree 6, and (3) a $33,014$ node AS-level map of the Internet (referred to as the Internet graph in this section) [46].

For $G(n,m)$ graphs and the Internet graph, link weights are 1; for geometric random graphs, a link's weight is the Euclidean distance between the position of its two vertices. For $G(n,m)$ graphs and for geometric random graphs, we generated 10 different topologies with the same parameters and our results are the average of evaluations of these topologies. For geometric random graphs, we sampled a set of "source" vertices and evaluated the performance of the schemes from these sources to all the destinations. We found [81] that sampling $1/4$ of the nodes as sources provided accurate results.

3.6.3 Results and Discussions

Fig. 3.2 shows the performance of the four schemes for various graph topologies (TZ is the original TZ scheme, TZ* scheme is discussed below in more detail). The most notable result of this evaluation is that our stretch 3 scheme allows retrieval of exact shortest paths for nearly all source-destination pairs: more than 98.4% in the $G(n,m)$ graph, and more than 99.9% in the Internet graph. Though $G(n,m)$ graphs and the Internet graph have highly different structures, these graphs have a common feature: for nearly all source-destination pairs, the two vicinities intersect, thus providing a shortest path. In the $G(n,m)$ graph (in which 96.2% source-destination pairs have intersecting vicinities), this occurs since, with high probability, the diameter of the graph is roughly at most twice the vicinity radius. In the Internet graph (in which 96.8% source-destination pairs have intersecting vicinities), vicinity intersection likely occurs at the "core" networks of the Internet. Since TZ scheme does not exploit the vicinity intersection, its performance is significantly worse than our schemes (only 34.4% of the source-destination pairs retrieved shortest paths).

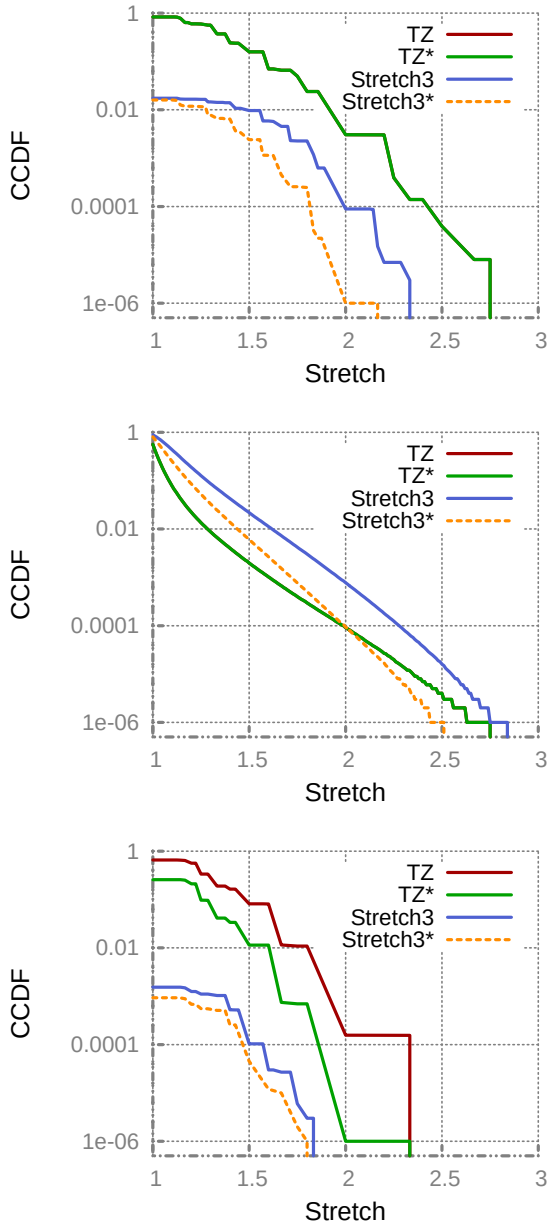


Figure 3.2: Comparison of the stretch for our stretch-3 oracle and the stretch-3 oracle of Thorup and Zwick [41] for $G(n,m)$ random graph (top), geometric random graph (middle) and AS-level internet map (bottom). As described in §3.6.2, TZ* is the scheme which uses the set of landmarks constructed by the algorithm of §3.4; Stretch-3* is the scheme which uses optimization discussed in §3.3.

The surprising difference between the performance of the two schemes may be due to the difference in which these schemes construct the landmark set L. We evaluated a modified version of the TZ scheme that uses the same set L as used by our schemes (see TZ* in Fig. 3.2). Although this improves the performance of the TZ scheme (74.2% of the source-destination pairs now retrieve shortest paths), it is still much worse than the our stretch 3 scheme. We, hence, believe that the high performance of our schemes is indeed due to the vicinity intersection idea.

For geometric random graphs, our stretch 3 scheme allows retrieval of shortest paths only for 19.2% of the source-destination pairs in comparison to 42.9% for the TZ scheme; indeed, only 4.8% of the source-destination pairs have intersecting vicinities. However, while the TZ-scheme performs better than our stretch 3 scheme on an average for the geometric random graph, the worst-case stretch for the TZ-scheme is consistently worse than our stretch 3 scheme. We believe that this is due to the P&S optimization, that allows many source-destination pairs to retrieve shorter paths due to short-cutting.

3.7 Summary

In this chapter, we have presented distance oracles for stretch 3 and larger. Our oracles have three interesting properties. First, our oracles significantly improve the space-stretch trade-off of Thorup-Zwick (TZ) oracles by exploiting graph sparsity. In particular, using the same space as that of TZ oracles, our oracles return paths with an improved worst-case stretch; on the other hand, for the same stretch as that of TZ-oracles, our oracles can reduce the space all the way down to linear space. Second, our oracles exhibit a more general space-time trade-off for any fixed stretch — this provides a separation between the dense and the sparse cases for the distance oracle problem. Finally, our oracles not only achieve an improved worst-case bound on the stretch, they perform extremely well over real-world network topologies, retrieving the exact shortest path for most source-destination pairs.

Chapter 4

Distance Oracles for Stretch 2

Chapter 3 presented distance oracles that return distances of stretch 3 and larger using space strictly less than the Thorup-Zwick oracle. This chapter explores the other dimension — reducing the stretch of Thorup-Zwick oracles. In particular, recall from §2.4.2 that the lower bound of Thorup and Zwick [41] states that any distance oracle for stretch less than 3 requires space $\Omega(n^2)$. However, the hard cases that constitute the lower bound are extremely dense graphs, those with $\Theta(n^2)$ edges. This raises a natural question: Is it possible to construct oracles of $o(n^2)$ size that return distances of stretch less than 3 for graphs with $m = o(n^2)$ edges? This chapter answers this question in affirmative, constructing oracles that achieve a space time trade-off of $S \times T = O(n^2)$ for sparse graphs.

4.1 Contributions and Techniques

This chapter makes three major contributions. First, we show a space-time trade-off for distance oracles for stretch 2. The result is as follows:

Theorem 4. *Let G be a weighted undirected graph with n vertices, m edges and average degree $\mu = 2m/n$. Then, for any fixed $1 \leq \alpha \leq n$, there exists a distance oracle of size $\widetilde{O}(m + n^2/\alpha)$ that returns stretch-2 distances in time $O(\alpha\mu)$. The query time can be reduced to $O(\alpha)$ using an additional $O(m\alpha)$ space.*

We make two observations. First, the above oracle achieves a smooth space-time trade-off similar to our oracles of Chapter 3 and, for any fixed space, achieves lower stretch at the cost of higher query time when compared to oracles of Chapter 3. Second, for graphs with $m = \widetilde{O}(n^{1+\varepsilon})$ edges, the above oracle can compute stretch-2 distances using linear space and sub-linear query time.

Pătraşcu and Roditty [76], independent to our work, constructed a stretch-2 oracle of size $O(n^{4/3}m^{1/3})$; interestingly, their oracle returns stretch-2 distances in constant time. However, their construction uses substantially more complex techniques than oracles for dense graphs and oracles with super-constant query time. For weighted graphs, their algorithm for constructing the oracle is particularly complex — it first samples a set of edges A and a set of vertices B (each with a different probability); it then constructs partial shortest path trees around each vertex in B with a stopping criteria that depends on edges in set A. Finally, the algorithm constructs partial shortest path trees around each remaining vertex with a new stopping criteria that depends on edges in set A, vertices in set B and the edges explored while constructing partial shortest path trees around vertices in set B.

Our second contribution is a new constant-time stretch-2 oracle for *weighted* graphs that admits significantly simpler construction and proofs. Our algorithm requires sampling a set A of vertices and constructing partial shortest path trees around each vertex using a single stopping criteria that depends only on vertices in set A:

Theorem 5. *Given a weighted undirected graph with n vertices and m edges with non-negative edge weights, there exists a distance oracle of expected size $8n^{4/3}m^{1/3}\log^{2/3} n$ that returns a stretch-2 distance in constant time.*

These results are relevant to the area of *compact routing* [58], which has applied distance oracle techniques to routing in networks, where routers should require limited state yet forward packets along short paths. Recent work has shown how these compact routing tables can be constructed using distributed protocols [82–84], and as discussed above, the networks in which these protocols might be applied are sparse. We describe how our stretch-2 scheme can be implemented in a distributed way similar to [83] – with the addition of a surprisingly lightweight end-to-end exchange of less than 5 KB (at most 4 packets) and a small amount of processing in order to set up a new end-to-end connection.

We complement our theoretical results with extensive simulations on empirical networks. Interestingly, we find that in the Internet AS-level topology, our stretch-2 scheme finds *shortest* paths for 99.98% of the source-destination pairs – compared with 34.4% using [41].

Techniques. Our distance oracle for stretch 2 is conceptually similar to the stretch 3 distance oracle of Thorup and Zwick [41]. For a given graph, they construct a set of nodes, known as *landmarks*, such that each node has a landmark in its ball. The distance oracle stores, for each node, the distance to each node in its ball and to its closest landmark; the landmarks store distances to all nodes in the graph. When queried for distance between nodes u and v, the query algorithm checks if v is in ball of u. If it is, then the exact distance is returned using information stored in the distance oracle; if not, the distance $d(u,\ell(u)) + d(\ell(u),v)$ is returned, which is at most stretch 3.

Intuitively, the cases that attain worst-case stretch in their distance oracle are the ones for which the destination v is just outside the ball of the source u. For such source-destination pairs, we exploit the idea of *ball-vicinity intersection*. Upon receiving a query, we search for nodes in $B(u) \cap B^\star(v)$. Finding such nodes takes some time; but if any such node w exists, we can return the distance $d(u,w) + d(w,v)$ using information stored in the distance oracle. If $B(u) \cap B^\star(v) = \emptyset$, the nodes must be relatively distant, giving us a lower bound on the exact distance between u and v. Using this lower bound, we show that a path via the landmark node has stretch 2. We need to store the vicinities of the nodes for some of our distance oracles; but if the graph is sparse, we show that this does not increase the space requirement significantly.

Our construction of constant-time stretch-2 oracles uses the notion of balls used in [41] and of vicinities used in [48,51,52]. We say that a pair of vertices have a ball-vicinity intersection if the ball of one vertex has a non-empty intersection with the vicinity of the other vertex. To bound the space requirements, we exploit graph sparsity to prove a non-trivial upper bound on the number of vertex pairs with ball-vicinity intersection; this requires a special ball construction algorithm previously used in design of compact routing schemes [58]. Furthermore, to bound the stretch, we show that for any pair of vertices with non-intersecting ball-vicinity, a stretch-2 distance can be computed by storing a small amount of information per vertex in the graph.

Table 4.1: Upper bounds for distance oracles for general undirected graphs. The (α, β) in column 3 denotes multiplicative α and additive β approximation ratio. μ denotes the *average* degree of the graph. Distance oracles with additive stretch are for unweighted graphs.

Reference	Space	Stretch	Query Time	Remarks
[41]	$O(n^2)$	2	$O(1)$	$\star$
[76]	$O(n^{5/3}\mu^{1/3})$	2	$O(1)$	
Unpublished	$O(m + n^2/\alpha)$	2	$O(\alpha\mu)$	$1 \leq \alpha \leq n$
[48]	$O(m\alpha + n^2/\alpha)$	2	$O(\alpha)$	$1 \leq \alpha \leq n$
[41]	$O(n^{3/2})$	$(3,0)$	$O(1)$	
[76]	$O(n^{5/3})$	$(2,1)$	$O(1)$	
[48]	$O(n\alpha + n^2/\alpha)$	$(2,1)$	$O(\alpha)$	$1 \leq \alpha \leq n$

Related Work. A detailed comparison of our results with previously known upper bounds on distance oracles for general graphs is presented in Table 4.1. Very recently, Pătraşcu and Roditty [76] obtained a distance oracle that returns stretch 2 paths in constant time with $O(\mu^{1/3}n^{5/3})$ space. These queries are faster than our stretch-2 scheme, but the distance oracle has larger size for $\alpha > (n/\mu)^{1/3}$ (recall, $1 \leq \alpha \leq n$ is the parameter in our distance oracle that provides the desired space/query-time trade-off). For general sparse graphs, no other results are known.

For unweighted graphs, the only known distance oracle of size $o(n^2)$ with stretch 2 is again due to the recent result of Pătraşcu and Roditty [76]. Their distance oracle requires space $O(n^{5/3})$ and constant query time. As earlier, our distance oracle requires less space but higher query time.

In terms of upper bounds for compact routing schemes, we note that the only known results are by Thorup and Zwick [58] for stretch 3 and larger. No compact routing schemes with worst-case stretch less than 3 are known. Although we believe that it may be possible to design compact routing schemes for the distance oracle of Pătraşcu and Roditty [76], it is not clear whether this can be done in a distributed fashion. Our compact routing schemes, on the other hand, can be constructed in a distributed fashion and have worst-case stretch 2.

4.2 Space-time Trade-off

In this section, we prove Theorem 4: for any fixed $1 \leq \alpha \leq n$, an oracle of size $\widetilde{O}(m + n^2/\alpha)$ that returns stretch-2 distances in time $O(\alpha\mu)$. We then show how to further reduce the query time to $O(\alpha)$ using an additional $O(m\alpha)$ space. Our oracles also allow retrieving paths in constant time per hop.

Constructing the distance oracle

Our construction of the oracle begins by first constructing a $\mu = 2m/n$-degree bounded graph using the result of Lemma 7 and Corollary 1. Let $G = (V, E)$ be the resulting μ-degree bounded graph. Fix some $1 \leq \alpha \leq n$. The algorithm then samples each vertex independently at random with probability $1/\alpha$, creating a set L of sampled "landmark" vertices. The distance oracle stores:

- For each node $v \in V$, a hash table containing its neighbors $N(v)$.
- For each node $v \in V$, a hash table containing the shortest distance to every vertex in L.
- For each node $v \in V$, $\ell(v)$ and the "ball radius" $r_v = d(v, \ell(v))$.

This completes the construction of the distance oracle.

Query algorithm

Suppose the query asks for the distance between vertices $s, t \in V$. The algorithm (see Algorithm 4) starts by constructing a hash table containing the candidate distance from s to each vertex in $B^\star(s)$; this can be done since the input graph is stored in the oracle in the form of the edges incident on each vertex. The algorithm does the same for vertex t and computes the exact distance from t to each vertex in the ball of t.

The algorithm then computes three sets of paths. First, it computes the least-cost path among paths of the form $s \rightsquigarrow w \rightsquigarrow t$ via vertices w that lie in $B^\star(s) \cap B(t)$. The second set of paths is the path from s to t via the landmark vertex of s and the third set of paths is the path from t to s via the landmark vertex of t. Finally, the algorithm returns the least-cost path among all the three sets of paths.

Algorithm 4 Query $(s,t,2)$: the query algorithm for computing stretch-2 paths.

1: Compute candidate distance from s to each vertex in $B^\star(s)$
2: Compute shortest distance from t to each vertex in $B(t)$
3: $\gamma_1 \leftarrow \infty$, $\gamma_2 \leftarrow \infty$, $\gamma_3 \leftarrow \infty$
4: $\gamma_1 \leftarrow \min_{w \in B^\star(s) \cap B(t)} \left\{ d'_s(w) + d(t,w) \right\}$
5: $\gamma_2 \leftarrow d(s,\ell(s)) + d(\ell(s),t)$
6: $\gamma_3 \leftarrow d(t,\ell(t)) + d(\ell(t),s)$
7: return $\min\{\gamma_1,\gamma_2,\gamma_3\}$

Analysis

We start by proving the size and query time of the distance oracle.

Claim 4. *The size of the distance oracle is $O(m+n^2/\alpha)$ and it takes time $O(\alpha\mu)$ to answer each distance query.*

Proof: Storing the list of neighbors for each vertex requires space $O(n\mu) = O(m)$. Note that $E[|L|] = n/\alpha$, and hence, storing shortest distances from each vertex in G to each vertex in L requires $O(n^2/\alpha)$ space. Hence, the size of the oracle is $O(m+n^2/\alpha)$.

To bound the query time, we note that it takes time $O(\alpha\mu)$ to compute the shortest distance from s to each vertex in $B(s)$; it follows from the definition of the candidate distance that the candidate distance from the vertex s to each vertex in $B^\star(s)$ can be computed in time $O(\alpha\mu)$. Regarding line (4), we note that checking for ball-vicinity intersection takes time $O(\alpha)$; lines (5) and (6) clearly take time $O(1)$, leading to the desired bound on the query time. □

Proof of Theorem 4. The bound on the size and query time follows from Claim 4. We prove the bound on stretch. Let $P(s,t) = (s,x_1,x_2,\ldots,t)$ be the shortest path between vertices $s,t \in V$. Let $w = x_{i_0}$, where $i_0 = \max\{i : x_j \in B(s), \forall j < i\}$. First, consider the case when $w \in B(t)$. Then, $\gamma_1 \leq d'_s(w) + d(t,w)$, which using Claim 2, gives us $\gamma_1 \leq d(s,w) + d(t,w) = d(s,t)$. Hence, the algorithm returns the shortest path via γ_1.

Consider the case when $w \notin B(t)$. From Lemma 8, we get that $d(s,t) \geq r_s + r_t$. Without loss of generality, assume that $r_s \leq r_t$. Hence, we have that $d(s,t) \geq 2r_s$; or equivalently, $r_s \leq d(s,t)/2$. Furthermore, we have using triangle inequality that $\gamma_2 = d(s,\ell(s)) + d(\ell(s),t) \leq d(s,t) + 2d(s,\ell(s))$.

Using the fact that $r_s = d(s, \ell(s)) \leq d(s,t)/2$, we get that $\gamma_2 \leq 2d(s,t)$. Since the algorithm returns a distance of at most $\min\{\gamma_2, \gamma_3\}$, we get that the returned distance is at most $2d(s,t)$, as desired. □

Note that the above oracle has query time $O(\alpha\mu)$ because one needs to compute candidate distances to each vertex in the vicinity of s. However, we can precompute and store these distances within the oracle. Since each vertex has vicinity of size $O(\alpha\mu)$, this requires additional space $O(n\alpha\mu) = O(m\alpha)$. However, the query time is now reduced to $O(\alpha)$.

4.3 A Simple Constant-time Oracle

Constructing the distance oracle

Our construction of the oracle begins by creating a set L of vertices using the result of Lemma 5 (the value of α will be specified later). The oracle stores, for each $v \in V$:

- a hash table storing the exact distance to each vertex in L;
- the nearest vertex $\ell(v)$ and the ball radius r_v; and
- a hash table storing the exact distance to each vertex in the set

$$S_v = \{w \ : \ B(v) \cap B^\star(w) \neq \emptyset\}$$

  that is, to each vertex w whose vicinity intersects with the ball of v.

Query algorithm

When queried for the distance between vertices $s, t \in V$, the algorithm returns the exact distance if $s \in S_t$ or if $t \in S_s$. Else, the algorithm returns $d(s, \ell(s)) + d(t, \ell(s))$ if $r_s \leq r_t$ and $d(t, \ell(t)) + d(s, \ell(t))$ otherwise.

4.3.1 Analysis

The proof uses two ideas. The first is used to bound the size of the oracle — intuitively, if each vertex has a small size inverse-ball (or equivalently, is contained in a few balls) as guaranteed by Lemma 5, then the number of vertex pairs with ball-vicinity intersection is also small, thereby bounding $\sum_v |S_v|$. The second is used to bound the stretch — any pair of vertices s, t with non-intersecting ball-vicinity must be rather far away and either the path $s \rightsquigarrow \ell(s) \rightsquigarrow t$ or the path $t \rightsquigarrow \ell(t) \rightsquigarrow s$ must be a stretch-2 path.

Lemma 10. *Let $G = (V, E)$ be a weighted undirected graph with n vertices and m edges. For any fixed $1 \leq \alpha \leq n$, if the oracle is constructed as above, then: $\sum_{v \in V} |S_v| \leq 2\alpha^2 m$.*

Proof: For any vertex $w \in V$, let $\gamma(w)$ be the number of vertex pairs whose ball-vicinity intersection contains w; that is, $\gamma(w) = |\{(u, v) \ : \ w \in B(u) \cap B^\star(v)\}|$. Then, by definition, we get that $\sum_{v \in V} |S_v| \leq \sum_{w \in V} \gamma(w)$. Recall, using Lemma 5, each vertex w (deterministically) belongs to at most α balls and at most $\alpha \deg(w)$ vicinities. Hence, the number of ball-vicinity intersections that can occur at w is bounded by $\gamma(w) \leq \alpha^2 \deg(w)$. Hence, $\sum_{v \in V} |S_v| \leq \sum_{w \in V} \gamma(w) \leq 2\alpha^2 m$. □

Proof of Theorem 5. We first bound the size of the oracle. Using Lemma 5, the expected size of set L is $8n \log n/\alpha$; and, using Lemma 10, the size of set $\sum_{v \in V} |S_v|$ is bounded by $2\alpha^2 m$. Hence, the oracle's size is bounded by $8n^2 \log n/\alpha + 2\alpha^2 m$; this expression is minimized for $\alpha = 2n^{2/3} m^{-1/3} \log^{1/3}(n)$, leading to the desired bound.

Next, we show that the query algorithm returns a distance of at most $2d(s, t)$. If $B(s) \cap B^\star(t) \neq \emptyset$, the algorithm returns the exact distance. For the case when $B(s) \cap B^\star(t) = \emptyset$, assume, without loss of generality, that $r_s \leq r_t$. Then, using Lemma 8, $d(s, t) \geq 2r_s$; or equivalently, $2r_s \leq d(s, t)$. The distance returned by the query algorithm is $d(s, \ell(s)) + d(t, \ell(s))$, which using triangle inequality, is at most $2d(s, \ell(s)) + d(s, t) = 2r_s + d(s, t) \leq 2d(s, t)$, as claimed. □

For the special case of unweighted graphs, it is possible to reduce the space requirements at the cost of a small additive stretch. Using ideas similar to above, we get a constant-time oracle of size $O(n^{5/3})$ for unweighted graphs that, for any pair of vertices at distance d, returns a path of length at most $2d + 1$.

4.4 Unweighted Graphs

In this section, we show that the space-time trade-off of our distance oracles from §4.2 can be further improved at the cost of a small additive stretch. In particular, let G be an unweighted graph and let u, v be a pair of nodes at distance d; then, for any fixed $1 \leq \alpha \leq n$, we design:

- a distance oracle of size $O(n\alpha + n^2/\alpha)$ that returns distances of at most $2d + 1$ in time $O(\alpha)$, and
- a distance oracle of size $O\left(n\alpha + (n/\alpha)^{(1+1/k)}\right)$ that returns distances of at most $(4k - 1)d + 2k$ in time $O(\alpha)$

The results can be generalized to weighted graphs without any increase in space or query time. The main observation that allows us to design these oracles is captured in the following lemma, which presents a lower bound on the distance between the source and the destination when the query algorithm checks for ball-ball intersection rather than ball-vicinity intersection as in Lemma 8:

Lemma 11 (**Ball-ball intersection**). *For any pair of nodes $u, v \in V$, let w_{uv} be the weight of the heaviest edge along the shortest path between u and v. If $B(u) \cap B(v) = \emptyset$, the distance between u and v is lower bounded as $d(u, v) \geq r_u + r_v - w_{uv}$.*

Proof: Assume that $B(u) \cap B(v) = \emptyset$ and let $P = (u, x_1, x_2, \ldots, v)$ be the shortest path between u and v. Let $i_0 = \max\{i | x_i \in P \cap B(u)\}$, $w = x_{i_0}$ and $w' = x_{i_0+1}$. By definition, $w' \notin B(u)$ and hence, $d(u, w') \geq r_u$. Furthermore, since w and w' are neighbors and $w \in B(u)$, we have that $d(u, w) \geq r_u - w_{uv}$. Furthermore, since $B(u) \cap B(v) = \emptyset$, $w \notin B(v)$ leading to the fact that $d(v, w) \geq r_v$; since w is on the shortest path between u and v, we have that $d(u, v) = d(u, w) + d(v, w) \geq r_u + r_v - w_{uv}$. □

Lemma 11 suggests that if the query algorithms from the previous section were to check for ball-ball intersection rather than ball-vicinity intersection, the loss in stretch can be bounded by a constant factor that depends on the heaviest weight along the shortest path between the source and the destination.

In contrast to the oracles of the previous section, performing ball-ball intersection neither requires storing the vicinities nor computing them on the fly; query is now performed only on the balls of each node leading to improvements in space and/or query time.

Constructing the distance oracles

Let $G = (V, E)$ be a Δ-degree bounded graph. The construction begins by sampling each node independently at random with probability $1/\alpha$, creating a set L of sampled "landmark" nodes.

Distance oracle for additive stretch 1. The distance oracle stores, for each node $v \in L$, a hash table containing the shortest distance to every other node in G and for each node $v \in V \backslash L$, distances to nodes in its ball, its landmark node $\ell(v)$ and the "ball radius" $r_v = d(v, \ell(v))$.

To bound the size of the distance oracle, we note that we have $O(n/\alpha)$ landmarks, in expectation, requiring $O(n^2/\alpha)$ space to store distances to each other node in the graph. Furthermore, each node has $O(\alpha)$ nodes in its ball and hence, storing distances to these nodes require $O(n\alpha)$ space; storing $\ell(v)$ and r_v for each node v requires an additional $O(1)$ space. Hence, the total space requirements are $O(n\alpha + n^2/\alpha)$, in expectation.

Distance oracle for additive stretch $2k$. First, a complete graph on nodes in L is computed, where weight of each edge is equal to the shortest distance between the two nodes. The distance oracle $\mathcal{D}$ stores, as a sub-data structure, the Thorup-Zwick distance oracle $\mathcal{D}'$ that returns stretch $(2k-1)$ distances for the complete graph over nodes in L. In addition, $\mathcal{D}$ stores, for each node $v \in V \backslash L$, distances to nodes in its ball, its landmark node $\ell(v)$ and the "ball radius" $r_v = d(v, \ell(v))$.

Recall that the expected number of nodes in the landmark set is $O(n/\alpha)$ and hence, size of the sub-data structure $\mathcal{D}'$ is $O((n/\alpha)^{(1+1/k)})$. Furthermore, since the size of ball for each node is $O(\alpha)$, the additional space required is $O(\alpha)$ for each node. The overall size of the distance oracle is, hence, $O(n\alpha + (n/\alpha)^{(1+1/k)})$, in expectation.

Query algorithms and analysis

The query algorithms for the above distance oracles are similar to their respective query algorithms from the previous section with the only change that it performs ball-ball intersection check rather than ball-vicinity intersection check. Regarding the query time, we note that since balls for each node are stored within the distance oracles, checking for ball-ball intersection requires $O(\alpha)$ time, leading to the claimed bound on the query time.

We prove the stretch bound for the first distance oracle; for larger stretch, the proof follows using straightforward modifications.

Theorem 6. *For any two nodes $u, v \in V$ at distance d, let w_{uv} be the weight of the heaviest edge along the shortest path between u and v. Then, the query algorithm returns a distance of at most $2d + w_{uv}$.*

Proof: For the case when $d(u,v) < r_u + r_v - w_{uv}$, using Lemma 11, it is easy to show that the query algorithm returns the exact distance between u and v.

Consider the case when $d(u,v) \geq r_u + r_v - w_{uv}$ and without loss of generality, assume that $r_u \leq r_v$. Then, the condition implies that $d(u,v) \geq 2 \cdot r_u - w_{uv}$. In such a case, the distance returned by the query algorithm is $d(u, \ell(u)) + d(\ell(u), v)$. By the triangle inequality, we have that $d(\ell(u), v) \leq d(\ell(u), u) + d(u,v)$. Hence, $\delta(u,v) \leq 2 \cdot d(u, \ell(u)) + d(u,v)$. Since $d(u, \ell(u)) \leq r_u$, we get $\delta(u,v) \leq 2 \cdot r_u + d(u,v)$. Using the lower bound of $2 \cdot r_u - w_{uv}$ on the distance between u and v, we get the desired bound of $2d(u,v) + w_{uv}$ on stretch. □

Similarly, one can prove that for any pair of nodes u, v at distance d, the second oracle returns a distance of at most $(4k-1)d + 2k \cdot w_{uv}$, where w_{uv} is the weight of the heaviest weight along the shortest path between u and v.

4.5 Application: Compact Routing

Work on compact routing has applied the traditional results from approximate distance oracles [41] to network routing problems [58] in order to route the packets along short paths while using little memory at routers. Thorup and Zwick [58] designed compact routing schemes for their distance oracles. Their scheme requires $\widetilde{O}(\sqrt{n})$ memory at each node in the network and routes along paths that have stretch 3. No compact routing schemes are known for stretch less than 3 for general graphs; in fact, it is known that even for extremely

sparse graphs, any compact routing scheme that routes along paths of stretch less than 2 must use $\Omega(n)$ memory at some nodes in the network [78]. Hence, all we can hope for is compact routing schemes with stretch 2 and larger.

The aforementioned solutions have been proposed as centralized algorithms [58] and more recently as distributed protocols for wireless sensor networks [82], the Internet [83] and peer-to-peer networks [84]. In this section, we present compact routing schemes for our distance oracles; by exploiting graph sparsity, our schemes significantly improve the memory/stretch trade-off of previously known results. In particular, we discuss a surprisingly lightweight scheme that can be incorporated in distributed routing protocol implementations of the Thorup-Zwick (TZ) scheme, [83] for instance, to get a distributed routing protocol for our oracles. In addition, by setting $\alpha = \sqrt{n}$ in results from the previous section, we get a compact routing scheme that, for any source-destination pair at distance d, routes along paths of length at most $2d + 1$ by using $O(\sqrt{n})$ memory at each router – independent of the density of the graph.

For graphs with average degree $\mu = o(n)$, our scheme is the first compact routing scheme with the optimal stretch. The scheme requires $O(\sqrt{n\mu})$ memory at each router and route along paths of worst-case stretch 2. Besides being the first compact routing scheme (for general graphs) with provably optimal stretch, our compact routing scheme has a particular property: it can be implemented on top of any implementation of the TZ scheme using a *handshaking* scheme – a surprisingly lightweight end-to-end exchange of a small number of packets – and a small amount of processing to set up a new end-to-end connection with worst-case stretch 2. Using a distributed protocol [83] to construct the TZ scheme (with appropriately setting the parameters), we get a distributed name-independent compact routing scheme for our distance oracles with roughly the same space requirements.

We primarily focus on designing compact routing schemes for stretch 2. Recall that our distance oracle for stretch 2 has size $O(m\alpha+n^2/\alpha)$; our scheme distributes the state uniformly across all routers, requiring each router to store $O(\mu\alpha + n/\alpha)$ entries. Using $\alpha = \sqrt{n/\mu}$, our scheme requires each router to store $O(\sqrt{n\mu})$ entries, while routing along paths of stretch 2. For graphs with $\mu = o(n)$, this gives us the first scheme that routes along paths of stretch less than 3 and requires sublinear state at routers in the network. In fact, for real-world networks, that is networks with $\mu = \Theta(\text{polylog}(n))$, our compact routing scheme requires the same amount of memory as [58,82–84] but routes

along paths that have worst-case stretch bounded by 2. Note that any routing scheme with stretch less than 2 must require linear state at some node in the network [77] even for extremely sparse graphs; our scheme, hence, achieves the optimal stretch with non-trivial memory requirements at routers.

TZ scheme and our distance oracle. Our distance oracle can be incorporated into the proposed distributed adaptations [82–84] of the TZ scheme with minimal changes. This is due to the fact that the construction in our oracle, in concept, is similar to the TZ scheme: both schemes construct a set L of nodes and each node v stores a corresponding nearest neighbor $\ell(v)$ and certain nodes in its neighborhood. The first difference between our oracle and the TZ scheme is that the set L is sampled proportional to node degree rather than uniform-randomly. Second, our oracle differs from TZ scheme in terms of the information stored in the oracle: for any node v, while TZ only requires storing the ball $B(v)$, our oracles stores $B^\star(v)$. Both modifications are easy changes to the distributed protocols of [82–84]; note that computing $B^\star(v)$ requires only neighbors of nodes in $B(v)$. Third, to route from the source u to the destination v, our distance oracle allows u to set up an initial connection to v by using the TZ algorithm for routing between u and v. This initial connection gives a path of stretch 3, via an essentially unmodified proof of [41, 58]. The final task is to improve the stretch from 3 to 2.

Implementing ball-vicinity intersection. In order to improve the stretch from 3 to 2, our distance oracle requires the source and the destination to perform a ball-vicinity intersection (see Lemma 8). We show how vicinity intersection can be implemented in practice with a surprisingly lightweight *handshaking* scheme; that is, exchange of very few bytes between the source and the destination. Recall, from the discussion above, that the initial connection gives the source a path to the destination with stretch 3. The source can then send the list of nodes in its *ball* to the destination using this path. For the router-level map of the Internet measured by CAIDA [46], which consists of $n = 192,244$ routers and has average degree $\mu \simeq 0.4 \log_2 n$, this requires the source to transfer roughly $4 \cdot \sqrt{n/\mu}$ bytes, since IPv4 addresses are 4 bytes and balls have size $\sqrt{n/\mu}$. This amount to approximately 661 bytes of data; on today's Internet, packets are generally allowed to be at least 1500 bytes long, so this would take just one packet.

The destination can then perform a ball-vicinity intersection, which requires $O(\sqrt{n/\mu})$ time asymptotically but using the above numbers requires less than 165 hash table lookups which is fast in practice.[1] The destination then informs the source whether the ball-vicinity intersection is an empty set or not. If they do intersect, it can inform the source of the node (or nodes) at which ball-vicinity intersection occurs. This requires at most one packet which can be routed from the destination through the source via a stretch-3 path. The source-destination pair, after the above handshaking scheme (that requires at most two packets), now have a route with stretch 2.

In practice, this is likely to be efficient even for relatively short-lived connections. For much larger networks, of course, the exchange of ball information would require more bandwidth and computation; but since a stretch-3 path is available immediately, the reduction to stretch 2 can be treated as an optimization for longer flows in order to amortize the overhead.

Probing and Shortcutting. The protocol for implementing ball-vicinity intersection discussed above does not exploit the optimization discussed in §3.3 for heuristically improving the stretch for the retrieved paths. We discuss the implementation aspects related to the optimization. Implementing the optimization in practice leads to a process, which we call *probing and shortcutting* (P&S). P&S requires the source node to *probe* the nodes in its vicinity for improving stretch. We argue that this can be achieved with an extremely low overhead probing scheme. Once the source node finds a node in its vicinity that provides a better stretch, the source can conveniently switch the traffic through the *shortcut* path. We only discuss the probing mechanism, since shortcutting can be implemented easily in practice (note that the destination is oblivious to the shortcutting mechanism and hence, P&S does not require any handshaking mechanism).

For the probing mechanism, assume that the source opens an initial connection to a destination. The source, every 10^{th} packet, can probe a node in its vicinity (the question on deciding an appropriate order of probing the nodes in vicinity is discussed below) requesting the length of the path available from this node to the destination. These packets can be extremely small compared to the other data packets, leading to an extremely small overhead

[1] If the destination is a server, this could be a burden; but note that we could just as easily flip the protocol around so the source does the computation.

in terms of bandwidth consumed (just a fraction 0.1 more packets that are of negligible size compared to the data packets). Since the source-destination connections that account for most of the bandwidth sent on the networks are very long [85], we believe it is reasonable to amortize the cost of the probing over the lifetime of the connection.

In terms of the order of probing, we consider two heuristics. *Farthest-first*, in which the source probes the nodes that are the *boundary* nodes of its vicinity; and, *closest-first*, in which the source performs probing starting with the closest nodes (its neighbors). Our evaluation results suggest that the former performs better than the latter.

4.6 Evaluation

In this section, we evaluate the performance of our stretch 2 scheme on large-scale synthetic and realistic topologies. We first present our methodology, followed by a summary of the evaluation results and conclude with a detailed discussion on the results.

4.6.1 Methodology

We evaluate four schemes: the stretch-3 TZ scheme with landmarks selected uniform randomly, the stretch-3 TZ scheme with landmarks selected using our scheme, and two version of our stretch 2 scheme: the stretch-2 scheme with $\alpha = \sqrt{n}$ with and without the P&S optimization discussed in earlier section. For the TZ scheme, we sampled each vertex (for set L) with probability $\sqrt{\log n/n}$. For our stretch 2 scheme, each vertex was sampled with probability $\sqrt{n \log n} \times \deg(v)/\log^2 n$. All the constants in the big-O notation were set to be 1. All these schemes were evaluated using static simulator, assuming static graph topologies, which we describe next.

We present evaluation results for three topologies. (1) $G(n,m)$ random graphs, *i.e.*, $n = 16384$ nodes with m uniform-random edges, with m set so that the average degree is 6, (2) geometric random graphs with $n = 16384$ nodes with average degree 6, and (3) a $33,014$ node AS-level map of the Internet (referred to as the Internet graph in this section) [46].

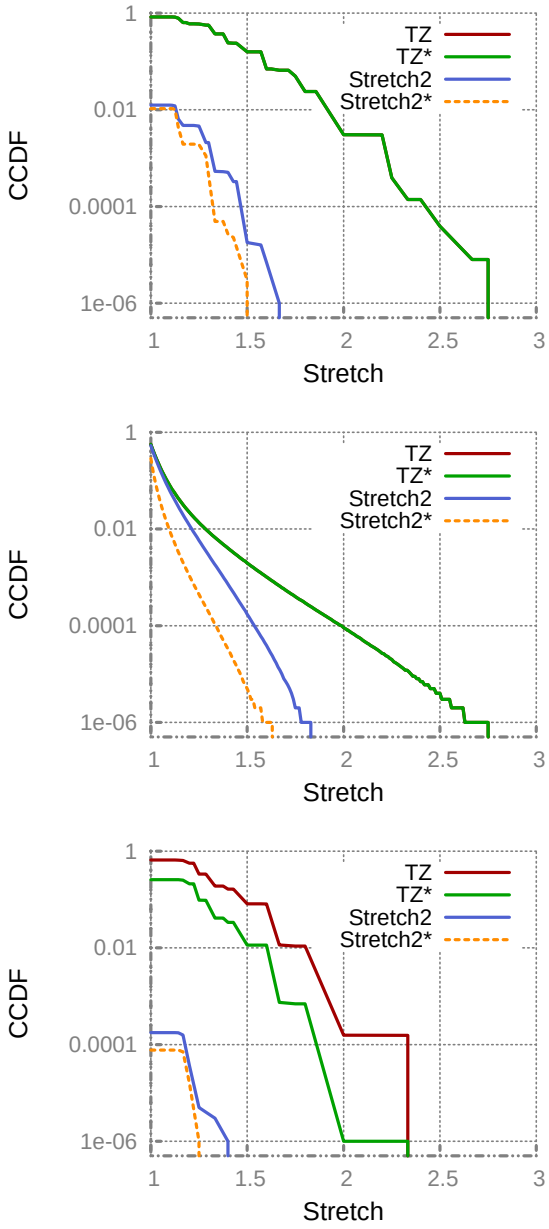


Figure 4.1: Comparison of the stretch for our stretch-2 oracle and the stretch-3 oracle of Thorup and Zwick [41] for $G(n,m)$ random graph (top), geometric random graph (middle) and AS-level internet map (bottom). As described in §4.6.1, TZ* is the scheme which uses the set of landmarks constructed by the algorithm of §4.2; Stretch-2* is the scheme which uses optimization discussed in §3.3.

For $G(n,m)$ graphs and the Internet graph, link weights are 1; for geometric random graphs, a link's weight is the Euclidean distance between the position of its two vertices. For $G(n,m)$ graphs and for geometric random graphs, we generated 10 different topologies with the same parameters and our results are the average of evaluations of these topologies. For geometric random graphs, we sampled a set of "source" vertices and evaluated the performance of the schemes from these sources to all the destinations. We found that sampling 1/4 of the nodes as sources provided accurate results.

4.6.2 Results and Discussions

Stretch comparison with the TZ-scheme. Fig. 4.1 shows the performance of the four schemes for various graph topologies. Note that our stretch 2 scheme returns exact shortest paths for nearly all source-destination pairs: 98.94% in the $G(n,m)$ graph, and 99.98% in the Internet graph. Though $G(n,m)$ graphs and the Internet graph have highly different structures, these graphs have a common feature: for nearly all vertex pairs, the two vicinities intersect, thus providing a shortest path. In the $G(n,m)$ graph (in which 96.2% source-destination pairs have intersecting vicinities), this occurs since, with high probability, the diameter of the graph is roughly at most twice the vicinity radius. In the Internet graph (in which 96.8% source-destination pairs have intersecting vicinities), vicinity intersection likely occurs at the "core" networks of the Internet. Since TZ scheme does not exploit the vicinity intersection, its performance is significantly worse than our schemes (only 34.4% of the source-destination pairs retrieved shortest paths).

The surprising difference between the performance of the two schemes may be due to the difference in the ways by which these schemes construct the landmark set L. We evaluated a modified version of the TZ scheme that uses the same set L as used by our schemes (see TZ* in Fig. 4.1). Although this improves the performance of the TZ scheme (74.2% of the source-destination pairs now retrieve shortest paths), it is still much worse than the our stretch 2 scheme. We, hence, believe that our schemes perform well due to the vicinity intersection idea.

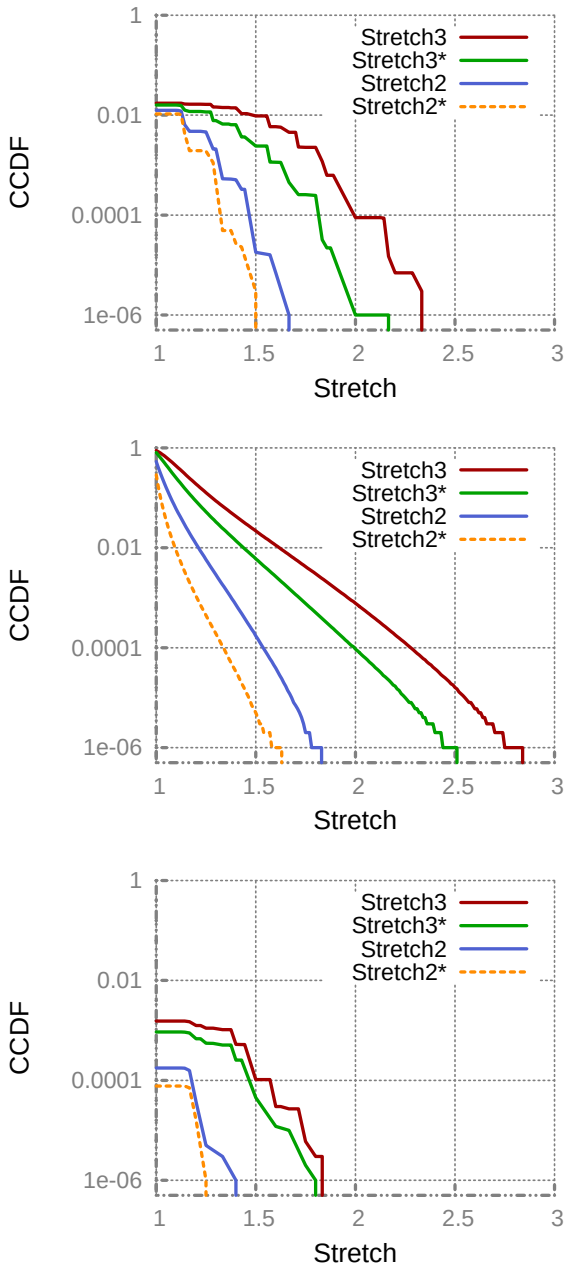


Figure 4.2: Comparison of the stretch for our stretch-2 and our stretch-3 oracle for $G(n, m)$ random graph (top), geometric random graph (middle) and AS-level internet map (bottom). Stretch-2* and stretch-3* are the schemes which use optimization discussed in §4.5.

For geometric random graphs, our stretch 2 scheme allows retrieval of shortest paths only for 70.7% of the source-destination pairs in comparison to 42.9% for the TZ scheme; indeed, only 4.8% of the source-destination pairs have intersecting vicinities. However, our stretch 2 scheme consistently performs better than the TZ-scheme.

Stretch comparison of stretch 2 and stretch 3 schemes. The performance of our stretch 2 scheme and our stretch 3 scheme for various graph topologies is compared in Fig. 4.2. We note that, as expected, our stretch 2 scheme consistently performs better than our stretch 3 scheme, even without the P&S optimization. However, the more interesting observation is that the P&S optimization is much more effective in our stretch 3 scheme. In particular, we note that the tail of our stretch 3 scheme without the P&S optimization is significantly reduced when the optimization is used.

For $G(n, m)$ graphs, the stretch for 99% of the source-destination pairs is less than 1.15 using our stretch 2 scheme. For our stretch 3 scheme, this is almost 1.3 (optimized version) and 1.5 (unoptimized version). The case of geometric random graphs is rather interesting: first, we observe that not many source-destination pairs have intersecting vicinities, otherwise our stretch 3 scheme without the P&S optimization would not have achieved such a low fraction of source-destination pairs retrieving shortest paths (only around 11%). Despite this, our stretch 2 scheme performs surprisingly well: almost 48% of the source-destination pairs retrieve shortest paths without the P&S optimization and almost 71% retrieve shortest paths with the P&S optimization.

4.7 Summary

In this chapter, we have presented distance oracles for stretch 2. Our oracles, using the same space as that of TZ-oracles, improve the worst-case stretch to 2. Moreover, our oracles exhibit a more general space-time trade-off for any fixed stretch — this provides a separation between the dense and the sparse cases for the distance oracle problem. Finally, our oracles not only achieve an improved worst-case bound on the stretch, they perform extremely well over real-world network topologies retrieving the exact shortest path for most source-destination pairs.

Chapter 5

Distance Oracles for Stretch Less Than 2

For general weighted undirected graphs, recall from Chapter 2 that Thorup and Zwick [41] showed a fundamental space-stretch trade-off — for any integer $k \geq 2$, they designed an oracle of size $O(kn^{1+1/k})$ that returned distances of stretch $(2k-1)$ in $O(k)$ time; the construction time of their oracle was $\widetilde{O}(kmn^{1/k})$, in expectation. The space-stretch trade-off of Thorup-Zwick oracle is essentially optimal for extremely dense graphs; they showed that any oracle for undirected graphs that returns distances of stretch less than $(2k+1)$ must have size $\Omega(n^{1+1/k})$.

Chapter 3 presented distance oracles that, for stretch 3 and larger, significantly break the space-stretch trade-off of Thorup-Zwick oracle by exploiting graph sparsity. We showed that it is possible to design oracles with linear space at the expense of a small super-constant query time. Chapter 4 extended these results for the case of stretch 2, giving a space-time trade-off and even a simple constant-time stretch-2 oracle. In this chapter, we achieve a space-stretch-time trade-off for stretch less than 2.

The results leading to this chapter were developed in two stages. In the first stage [51], we designed the first distance oracle for general weighted undirected graphs that computes distances of stretch less than 2 using subquadratic space and sub-linear query time. This chapter presents new oracles and a new query algorithm for stretch less than 2 that significantly outperform the oracles in [51] for *each point in the space-stretch-time trade-off space*. For instance, consider graphs with $m = \widetilde{O}(n)$ edges and fix the stretch to 1.66... Then, the oracle of [51] requires time $\widetilde{O}(n^{3/8})$ and $\widetilde{O}(n^{2/3})$ using space $O(n^{7/8})$ and $O(n^{5/3})$, respectively. A special case of oracles presented in this chapter further reduces the query time in the above examples to $\widetilde{O}(n^{1/8})$ and $\widetilde{O}(n^{1/3})$ without any increase in the space requirements.

5.1 Contributions and Techniques

This chapter presents distance oracles and a query algorithm for stretch less than 2. Our main result is as follows:

Theorem 7. *Let G be a non-negatively weighted undirected graph with n vertices, m edges and average degree $\mu = 2m/n$. Then, for any fixed $1 \leq \alpha \leq n$ and for any integer $k \geq 1$, there exist following distance oracles:*

Stretch	Space	Query time	Construction time	Remarks
$1+\frac{1}{k}$	$\widetilde{O}(m+n^2/\alpha)$	$O((\alpha\mu)^k)$	$\widetilde{O}(mn/\alpha)$	$1 \leq \alpha \leq n$
$1+\frac{1}{k+0.5}$	$\widetilde{O}(m+n^2/\alpha)$	$O(\alpha(\alpha\mu)^k)$	$\widetilde{O}(mn/\alpha)$	$1 \leq \alpha \leq n$
$1+\frac{2}{k+2}$	$\widetilde{O}(m+n^2/\alpha)$	$O((\alpha\mu)^k)$	$\widetilde{O}(mn/\alpha)$	$1 \leq \alpha \leq n^{2/3}m^{-1/3}$

Besides being the first stretch-less-than-2 distance oracle for general weighted undirected graphs, our results are interesting for the following reason. The problem of computing all-pair stretch-less-than-2 distances in undirected graphs is equivalent to combinatorial Boolean Matrix Multiplication (BMM) over the (OR, AND) semiring [86]. Hence, for $k = 1$, if the query time of the third oracle of Theorem 7 can be reduced to $O((\alpha\mu)^{1-\varepsilon})$ for any $\varepsilon > 0$, it would be possible to multiply two boolean matrices in time $\widetilde{O}(mn/\alpha + n^2(\alpha\mu)^{1-\varepsilon})$. By setting $\alpha = o((m/n)^\beta)$ for $\beta = \frac{\varepsilon}{1-\varepsilon}$, we get that the time would be $o(mn)$; note that the remark that $1 \leq \alpha \leq n^{2/3}m^{-1/3}$ is only for highlighting interesting cases for distance oracles. Hence, an improvement in the query time would lead to a purely $o(mn)$ time combinatorial algorithm for BMM, a long standing open problem [87, 88]. Alternatively, conditioned upon the hardness of $o(mn)$-time combinatorial BMM algorithm, our oracles may achieve an optimal trade-off between query time and construction time.

More generally, the parameters α and k in our results give a smooth trade-off between query time and stretch (for any fixed space) or between query time and space (for fixed stretch), leading to a smooth three-way trade-off between space, stretch and query time — a phenomenon that does not occur in dense graphs. The theorem also implies that on sufficiently sparse graphs, it is possible to retrieve distances with stretch arbitrarily close to 1 using sub-quadratic space and sub-linear query time.

Note that our oracles for stretch 1.67 achieve a space-time trade-off of $S \times T = O(n^2)$ for sparse graphs. For stretch $1 + 1/k$, the oracles of Theorem 7 achieve a space-stretch-time trade-off of $S \times T^{1/k} = O(n^2)$ for sparse graphs. Finally, we note that the query time in Theorem 7 can be further reduced using a small additive stretch, that depends only on k (see [51]).

Techniques. Far from being a narrow special case of the distance oracle problem, the problem of designing oracles with super-constant query time has a dramatically new mathematical structure, in that query time appears non-trivially within the space-stretch trade-off. It is, hence, not surprising that in contrast to constant-time oracles [41, 75, 76, 80] where the entire focus is on designing elegant and compact data structures, the techniques used in super-constant time oracles are often focused on designing algorithms that query a relatively simpler data structure but allow exploring a trade-off between space, stretch and query time (as in Chapter 3 and Chapter 4). Our main contribution is, indeed, such a query algorithm. We give a high-level idea of our query algorithm starting with the query algorithm of [51], followed by the query algorithm that allows us to achieve the results of Theorem 7.

Recall from Chapter 2 that the distance oracle of Thorup and Zwick [41], when queried for the distance between two vertices u, v, returns the exact distance if $u \in B(v)$ or if $v \in B(u)$; if not, it returns a distance of $d(u, \ell(u)) + d(\ell(u), v)$. In the latter case, by triangle inequality, we get that the returned distance is at most $2 \cdot d(u, \ell(u)) + d(u, v)$, which is at most $2 \cdot d(u, \ell(u))$ more than the exact distance. Hence, the stretch is given by $1 + 2d(u, \ell(u))/d(u, v)$. For Thorup-Zwick oracle, $d(u, \ell(u)) \approx d(u, v)$ in the worst-case, and hence, the resulting distance is of stretch 3 [41].

Our technique builds upon the above technique using two observations. First, given the above stretch-3 oracle, it may be possible to retrieve distances of lower stretch by carefully querying the oracle. More specifically, let u' be some vertex along the shortest path between u and v and suppose we know the exact distance $d(u, u')$. We can then query the oracle for distance between u' and v and return the distance $d(u, u') + \delta(u', v)$. As above, the exact distance is returned if $u' \in B(v)$ or $v \in B(u')$; if not, the returned distance is $d(u, u') + d(u', \ell(u')) + d(\ell(u'), v)$. Using triangle inequality, we get that the returned distance is at most $d(u, u') + 2 \cdot d(u', \ell(u')) + d(u', v) = 2 \cdot d(u', \ell(u')) + d(u, v)$. If $d(u', \ell(u')) < d(u, \ell(u))$, this in fact leads to a better

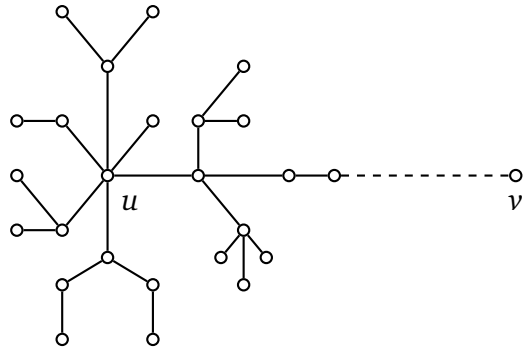
stretch; in fact, if we can find a vertex u' such that $d(u', \ell(u')) \leq d(u,v)/2k$, we will get the desired bound on stretch. We will show that such a vertex always exists and can be found within the desired bound on the query time.

We now give the high-level idea of proving the stretch bound of $(1+2/(k+1))$ in [51]. To achieve this bound, we use a recursive query algorithm that recursively explores the neighborhoods of the source or the destination in search of a good candidate vertex. More specifically, to find a good candidate vertex, we grow a partial shortest path tree around the source u until all the neighbors of $B(u)$ have been explored (see Figure 5.1(a)); alternative algorithms for finding these candidate vertices may grow shortest path trees around v (as in Figure 5.1(b)) or even around both u and v (as in Figure 5.1(c)). Growing these shortest path trees contributes to the query time of our algorithm. Once we have found a good candidate vertex u' among the neighbors of vertices in $B(u)$, we recurse; that is, we find a good candidate vertex u'' among the neighbors of vertices in $B(u')$ and so on. Once the depth of recursion has reached k, we are able to show that among all the candidate vertices explored during the recursive queries, we would have found a vertex w along the shortest path between u and v such that $d(w, \ell(w)) \leq d(u,v)/(k+1)$. This leads to the desired bound on stretch.

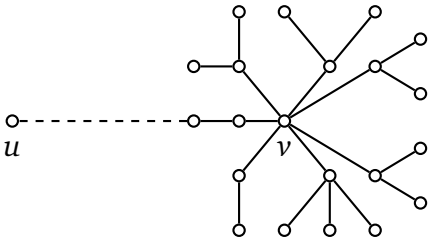
Our query algorithm for Theorem 7 improves upon the results in [51] using the idea of *bidirectional recursion* — it recursively computes (not necessarily shortest) distances between carefully chosen vertices, with recursion trees originating from both the source s and destination t. The main challenge is to define the vertices explored during such a bidirectional recursion in a manner that either (1) the algorithm explores[1] each edge along the shortest path between s and t; or (2) a lower bound on the exact distance between s and t can be proved. Indeed, a stronger lower bound on the exact $s-t$ distance is desired as the depth of the recursion increases.

Finally, the third oracle precomputes and stores certain "shortcuts" that allow to further speed-up the query algorithm. The shortcuts are computed so as to prove a lower bound similar to the query algorithm with bidirectional recursion but using a lower depth of recursion.

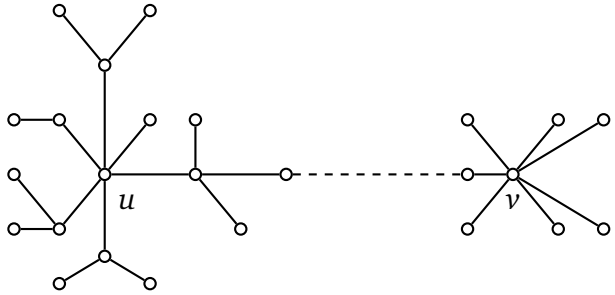
[1]Note that this, while similar in spirit, is substantially different from bidirectional shortest path algorithms [89–92]. In particular, the main challenge in reducing the query time is to avoid exploring dense neighborhoods of nodes that do not lie along the shortest path, which we achieve by carefully defining the neighborhoods explored by the query algorithm.



(a) Exploring the neighborhood of u in search of candidate vertex u'



(b) Exploring the neighborhood of v in search of candidate vertex u'



(c) Exploring the neighborhood of u and v in search of candidate vertex u'

Figure 5.1: Various possibilities of exploring the neighborhoods of the source and the destination in search of candidate vertex u'.

The space-stretch-time trade-off. We comment on the three-way trade-off between space, stretch and query time in our distance oracle.

For any fixed stretch, our distance oracles achieve the trade-off between space and query time by way of construction. Unlike the construction algorithms in [41, 76], the size of the landmark set L in our oracles is controlled by a parameter $1 \leq \alpha \leq n$. As we increase the size of L, the size of the oracle increases since it stores the distance from each vertex in L to each other vertex. On the other hand, as the size of L increases, fewer edges need to be explored while growing the shortest path trees in each recursive step, leading to a smaller query time. Hence, for any fixed stretch, we get a smooth space-time trade-off using the parameter α.

The other spectrum of the trade-off is achieved by using the recursive query algorithm — for a fixed size of the oracle, we get a trade-off between stretch and query time. Fix some $1 \leq \alpha \leq n$ and hence, the size of the oracle. Then,

we get a stretch-time trade-off by controlling the depth of recursion — the lower the desired stretch, the higher the query time. This is due to the fact that we are simply querying the same data structure (recursively) and hence, the size of the oracle is fixed; the query algorithm simply allows us to trade-off query time for improved stretch.

Related Work. Finally, we compare the results achieved in this chapter against related work. Perhaps, the work most closely related to ours is the result of Porat and Roditty [93]; they designed a distance oracle that returns distances of stretch less than 2 *for the special case of unweighted graphs*. Their oracle is of size $O(nm^{1-\varepsilon})$ and returns stretch-$(1+2\varepsilon)/(1-2\varepsilon)$ distances in time $O(m^{1-\varepsilon})$ for any unweighted undirected graph.

There are three main aspects in our distance oracle of Theorem 7 improves upon their oracle. First, our oracle significantly improves upon their results *for each point in the space-time-stretch trade-off space*. For instance, consider graphs with $m = \tilde{O}(n)$ edges. By setting $\alpha = n^{\varepsilon}$ and $k = (1-\varepsilon)/\varepsilon$, our oracle from [51] returns distances of stretch at most $1+2\varepsilon$ and our oracle of Theorem 7 returns distances of stretch at most $1+\min\{2\varepsilon/(1+\varepsilon), \varepsilon/(1-\varepsilon)\}$, an improvement over their oracle for each possible value of ε. For stretch 1.67, this is compared in Figure 5.2. Even for slightly denser graphs, the improvement is much more significant.

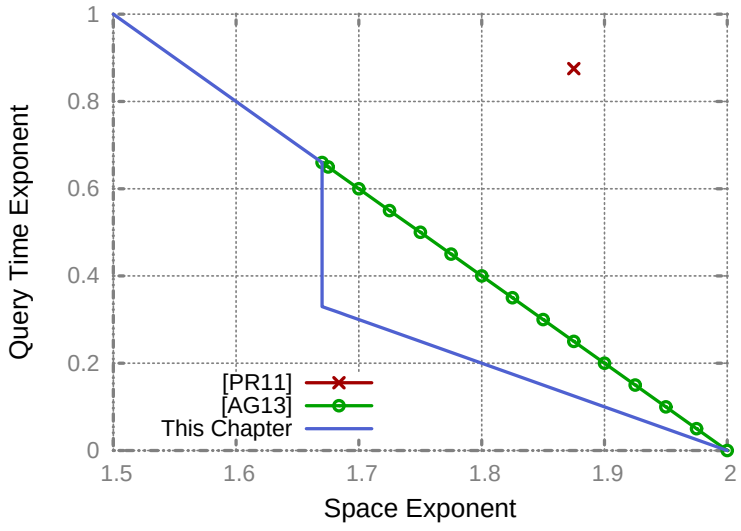


Figure 5.2: Space-time trade-off for stretch-1.66.. oracles in [93], [51] and this chapter for graphs with $m = \widetilde{O}(n)$ edges.

Second, unlike their oracle, our oracle works for general *weighted* graphs. Finally, their oracle exhibits the space-stretch trade-off as in classical distance oracles for dense graphs [41]; once the stretch is fixed, the space and query time is fixed. Our oracle exhibits a more general three-way trade-off highlighting a fundamental difference between the dense and the sparse cases.

For stretch values smaller than 1.67, we get similar improvements. The oracles of Theorem 7 reduce the query time of oracles in [51] from $O((\alpha\mu)^{2k-1})$ to $O((\alpha\mu)^k)$.

5.2 Stretch $\left(1+\frac{1}{k}\right)$ Oracle

In this section, we prove the first part of Theorem 7: for a weighted undirected graph with n vertices, m edges with non-negative weights and average degree $\mu = 2m/n$, and for any $1 \leq \alpha \leq n$, an oracle of size $\widetilde{O}(m + n^2/\alpha)$ that returns distances of stretch $1 + 1/k$ in time $O((\alpha\mu)^k)$. We need some notation to succinctly describe the construction.

5.2.1 i-Balls and i-Vicinities

We will generalize the idea of balls and vicinities from Chapter 2, §2.2. We define the i-vicinity of a vertex $v \in V$, denoted as $\Gamma^\star_i(v)$ as follows:

$$\Gamma^\star_0(v) = \{v\}; \qquad \text{and} \qquad \Gamma^\star_i(v) = \bigcup_{w \in \Gamma^\star_{i-1}(v)} B^\star(w) \tag{5.1}$$

For instance, the 1-vicinity of any vertex includes all the vertices is its vicinity and the 2-vicinity of any vertex v is the union of all the vicinities of vertices in $B^\star(v)$. We now define the i-ball of a vertex v:

$$\Gamma_0(v) = \emptyset; \qquad \text{and} \qquad \Gamma_i(v) = \bigcup_{w \in \Gamma^\star_{i-1}(v)} B(w) \tag{5.2}$$

Note that $\Gamma_i(v) \subseteq \Gamma^\star_i(v)$ for any vertex v. We will also need a generalization for the definition of the *candidate* distance from Chapter 2, §2.2.1. Given a vertex v and a vertex w in the i-vicinity of v, the candidate distance from v to w, denoted by $d'_v(w)$, is given by the cost of the least-cost path from v to w such that all intermediate vertices are contained in the i-ball of v.

5.2.2 Oracle and query algorithm

We will use the oracle of Chapter 4, §4.2. Our contribution here is a new query algorithm that we describe next (see Algorithm 5). In the first two steps, the query algorithm computes candidate distance from s and from t to each vertex in their respective k-vicinities; these distances are temporarily stored in a hash table. Then, the algorithm computes three sets of paths between s and t. The first set of paths are of the form $s \rightsquigarrow w \rightsquigarrow t$ via vertices w in $\Gamma_k^\star(s) \cap \Gamma_k^\star(t)$. The second set of paths are of the form $s \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow t$ via vertices $w \in \Gamma_k^\star(s)$. The third set of paths are of the form $t \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow s$ via vertices $w \in \Gamma_k^\star(t)$. Finally, the least-cost path among all the above three sets of paths is returned.

Algorithm 5 Query (s,t,k): the query algorithm for computing stretch-$(1+1/k)$ paths.

1: Compute candidate distance from s to each vertex in $\Gamma_k^\star(s)$
2: Compute candidate distance from t to each vertex in $\Gamma_k^\star(t)$
3: $\gamma_1 \leftarrow \infty$, $\gamma_2 \leftarrow \infty$, $\gamma_3 \leftarrow \infty$
4: $\gamma_1 \leftarrow \min_{w \in \Gamma_k^\star(s) \cap \Gamma_k^\star(t)} \left\{ d'_s(w) + d'_t(w) \right\}$
5: $\gamma_2 \leftarrow \min_{w \in \Gamma_k^\star(s)} \left\{ d'_s(w) + d(w, \ell(w)) + d(\ell(w), t) \right\}$
6: $\gamma_3 \leftarrow \min_{w \in \Gamma_k^\star(t)} \left\{ d'_t(w) + d(w, \ell(w)) + d(\ell(w), s) \right\}$
7: return $\min\{\gamma_1, \gamma_2, \gamma_3\}$

5.2.3 Analysis

For any pair of vertices $s, t \in V$, let $P(s,t) = (s, x_1, x_2, \ldots, t)$ denote the shortest path between s and t. Let $w_i^s(t)$ be the first vertex from $w_{i-1}^s(t)$ along $P(s,t)$ that is not contained in the ball of w_{i-1}^s, that is, let

$$w_i^s(t) = x_{i_0} \qquad \text{where} \qquad i_0 = \max\{i \ : \ x_j \in B(w_{i-1}^s(t)) \cap P(s,t), \forall j < i\}$$

and let

$$r_i^s(t) = \min_{j \le i} \{ d(w_j^s(t), \ell(w_j^s(t))) \}$$

When the context is clear, we will denote $w_i^s(t)$ and $r_i^s(t)$ simply as w_i^s and r_i^s.

Claim 5. *Let $P = (s, x_1, x_2, \ldots, t)$ be the shortest path between vertices s, t. Let x_{i_0} be the first vertex from s along P that does not lie in $B(s)$; that is, let $i_0 = \max\{i \ : \ x_j \in B(s) \cap P, \forall j < i\}$. Then, $d'_s(x_{i_0}) = d(s, x_{i_0})$.*

Proof: Let $w = x_{i_0}$ and let $w' = x_{i_0-1}$. Then, since $w_j \in B(s), \forall j < i_0$, each edge along the shortest path between s and w' is contained within $B(s)$ and hence, $d'_s(w') = d(s,w')$. Furthermore, w' is a neighbor of w and the edge (w',w) lies along the shortest path between s and w. Hence, by definition of the candidate distance, we get that $d'_s(w) \leq d(s,w') + \text{weight of edge}(w',w) = d(s,w') + d(w',w) = d(s,w)$. □

Claim 6. *Let $P(s,t) = (s, x_1, x_2, \ldots, t)$ be the shortest path between a pair of vertices s and t. Let i_0 and j_0 be such that $w^s_k = x_{i_0}$ and $w^t_k = x_{j_0}$. Then, for all $i \leq i_0$, $d'_s(x_i) = d(s,x_i)$ and for all $j \geq j_0$, $d'_t(x_j) = d(t,x_j)$.*

Proof: We prove the claim for the source s; the proof for the destination follows along similar lines. To start with, using Claim 5, we have that $d'_{w^s_{k-1}}(w^s_k) = d(w^s_{k-1}, w^s_k)$ since w^s_k lies along the shortest path between w^s_{k-1} and t and each edge along the shortest path between w^s_{k-1} and w^s_k is contained in $B(w^s_{k-1})$. Note that since each w^s_i lies along the shortest path between s and t, all that remains to prove is that all vertices x_i, $i < i_0$, are contained in the i-ball of s. Since $w^s_{i-1} \in \Gamma^\star_{i-1}(s)$ for all i, we have by definition of i-balls that each intermediate vertex along the shortest path between w^s_{i-1} and w^s_i is contained in the i-ball of s. Combined with the fact that $w^s_{i-1} \in \Gamma^\star_{i-1}(s)$, we get that each vertex along the shortest path between s and w^s_k is contained in the k-ball of s. The proof follows. □

Claim 7. *For any pair of vertices s,t, we have that $d(s,w^s_i) \geq i \cdot r^s_{i-1}$ and $d(t,w^t_i) \geq i \cdot r^t_{i-1}$.*

Proof: We start by observing that for any $j \geq 1$, we have that $w^s_j \in B^\star(w^s_{j-1}) \setminus B(w^s_{j-1})$. To see this, let $P(s,t) = (s, x_1, x_2, \ldots, t)$ be the shortest path between s and t and let j_0 be some index such that $w^s_j = x_{j_0}$; then, by definition of w^s_j, we have that $w^s_j \notin B(w^s_{j-1})$ and that $x_{j_0-1} \in B(w^s_{j-1})$. However, since x_{j_0-1} and $w^s_j = x_{j_0}$ are neighbors, we also have by definition of vicinities, $w^s_j \in B^\star(w^s_{j-1})$.

It follows from the above observation that, for any $j \geq 1$, we have that $d(w^s_{j-1}, w^s_j) \geq d(w^s_{j-1}, \ell(w^s_{j-1}))$. Furthermore, since all the w^s_j, $j \geq 0$, lie along the shortest path from s to t, we have that

$$d(s,w^s_i) = \sum_{j=0}^{i-1} d(w^s_j, w^s_{j+1}) \geq \sum_{j=0}^{i-1} d(w^s_j, \ell(w^s_j)) \geq i \cdot r^s_{i-1}$$

The proof for t follows along similar lines. □

We will also need the following two claims that are used to bound the stretch of the oracle:

Claim 8. *For any pair of vertices $s, t \in V$, if $w_k^s \notin \Gamma_k^\star(t)$, then we have that $d(s,t) \geq 2k \min\{r_{k-1}^s, r_{k-1}^t\}$.*

Proof: Note that w_k^t lies along the shortest path between t and w_k^s. Then, using Claim 7, we have that if $w_k^s \notin \Gamma_k^\star(t)$ then $d(t, w_k^s) \geq d(t, w_k^t) \geq k r_{k-1}^t$. Also, $d(s, w_k^s) \geq k r_{k-1}^s$. Since w_k^s lies along the shortest path between s and t, we get that $d(s,t) = d(s, w_k^s) + d(t, w_k^s) \geq k(r_{k-1}^s + r_{k-1}^t) \geq 2k \min\{r_{k-1}^s, r_{k-1}^t\}$. □

Claim 9. *For any pair of vertices $s, t \in V$, the query algorithm returns a distance estimate of at most $d(s,t) + 2\min\{r_{k-1}^s, r_{k-1}^t\}$.*

Proof: Let w_0 be one of the vertices in $P(s,t) \cap \Gamma_k^\star(s)$ with $d(w_0, \ell(w_0)) = r_{k-1}^s$ (such a vertex w_0 exists, by definition). Then, we have that $\gamma_2 \leq d_s'(w_0) + d(w_0, \ell(w_0)) + d(\ell(w_0), t)$, which using Claim 6 gives us that $\gamma_2 \leq d(s, w_0) + d(w_0, \ell(w_0)) + d(\ell(w_0), t)$. By triangle inequality, we get that $\gamma_2 \leq d(s, w_0) + 2d(w_0, \ell(w_0)) + d(w_0, t)$. Since w_0 lies along the shortest path between s and t, we get that $\gamma_2 \leq d(s,t) + 2d(w_0, \ell(w_0)) = d(s,t) + 2r_{k-1}^s$. Similarly, we get that $\gamma_3 \leq d(s,t) + 2r_{k-1}^t$. Since the algorithm returns the minimum of γ_2 and γ_3, the distance returned is at most $d(s,t) + 2\min\{r_{k-1}^s, r_{k-1}^t\}$, as claimed. □

Proof of first oracle of Theorem 7. Since the oracle stores only the input graph and the distance from each vertex in the graph to each vertex in a set L of size $\widetilde{O}(n/\alpha)$, we get that the size of the oracle is $\widetilde{O}(m + n^2/\alpha)$. Constructing the oracle requires computing a shortest path tree from each vertex in set L, and hence, requires time $\widetilde{O}(mn/\alpha)$.

Next, we bound the query time of the query algorithm. We first claim that the size of the k-vicinity of each vertex is bounded by $O((\alpha\mu)^k)$. This follows from the definition of the i-vicinity and from the fact that the vicinity of each vertex is bounded by $O(\alpha\mu)$. Furthermore, note that it takes $O(\alpha\mu)$ time to compute the candidate distance from any vertex v to vertices in $B^\star(v)$. Hence, by definition of i-vicinity, it takes time $O((\alpha\mu)^k)$ to compute the candidate distance from s to vertices in $\Gamma_k^\star(s)$. Finally, lines (4), (5) and (6) of Algorithm 5 clearly take time linear in the size of the i-vicinities of s and t, leading to the desired bound of $O((\alpha\mu)^k)$ on the query time.

Finally, we prove a bound on stretch. If $w_k^s \in \Gamma_k^\star(t)$, then $\gamma_1 \leq d'_s(w_k^s) + d'_t(w_k^s) = d(s, w_k^s) + d(t, w_k^s) = d(s,t)$; hence, the exact distance is returned. Consider the case when such is not the case. Then, by Claim 8, we have that the distance between s and t is lower bounded by $d(s,t) \geq 2k \min\{r_{k-1}^s, r_{k-1}^t\}$. On the other hand, from Claim 9, we have that the distance returned by the query algorithm is at most $d(s,t) + 2\min\{r_{k-1}^s, r_{k-1}^t\} \leq d(s,t) + 2d(s,t)/(2k)$, leading to the desired bound on stretch. □

5.3 Stretch $\left(1 + \frac{1}{k+0.5}\right)$ Oracle

We now describe our oracle and query algorithm for the second oracle of Theorem 7: for a weighted undirected graph with n vertices, m edges with non-negative weights and average degree $\mu = 2m/n$, and for any $1 \leq \alpha \leq n$, an oracle of size $\widetilde{O}(m + n^2/\alpha)$ that returns distances of stretch $1 + 1/(k + 0.5)$ in time $O(\alpha(\alpha\mu)^k)$. We will use the notation as described in §2 and §5.2.1.

5.3.1 Oracle and query algorithm

We will again use the oracle of Chapter 4, §4.2 with the addition that the exact distance from each vertex v to each vertex in $B(v)$ will be stored within the oracle. This will be required to speed-up our query algorithm, and as we will discuss later, does not increase the space requirements for the points of interest on the space-time trade-off. The query algorithm for this oracle (see Algorithm 6) is similar to that of Algorithm 5 with the only difference that the k-vicinities $\Gamma_k^\star(s)$ and $\Gamma_k^\star(t)$ are now replaced by $(k+1)$-balls $\Gamma_{k+1}(s)$ and $\Gamma_{k+1}(t)$, respectively (and γ_1, γ_2 and γ_3 modified accordingly).

Algorithm 6 Query (s, t, k): the query algorithm.

1: Compute candidate distance from s to each vertex in $\Gamma_{k+1}(s)$
2: Compute candidate distance from t to each vertex in $\Gamma_{k+1}(t)$
3: $\gamma_1 \leftarrow \infty$, $\gamma_2 \leftarrow \infty$, $\gamma_3 \leftarrow \infty$
4: $\gamma_1 \leftarrow \min_{w \in \Gamma_{k+1}(s) \cap \Gamma_{k+1}(t)} \left\{ d'_s(w) + d'_t(w) \right\}$
5: $\gamma_2 \leftarrow \min_{w \in \Gamma_{k+1}(s)} \left\{ d'_s(w) + d(w, \ell(w)) + d(\ell(w), t) \right\}$
6: $\gamma_3 \leftarrow \min_{w \in \Gamma_{k+1}(t)} \left\{ d'_t(w) + d(w, \ell(w)) + d(\ell(w), s) \right\}$
7: return $\min\{\gamma_1, \gamma_2, \gamma_3\}$

5.3.2 Analysis

The proof is facilitated by the following two claims that are used to bound the stretch of the oracle:

Claim 10. *For any pair of vertices s, t, if $w_k^s \notin \Gamma_{k+1}(t)$ then $d(s,t) \geq (2k+1)\min\{r_{k-1}^s, r_k^t\}$.*

Proof: Observe that $d(t,w_k^s) = d(t,w_k^t) + d(w_k^t, w_k^s) \geq d(t,w_k^t) + d(w_k^t, \ell(w_k^t)) \geq kr_{k-1}^t + r_k^t \geq (k+1)r_k^t$. Also, $d(s,w_k^s) \geq kr_{k-1}^s$. Since w_k^s lies along the shortest path between s and t, we get that $d(s,t) = d(s,w_k^s) + d(t,w_k^s) \geq kr_{k-1}^s + (k+1)r_k^t \geq (2k+1)\min\{r_{k-1}^s, r_k^t\}$, as desired. □

Claim 11. *For any pair of vertices s, t, the query algorithm returns a distance estimate of at most $d(s,t) + 2\min\{r_{k-1}^s, r_k^t\}$.*

Proof: Let w_0 be one of the vertices in $P(s,t) \cap \Gamma_k^\star(s)$ with $d(w_0, \ell(w_0)) = r_{k-1}^s$ (again, such a vertex w_0 exists, by definition). Then, as in the proof of Claim 9, we have that $\gamma_2 \leq d(s,t) + 2r_{k-1}^s$. Similarly, we get that $\gamma_3 \leq d(s,t) + 2r_k^t$ (here, we choose a vertex w_0 in $P(s,t) \cap \Gamma_{k+1}(t)$ with $d(w_0, \ell(w_0)) = r_k^t$). Since the algorithm returns the minimum of γ_2 and γ_3, we get that the distance returned is at most $d(s,t) + 2\min\{r_{k-1}^s, r_k^t\}$, as claimed. □

Proof of second oracle of Theorem 7. We start with the size of the oracle. The oracle stores, in addition to the oracle of Chapter 4, §4.2, the distance from each vertex v to each vertex in $B(v)$. Hence, the size of the oracle is $\widetilde{O}(m + n\alpha + n^2/\alpha)$. Clearly, the cases of interest for this oracle require $\alpha = o(\sqrt{n})$ (since we require sub-linear query time), for which the size of the oracle is $\widetilde{O}(m + n^2/\alpha)$. Constructing the oracle requires computing the ball of each vertex and computing a shortest path tree from each vertex in the set L, and hence, requires time $\widetilde{O}(m\alpha + mn/\alpha) = \widetilde{O}(mn/\alpha)$ for the cases of interest, that is, $\alpha = o(\sqrt{n})$.

Regarding the query time of the oracle, we first note that the size of the $k+1$-ball of any vertex is at most $O(\alpha(\alpha\mu)^k)$ since the size of the ball of each vertex is at most $O(\alpha)$. Furthermore, note that the distance from each vertex v to all vertices in the ball of v is stored in the oracle during the preprocessing phase. Hence, by definition of i-balls, the time taken to compute the candidate distance from any vertex v to each vertex in the $\Gamma_{k+1}(v)$ is a factor $O(\alpha)$ more than the time taken to compute the candidate distance to each vertex in $\Gamma_k^\star(v)$. It follows that the query time is $O(\alpha(\alpha\mu)^k)$, as desired.

Finally, we prove a bound on stretch. If $w_k^s \in \Gamma_{k+1}(t)$, then $\gamma_1 \leq d'_s(w_k^s) + d'_t(w_k^s) = d(s, w_k^s) + d(t, w_k^s) = d(s,t)$; hence, the exact distance is returned. Consider the case when such is not the case. Then, by Claim 10, we have that the distance between s and t is lower bounded by $d(s,t) \geq (2k+1)\min\{r_{k-1}^s, r_k^t\}$. On the other hand, from Claim 11, we have that the distance returned by the query algorithm is at most $d(s,t)+2\min\{r_{k-1}^s, r_k^t\} \leq d(s,t)+2d(s,t)/(2k+1)$, leading to the desired bound on stretch. □

5.4 Stretch $\left(1 + \frac{2}{k+2}\right)$ Oracle

We now describe our oracle and query algorithm for the third oracle of Theorem 7: for a weighted undirected graph with n vertices, m edges with non-negative weights and average degree $\mu = 2m/n$, and for any $1 \leq \alpha \leq n^{2/3}m^{-1/3}$, an oracle of size $\widetilde{O}(m + n^2/\alpha)$ that returns distances of stretch $1+2/(k+2)$ in time $O(\alpha(\alpha\mu)^k)$. We will use the notation as described in §2 and §5.2.1.

5.4.1 Oracle and query algorithm

Fix some $1 \leq \alpha \leq n$. The preprocessing algorithm begins by replacing the original graph with a degree-bounded graph using the result of Corollary 1. The algorithm then samples a set L of vertices of size $\widetilde{O}(n/\alpha)$ using the result of Lemma 6. The algorithm then constructs a data structure that stores, for each $v \in V$:

- a hash table storing the shortest distance to each vertex in L;
- the nearest neighbor $\ell(v)$ and the ball radius r_v;
- a hash table storing the distance $d'_s(w) = \min_{x \in B^\star(v) \cap B(w)} d'_s(x) + d(x,w)$ to each vertex w in the set $S_v = \{w \ : \ B^\star(v) \cap B(w) \neq \emptyset\}$, that is, to all vertices w whose ball intersects with the *vicinity* of v.

The oracle also stores the degree-bounded graph computed in the first step of the preprocessing algorithm.

We now describe our query algorithm (see Algorithm 7). In the first and the second step, the query algorithm computes candidate distances from s and t to vertices in their respective vicinities; these distances and temporarily stored in a hash table. The algorithm then computes three set of paths. The first set of paths is of the form $s \rightsquigarrow w \rightsquigarrow w' \rightsquigarrow t$ for some $w \in B^\star(s)$ and $w' \in S_s \cap B^\star(t)$. The second set of paths are of the form $s \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow t$ for vertices $w \in B^\star(s)$ and the final set of paths are of the form $t \rightsquigarrow w \rightsquigarrow \ell(w) \rightsquigarrow s$ for vertices $w \in B^\star(t)$. The least-cost path among these paths is returned by the algorithm. This completes the description of the preprocessing and the query algorithms.

Algorithm 7 Query (s, t): the query algorithm.

1: Compute candidate distance from s to each vertex in $B^\star(s)$
2: Compute candidate distance from t to each vertex in $B^\star(t)$
3: $\gamma_1 \leftarrow \infty$, $\gamma_2 \leftarrow \infty$, $\gamma_3 \leftarrow \infty$
4: $\gamma_1 \leftarrow \min_{w \in S_s \cap B^\star(t)} \left\{ d(s,w) + d'_t(w) \right\}$
5: $\gamma_2 \leftarrow \min_{w \in B^\star(s)} \left\{ d'_s(w) + d(w, \ell(w)) + d(\ell(w), t) \right\}$
6: $\gamma_3 \leftarrow \min_{w \in B^\star(t)} \left\{ d'_t(w) + d(w, \ell(w)) + d(\ell(w), s) \right\}$
7: return $\min\{\gamma_1, \gamma_2, \gamma_3\}$

5.4.2 Analysis

Lemma 12. *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = O(m/n)$. For any fixed $1 \leq \alpha \leq n$, let L be the set of vertices sampled using the algorithm of Lemma 6. Then, $\sum_{v \in V} |S_v| \leq O(m\alpha^2)$.*

Proof: Let $\gamma(w)$ be the number of pairs of vertices whose ball-vicinity intersection contains w; that is, $\gamma(w) = |\{(u, v) \ : \ w \in B(u) \cap B^\star(v)\}|$. Then, by definition, $\sum_{v \in V} |S_v| \leq \sum_{w \in V} \gamma(w)$. Since the set of vertices in L are sampled using Lemma 6, for each vertex $w \in V$, we have that $|\overline{B}(w)| = O(\alpha)$ and hence, $|\overline{B}^\star(w)| = O(\alpha\mu)$. Hence, any vertex w (deterministically) belongs to at most $O(\alpha)$ balls and at most $O(\alpha\mu)$ vicinities. How many ball-vicinity intersections can occur at w then? Clearly, this is bounded by $O(\alpha^2\mu)$. Hence, we have that for any vertex $w \in V$, $\gamma(w) = O(\alpha^2\mu)$. Hence, $\sum_{v \in V} |S_v| \leq \sum_{w \in V} \gamma(w) = O(m\alpha^2)$. □

Claim 12. *Let $G = (V, E)$ be a weighted undirected graph with n vertices, m edges and maximum degree $\mu = O(m/n)$. For any fixed $1 \leq \alpha \leq n$, let L be the set of vertices sampled using the algorithm of Lemma 6. Then, constructing a hash table that contains, for each vertex $v \in V$, distance to each vertex in S_v can be constructed in time $O(m\alpha^2)$.*

Proof: We begin by constructing, for each vertex v, a hash table containing the shortest distance to each vertex in $B(v)$ and a hash table containing the candidate distance to each vertex in $B^\star(v)$; for a $\mu = O(m/n)$-degree bounded graph, this takes time $O(\alpha\mu)$ per vertex and a total of $O(m\alpha)$ time. Using these hash tables, it is rather trivial to construct, for each vertex v, a hash table containing the distances to each vertex in $\overline{B}(v)$ and in $\overline{B}^\star(v)$. Finally, the distances required for the set $\cup_{v \in V} S_v$ can be computed in time $O(m\alpha^2)$ by iterating through each vertex w and computing the distance $d(w, x) + d'_y(w)$ for each $x \in \overline{B}(w)$ and for each $y \in \overline{B}^\star(w)$. □

Proof of the third oracle of Theorem 7 for $k = 1$. The oracle stores, in addition to the oracle of Chapter 4, §4.2 the distance from each vertex v to vertices in set S_v. Using CLemma 12, it follows that the size of the oracle if $\widetilde{O}(m\alpha^2 + m + n^2/\alpha^2)$, as desired. The construction of the oracle requires running a shortest path algorithm from each vertex in L and computing distances to vertices in set S_v for each vertex v. Using Lemma 6 and Claim 12, it follows that the oracle can be constructed in time $\widetilde{O}(m\alpha^2 + n^2/\alpha)$. Finally, to bound the query time, recall that the size of the vicinity of each vertex is bounded by $O(\alpha\mu)$ and a candidate distance to each vertex in the vicinity can be computed in time $O(\alpha\mu)$; the bound follows.

Let $P = (s, x_1, x_2, \ldots, t)$ be the shortest path between s and t. Let $i_0 = \max\{i | x_i \notin P \cap B^\star(t)\}$ and $w = x_{i_0+1}$; note that $x_{i_0} \notin B^\star(t)$ and hence, $w \in B^\star(t) \setminus B(t)$. If $w \in S_s$, we get that $\gamma_1 \leq d(s, w) + d'_t(w) = d(s, w) + d(t, w) = d(s, t)$, since w lies along P; hence, the algorithm returns the exact distance. Consider the case when $w \notin S_s$. In this case, using Lemma 8, we get that $d(s, w) \geq 2\min\{r_s, r_w\}$; also $d(t, w) \geq r_t$. Using the fact that w lies along the shortest path between s and t, we get that $d(s, t) \geq 2\min\{r_s, r_w\} + r_t \geq 3\min\{r_s, r_w, r_t\}$. We now give an upper bound on the distance returned by the query algorithm. Note that $s \in B^\star(s)$ and $t \in B^\star(t)$; it follows that $\gamma_2 \leq d(s, \ell(s)) + d(\ell(s), t) \leq 2d(s, \ell(s)) + d(s, t) = 2r_s + d(s, t)$. Similarly, we get that $\gamma_3 \leq 2r_t + d(s, t)$.

Finally, since $w \in B^\star(t)$, we get that $\gamma_3 \leq d'_t(w) + d(w, \ell(w)) + d(\ell(w), s)$. Since w lies along the shortest path between s and t, we get that $d'_t(w) = d(t, w)$; using this fact along with the triangle inequality, we get that $\gamma_3 \leq d(t, w) + 2d(w, \ell(w)) + d(w, s) = 2r_w + d(s, t)$. Hence, $\gamma_3 \leq 2\min\{r_w, r_t\} + d(s, t)$. Since the query algorithm returns the minimum of γ_2 and γ_3, we get that the returned distance is at most $2\min\{r_s, r_t, r_w\} + d(s, t)$. The proof follows using the upper bound established above, which says that $\min\{r_s, r_t, r_w\} \leq d(s, t)/3$. □

Chapter 6

ASAP: Shortest Paths in Microseconds

Computing distances and paths is a fundamental primitive in social network analysis. We described several concrete applications in §1.1.2, including social search and ranking, socially-sensitive and location-aware search, and social auctions. This chapter is particularly motivated by the first three applications mentioned in §1.1.2; these applications fall into one of the following two categories. The first category of applications are the ones that require quickly computing multiple short paths between a given pair of users, and the second category of applications require quickly computing a single short path between a user X and multiple users (or contents). These applications require or can benefit from computing *shortest paths for most queries*, which we focus on in this chapter.

Scaling shortest path computation to massive social networks is challenging for two reasons. First, the applications above compute paths in response to user queries and hence have rather stringent latency requirements [38]. This precludes the obvious option of running a shortest path algorithm like $A^\star$ search [91, 92] or bidirectional search [92] for each query — our evaluation results in §6.5 suggest that these algorithms require hundreds of milliseconds even on moderate size networks.

Second, the massive size of social networks makes it infeasible to precompute and store shortest paths; even for a social network with 3 million users, this would require 4.5 trillion entries. Citing lack of efficient techniques for computing shortest paths, a number of papers have developed techniques to compute approximate distances and paths [26, 27, 41, 94, 95]. We delay a complete discussion of related work to §6.6; however, we note that these techniques either compute paths that are significantly longer than the actual shortest path or do not meet the latency requirements.

6.1 Overview and Contributions

We present ASAP, a system that quickly computes shortest paths for most queries on social networks while maintaining feasible memory requirements. ASAP preprocesses the network to compute a *partial shortest path tree* (PSPT) for each node. PSPTs have the property that for any two nodes s, t, each edge along the shortest path is highly likely to be contained in the PSPT of either s or t; that is, there is one node w that belongs to the PSPT of both s and t. Hence, a shortest path can be computed by combining paths $s \rightsquigarrow w$ and $t \rightsquigarrow w$. For the unlikely case of PSPTs not intersecting along the shortest path, ASAP computes a path that is at most one hop longer than the shortest path.

ASAP presents several contributions. First, compared to techniques for approximate shortest path computation [26, 27, 41, 94, 95], ASAP focuses on a much harder problem of computing shortest paths for most queries and, even on networks with millions of nodes and edges, computes shortest paths in *tens of microseconds*. Second, ASAP demonstrates and exploits the observation that the structure of social networks enable the PSPT of each node to be an extremely small fraction of the entire network. It is known that planar graphs exhibit a similar structure [96], but that social networks exhibit such a structure despite having significantly different properties is interesting in its own right. Finally, unlike most previous works, ASAP admits efficient distributed implementation and can be easily mapped on distributed programming frameworks like MapReduce.

ASAP, for a real-world network with 5 million nodes and 69 million edges, computes the shortest path between 99.83% of the node pairs in less than 49 μs — 3196× faster than the bidirectional shortest path algorithm [92]; computes a path that is at most one hop longer than the shortest path (when the PSPTs of the node pair intersect, but not along the shortest path) for an additional 0.15% of the node pairs in 49 μs; and runs a bidirectional shortest path algorithm for the remaining 0.02% of the node pairs (when the PSPTs of the node pair do not intersect). These results enable the first set of applications discussed earlier. For the second set of applications, ASAP allows computing hundreds of paths and corresponding distances between more than 99.98% of the node pairs in less than 100 μs without any change in the data structure for single shortest path computation, thus enabling distance-based social search and ranking in a unified way.

6.2 ASAP Sketch and Queries

We start the section by formally defining the PSPT of a node, a structure that forms the most basic component of ASAP (§6.2.1). We then describe the ASAP algorithm for computing the shortest path between a given pair of nodes (§6.2.2). In §6.2.3 and §6.2.4, we describe a low-memory, low-latency implementation of ASAP and extensions of ASAP that allow computing multiple paths. We assume that the input network $G = (V, E)$ is an undirected weig-hted network; each edge is assigned a non-negative weight and each node is assigned a unique identifier.

6.2.1 Partial Shortest Path Trees

We now define the PSPT of each node. At a high level, we will require that the PSPTs of any pair of nodes s, t satisfy the following property: there exists a node w along the shortest path between s and t such that (1) w is contained in the PSPT of both s and t (or equivalently, the two PSPTs *intersect* along the shortest path); (2) the path $s \rightsquigarrow w$ is contained in the PSPT of s; and (3) the path $t \rightsquigarrow w$ is contained in the PSPT of w.

Note that nodes that have only one neighbor (degree-1 nodes) can never lie along any shortest path; hence, PSPTs do not need to contain degree-1 nodes. To this end, let $G' = (V', E')$ be the network achieved by removing from V all degree-1 nodes and from E all edges incident on degree-1 nodes. Then, the PSPT of size β of any node u is the set of β closest nodes of u in G', *ties broken lexicographically [97] using the unique identifiers of the nodes*.

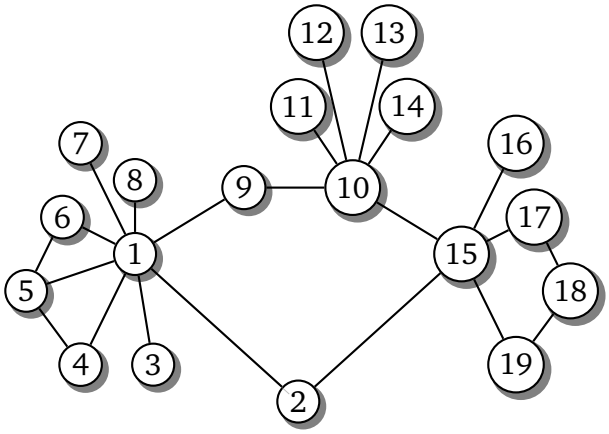
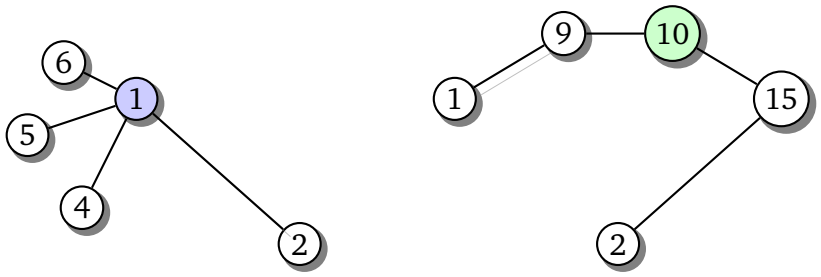


Figure 6.1: A running example network for the chapter.

An example. We explain the idea of PSPT using an example (see Figure 6.1 for the example network; and, Figure 6.2 for the example PSPTs of several nodes). Suppose we want to compute PSPT of size 5 for each node. We first remove all degree-1 nodes from the network of Figure 6.1, namely, nodes $\{3, 7, 8, 11, 12, 13, 14, 16\}$. Now, let us construct the PSPT of node 1. Among the remaining nodes, the nodes at distance 1 from node 1 are $\{1, 2, 4, 5, 6, 9\}$. By breaking ties lexicographically, we get that the PSPT of node 1 is given by $\{1, 2, 4, 5, 6\}$ (node 9 is lexicographically larger than the other nodes). The resulting PSPT is shown in Figure 6.2(a). Similarly, we can construct the PSPT of node 10, which is given by $\{1, 2, 9, 10, 15\}$ as shown in Figure 6.2(b). Finally, note that the PSPT of node 15 is given by $\{2, 10, 15, 17, 19\}$.



(a) PSPT of node 1 (b) PSPT of node 10

Figure 6.2: An example to explain the idea of node PSPTs for the network of Figure 6.1. Here, we construct PSPT of size 5 for each node.

We make several interesting observations using the above example. First, for some networks, it may be the case that all the nodes at a specific distance may be contained in the PSPT; for instance, all nodes within distance 1 of node 15 are contained in its PSPT. On the other hand, it may be the case that the PSPT of a node may not even include all its immediate neighbors; for instance, due to the tie-breaking scheme used in the PSPT definition, node 9 is not contained in the PSPT of node 1. Finally, we note that for different nodes, the PSPT may expand to different distances (distance 2 for node 10 while only distance 1 for node 15). We remark that the network in the example has unit weight edges only for simplicity; our definition of PSPTs and the following discussion does not make any assumption on edge weights.

6.2.2 The Algorithm

During the preprocessing phase, ASAP computes a data structure that is used to quickly compute paths during the query phase. We will describe the data structure in §6.2.3; here we describe the meta information stored in this data structure. ASAP computes and stores three pieces of information during the preprocessing phase:

- for each degree-1 node, the identifier of its neighbor and the distance to this neighbor;
- for each node u of degree greater than 1, the identifier of and the distance to each node w in the PSPT of u; ASAP constructs, for each node of degree greater than 1, a PSPT of size $4\sqrt{n}$ — the precise reasons for constructing PSPTs of this specific size are discussed in §6.6.
- for each node u of degree greater than 1 and for each node w in the PSPT of u, the identifier of the first node along the shortest path from w to u.

Using the above three pieces of information, ASAP computes the shortest path between any pair of nodes s and t in two steps (see Algorithm 8). Assume that both nodes are of degree greater than 1 and that the two PSPTs intersect along the shortest path. Then, the first step of the algorithm (lines 5–10) finds the node w_0 along the shortest path that is contained in both the PSPTs — it iterates through each node w in the PSPT of s and checks if w is contained in the PSPT of t; if it does, then the sum of distance from s to w and from t to w is a candidate shortest distance. The node w that corresponds to the minimum of all the candidate distances is in fact node w_0.

Once the node w_0 that lies along the shortest path between s and t is identified, the next step is to compute the path itself. This is done using the second step of the algorithm (lines 11–13) and using the third piece of information stored during the preprocessing stage. In this step, the algorithm computes the subpath $w_0 \rightsquigarrow s$ by following the series of next-hops starting w_0 until s and the subpath $w_0 \rightsquigarrow t$ by following the series of next-hops starting w_0 until t. Note that both of these subpaths are completely contained within the PSPT of s and t, respectively. The path from s to t is then returned by concatenating the two subpaths.

Algorithm 8 QuerySP(s,t) — ASAP algorithm for computing the exact distance between nodes s and t.

```
1:  If |N(s)| = 1
2:      return d(s, N(s)) + QuerySP(N(s), t)
3:  If |N(t)| = 1
4:      return d(t, N(t)) + QuerySP(s, N(t))
5:  δ ← ∞;      w_0 ← ∅
6:  For each w in PSPT of s
7:      If w in PSPT of t
8:          If d(s, w) + d(t, w) < δ
9:              δ ← d(s, w) + d(t, w)
10:             w_0 ← w
11: If w_0 ≠ ∅
12:     Compute path s ⇝ w_0 and path w_0 ⇝ t
13:     Return path s ⇝ w_0 ⇝ t
14: Else
15:     Run a bidirectional shortest path algorithm
```

We now resolve the two assumptions made in the above description. First, if either of the nodes has degree 1, we replace the node by its neighbor and add the corresponding distance in the result (lines 1–4). Regarding the second assumption, we consider two cases. First, when the PSPT of the two nodes intersect but not along the shortest path, ASAP returns a path that is not the shortest path; however, ASAP provides the following guarantee on the length of the returned path. Let $P(s,t)$ denote the shortest path between s and t and let w_{st} be the weight of the heaviest edge along $P(s,t)$; then, when the PSPT of s and t intersect but not along the shortest path, ASAP returns a path of length which is at most w_{st} longer than the shortest path length. The second case is when the PSPT of the two nodes do not intersect at all; the current implementation of ASAP simply runs a bidirectional shortest path algorithm for this case. As we will show in the next section, the latter two cases occur with an extremely low probability.

In §6.2.4, we will discuss an extension to Algorithm 8 that allows computing multiple paths between a given pair of nodes. We first discuss a low-memory low-latency implementation of ASAP.

6.2.3 An Efficient Implementation

One trivial but inefficient way of storing PSPTs and checking for PSPT intersection is by using hash tables. In particular, for each node u, we construct a hash table with each key being the identifier of a node w in the PSPT of u and the corresponding value being the distance between u and w and the next-hop along the shortest path from u to w. The PSPT intersection step in Algorithm 8 can then be trivially implemented using hash table lookups. However, a hash table based implementation is inefficient due to two reasons. First, storing PSPTs using hash tables has a non-trivial memory overhead; our experiments suggest that hash tables require up to 6–48× more memory when compared to on-disk space requirements. Second, while hash tables have a constant lookup time on an average, the absolute time required for each lookup may be large when compared to, say, comparing two integers.

For our implementation of ASAP, we use arrays for storing the PSPTs. In our experiments, an array based implementation required 3-24× less memory and 4-5× lower latency compared to a hash table based implementation. An array-based implementation of Algorithm 8 is fairly straightforward; we briefly describe it here for sake of completeness.

For each node u, an array stores the nodes in the PSPT of u in increasing order of their node identifiers; hence, the node at index i in the array has the i^{th} smallest identifier among the nodes in the PSPT of u. To check for PSPT intersection for a pair of nodes u and v, one pointer per array is maintained; each of the pointers is initially set to the first index of the respective array. In each step, the node identifiers corresponding to the two pointers are compared (say u_i and v_j). Note that if $u_i > v_j$, none of the nodes v_k, $k \leq j$ can have an identifier same as that of u_i (this is where storing nodes in the PSPT in increasing order of identifiers help!); hence, the pointer of v is advanced to v_{j+1}. Using the same argument, if $u_i < v_j$, the pointer of u is advanced to u_{i+1}. The final case is when $u_i = v_j$. In this case, the node with identifier $u_i = v_j$ lies in both the PSPTs and hence there is a candidate path of length $d(u,u_i) + d(v,v_j)$ between u and v. In this case, the pointers are advanced to u_{i+1} and v_{j+1}, respectively. The algorithm terminates when one of the pointers attempt to move beyond the length of the array, and returns the minimum of all candidate path lengths.

To aid path computations, we slightly modify the structure of our array — for each node u and each node w in the PSPT of u, the array will now store (in addition to the node identifiers and corresponding distances) the index at which the node identifier of the first hop along the shortest path *from w to v* is stored in the array.

6.2.4 Extension for Computing Multiple Paths

We now extend Algorithm 8 to facilitate computing multiple paths between any node pair. The high-level idea is to output a path corresponding to each intersection of the two PSPTs while avoiding duplicate paths. To achieve this, we maintain a list of "visited" nodes during the execution of Algorithm 8; these are the nodes that lie along paths that have already been output by the algorithm. More specifically, for any pair of nodes s, t, upon finding a node w that belongs to the PSPT of both s and t, the algorithm first checks if w is marked as visited; if yes, we ignore node w. If not, we compute subpaths $s \rightsquigarrow w$ and $w \rightsquigarrow t$ and mark each node along these two subpaths as visited and outputs the path $s \rightsquigarrow w \rightsquigarrow t$, as earlier. It is easy to see that by maintaining such list of visited nodes, the algorithm never outputs duplicate paths.

6.3 Unweighted Graphs

The technique of §6.2 computes shortest paths by performing as many PSPT intersection checks as the size of the PSPT — it iterates through *each node* in the PSPT of the source and checks if the node is contained in the PSPT of the destination, while keeping track of the shortest path seen so far. Some of these checks may be unnecessary; for example, if the first intersection occurs along the shortest path, the later checks are deemed unnecessary. However, for weighted networks, there is no trivial condition to establish that further PSPT intersection checks are unnecessary.

Many applications do model networks as unweighted networks — LinkedIn uses hop counts on user profiles, Microsoft citation network computes distances between authors, to name a few. For such networks, we can precisely identify two such conditions; if either of the conditions is satisfied, it is ensured that further checks cannot lead to a path of shorter length and the currently

lowest cost path is indeed the shortest path. These conditions allow us to reduce the number of PSPT intersection checks, and hence the latency, required to compute the shortest path for unweighted networks. For instance, by using these conditions on top of Algorithm 8, the resulting algorithm can compute shortest paths for the LiveJournal network up to 10× faster than the algorithm for weighted networks, without any loss in accuracy.

6.3.1 The Algorithm

For this technique, we will use the idea of the **boundary of a node**. For any node u and a node v in the PSPT of u, we say that v belongs to the **boundary of** u if there is at least one neighbor of v that has degree greater than 1 and does not belong to the PSPT of u; in other words, if v has links to nodes of degree greater than 1 that are outside the PSPT of u. For example, consider node 18 in Figure 6.1; if we construct PSPTs of size 5, then the PSPT of node 18 is given by $\{2, 15, 17, 18, 19\}$ and the boundary of node 18 is given by $\{2, 15\}$. If r_s denotes the PSPT radius of a node s, then we note that each node on the boundary of s is either at distance r_s or at distance $r_s - 1$; for instance, in the above example, node 2 is at distance $r_s = 3$ and node 15 is at distance $r_s - 1 = 2$ from node 18. The notion of the boundary of a node is interesting due to the observation that for any pair of nodes s and t, if their PSPTs have non-empty intersection then the boundary of s and the PSPT of t have a non-empty intersection.

We now discuss how the boundary of a node can be used to avoid unnecessary PSPT intersection checks. Consider a pair of nodes s and t. Let w be the closest node to s such that w belongs to the PSPT of s and the boundary of t. We make two observations. First, if $d(t, w) = r_t - 1$, we have a path of length at most $d(s, w) + r_t - 1$ between s and t and this is in fact the shortest path; to see this, note that for any other node w' which belongs to the PSPT of s and the boundary of t, $d(s, w') \geq d(s, w)$ and $d(t, w')$ is either $r_t - 1$ or r_t. Second, for all nodes w' at distance greater than $d(s, w)$ from s, a path between s and t via w' has to have a length of at least $d(s, w') + r_t - 1 \geq d(s, w) + r_t$. Hence, it suffices to check PSPT intersection for all the nodes in the PSPT of s that are at distance $d(s, w)$ from s, where w is the first node that lies in the intersection of PSPT of s and boundary of t.

Using the above conditions as stopping criteria for PSPT intersection checks, we now modify the technique of §6.2 to quickly compute shortest paths on unweighted networks. Our data structure will store three pieces of information:

- for each node of degree 1, a pointer to its neighbor and the distance to this neighbor;
- for each node u of degree greater than 1, a hash table storing the exact distance to each node in the *boundary* of u;
- for each node u of degree greater than 1, an array storing all nodes in the PSPT of u sorted in increasing distance from u; for each node w in the PSPT of u, this array stores the identifier of w, its distance to u and the identifier of the first node along the shortest path from w to u.

Given the above data structure, our query algorithm works as follows (see Algorithm 9). Assume that both nodes are of degree greater than 1; then, the algorithm returns the exact distance within two hash table lookups if either node is contained in the *boundary* of the other node. If such is not the case, the algorithm goes through each node w in the PSPT of s (in order of increasing distance from s by scanning the array corresponding to the PSPT of s from left to right) and checks if w is contained in the boundary of t. The first instance such a node w is found, we return the shortest path if w is at distance $r_t - 1$ from t; else, we iterate through all nodes at distance $d(s, w)$ from s and return the shortest path among all these nodes. As earlier, if either of the nodes is of degree 1, we replace the node by its neighbor and add the corresponding distance in the result.

Extension to retrieve paths. As in §6.2, Algorithm 9 can be easily extended to compute the corresponding paths. Once the node w_0 that lies along the shortest path between s and t is identified (the one that corresponds to the returned distance δ), the subpaths $w_0 \rightsquigarrow s$ and $w_0 \rightsquigarrow t$ can be computed by following the series of next-hops starting w_0 until s and t, respectively. As earlier, note that both of these subpaths are completely contained within the PSPT of s and t, respectively. The path from s to t is then returned by concatenating the two subpaths.

Algorithm 9 QueryUSP(s,t) — algorithm for computing the exact distance between nodes s and t in unweighted networks. $\mathcal{B}(s)$ and $\mathcal{B}(t)$ denote the boundary of nodes s and t respectively.

```
If deg(s) = 1
    return d(s, N(s)) + QueryUSP(N(s), t)
If deg(t) = 1
    return d(t, N(t)) + QueryUSP(s, N(t))
If s ∈ B(t) OR t ∈ B(s)
    Return d(s, t)
δ ← ∞
For i = 0 to r_s
    For each node w ∈ PSPT of s at distance i from s
        If w ∈ B(t)
            If d(t, w) = r_t − 1
                return d(s, w) + d(t, w)
            Else
                δ = min{δ, d(s, w) + d(t, w)}
    If δ < ∞
        return δ
```

6.4 A Distributed Implementation

ASAP, as presented in the previous sections, computes the shortest path between a given node pair in tens of microseconds. In this section, we show how to implement ASAP in a distributed fashion. This enables ASAP to answer batch shortest path queries without replicating the entire data structure along multiple machines which may be useful for applications with high workload. We will also discuss how to exploit the functionalities offered by distributed programming frameworks like MapReduce [98] and Pregel [99] for an efficient distributed implementation of ASAP.

Recall that the data structure of the previous section stores, for each node u in the network[1], the exact distance to each node in the PSPT of u; in other words, the data structure stores $\alpha\sqrt{n}$ triplets of the form $\langle u,(w,d(u,w))\rangle$, each corresponding to some node w in the PSPT of u. In the following description, we assume that each node u is assigned a machine in the cluster (for instance, using a hash function) and all the triplets corresponding to u are stored on that machine; it is rather trivial to extend ASAP to the case when the triplets for a single node u are split across machines.

[1] In this section, we assume that all nodes are of degree greater than 1.

Algorithm 10 A distributed implementation of ASAP; the algorithm computes the shortest distance between all node pairs in a set Q.

STEP 1 (AT EACH MACHINE):
Input: triplets for a subset of nodes $S \subset V$
For each node $u \in S \cap Q$
    For each w in PSPT of u
        Output $\langle \text{key}, \text{value} \rangle = \langle w; (u, d(u,w)) \rangle$

STEP 2 (AT MACHINE ASSIGNED KEY w):
Input: All $\langle \text{key}, \text{value} \rangle$ pairs with w as the key
For each pair of values $(u, d(u,w))$ and $(v, d(v,w))$
    Output $\langle \text{key}, \text{value} \rangle = \langle (u,v); d(u,w) + d(v,w) \rangle$

STEP 3 (AT MACHINE ASSIGNED KEY (u,v)):
Input: All $\langle \text{key}, \text{value} \rangle$ pairs with (u,v) as the key
Output the minimum of all the values received

A distributed implementation of ASAP is formally described in Algorithm 10. We explain the algorithm for a particular pair of nodes s and t. We start the query process by sending the query to the machines that store the triplets for nodes s and t. In the first step, the machine storing triplets for node s outputs, for each node w in the PSPT of s, a $\langle \text{key}, \text{value} \rangle$ pair with w being the key and with $(s, d(s,w))$ being the value; we denote this $\langle \text{key}, \text{value} \rangle$ pair as $\langle w; (s, d(s,w)) \rangle$. The machine storing triplets for node t does the same. In the next step, **the algorithm implements PSPT intersection in a distributed fashion**. Specifically, each distinct key is assigned to one machine and all values associated with that key (from any machine) are transferred to that machine. Note that for any key w, if the machine assigned key w receives two values corresponding to nodes s and t, then node w must belong to the PSPTs of both s and t and hence in the intersection of the two PSPTs; hence, there must be a candidate path of length $d(s,w) + d(t,w)$ between s and t — all such candidate paths constitute the output of the second step. As long as the PSPTs intersect along the shortest path, one of these paths (precisely, the path of shortest length) must be the shortest path between s and t; the final step computes this path by finding the minimum over all the paths output by machines in the second step.

Extension for retrieving shortest paths. Let $P(u, v)$ denote the shortest path between any pair of nodes u and v. To extend Algorithm 10 to retrieve the shortest path, we use the trick from §6.2.4 that allows computing the path from s to any node w in the PSPT of s. Algorithm 10 can then be modified to return the corresponding paths by simply appending the path information in Step 1. In particular, rather than having values of the form $\langle w; (s, d(s, w))\rangle$, we use values of the form $\langle w; (s, P(s, w), d(s, w))\rangle$. The machines in Step 2 simply concatenate the paths $P(s, w)$ and $P(t, w)$ to return the corresponding path $P(s, t)$.

Implementing on MapReduce. We now show how to implement Algorithm 10 on MapReduce using two rounds of operations. The first and the second steps of the algorithm form the Map and the Reduce steps of the first round. The outputs of the second step can be written to the Hadoop distributed file system (HDFS) and can be fed to the mapper in the next round. To implement our algorithm, the mapper of the second round will be an identity function — it simply outputs all $\langle \text{key}, \text{value}\rangle$ pairs as read; finally, the step three forms the reducer step of the second round.

Memory requirements for a distributed implementation. Let p be the total number of machines in the cluster. We now argue that ASAP requires each machine to store at most $\alpha n\sqrt{n}/p$ entries. We start by noting that since the data structure is distributed across the set of machines, the memory required at any single machine in the first step is simply a factor $1/p$ when compared to a single machine implementation. In the second step, each node w can be in the PSPT of at most n nodes, and hence each machine requires storing n entries. In the last step, each machine requires storing exactly one entry (to keep track of the shortest path seen so far). For the bandwidth requirements, we note that ASAP transfers $\alpha\sqrt{n}$ entries corresponding to each node that participates in the query.

Bandwidth requirements for distributed implementation. For the bandwidth requirements, we note that ASAP transfers $\alpha\sqrt{n}$ entries corresponding to each node that participates in the query. In particular, to compute distances between all pair of nodes in some set Q, ASAP transfers $|Q| \times \alpha\sqrt{n}$ entries.

6.5 Evaluation

In this section, we evaluate the performance of ASAP over several real-world datasets. We start by describing the datasets and experimental setup (§6.5.1). We then discuss several properties of node PSPTs (§6.5.2). Finally, we discuss the performance of ASAP for weighted networks (§6.5.3), for unweighted networks (§6.5.4) and for distributed implementation (§6.5.5).

6.5.1 Datasets and Experimental Setup

The datasets used in our experiments are shown in Table 6.1. The DBLP dataset is from [100]; the LiveJournal dataset is from [101] and the rest of the datasets are from [12].

For each dataset, we sampled 1000 nodes uniform randomly per experiment and repeated the experiment 10 times. The results presented below are, hence, for $10,000$ unbiased samples for nodes and 10 million unbiased samples for node pairs.

6.5.2 Properties of PSPTs

We start by empirically studying several interesting properties of node PSPTs with the goal of explaining our specific definition of node PSPTs and of choosing the size of node PSPTs to be $4\sqrt{n}$. To do so, for each dataset, we constructed node PSPTs of size $\alpha \cdot \sqrt{n}$ for α varying from $1/16$ to 32 in steps of multiplicative factor 2.

Table 6.1: Social network datasets used in evaluation.

Topologies	# Nodes (Million)	% Nodes with degree ≤ 1	# Edges (Million)
DBLP	0.71	13.77%	2.51
Flickr	1.72	50.95%	15.56
Orkut	3.07	2.21%	117.19
LiveJournal	4.85	21.83%	42.85

PSPT intersection

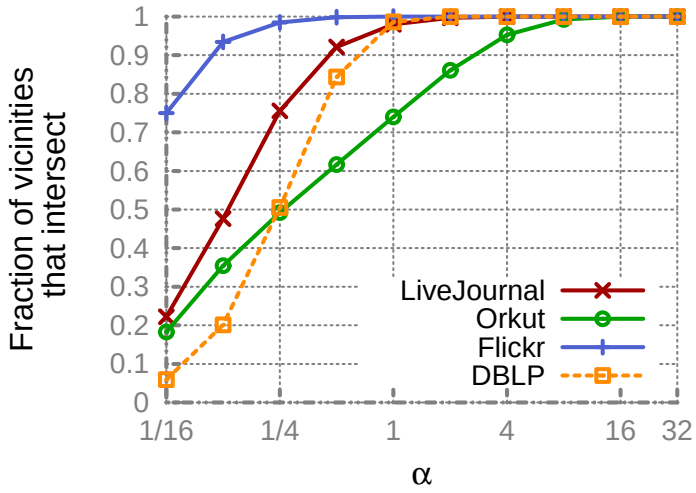
ASAP builds upon the idea of PSPT intersection to quickly compute shortest paths. Recall that the PSPTs of a pair of nodes s, t are said to intersect if there is a node w that is contained in both the PSPT of s and the PSPT of t. We evaluate, using the setup described in §6.5.1, the fraction of pairs of nodes that have intersecting PSPTs in real-world datasets for PSPTs of varying size. Figure 6.3(a) and Table 6.2 show the variation of fraction of PSPT intersections with size of varying PSPTs. Note that (with the exception of the Orkut network) for PSPTs of size $4\sqrt{n}$ and larger, the PSPTs of any two randomly selected nodes intersect with an extremely high probability (more than 0.9998). In fact, for all datasets, PSPTs of size $16\sqrt{n}$ intersect for *each* source-destination pair. We note that the Orkut network has a significantly different structure — it has an extremely high average degree (up to 11× larger than other networks), has very few degree-1 nodes — and yet, shows trends similar to other networks.

We remark that the results on PSPT intersection shown in Figure 6.3(a) and Table 6.2 are agnostic to whether the underlying network is weighted or unweighted. Furthermore, these results also apply to the distributed implementation. Specifically, the only difference between the algorithms for weighted and unweighted networks, and for single machine and distributed implementation was for the case of query time; the PSPT construction remains the same across all these algorithms and implementations.

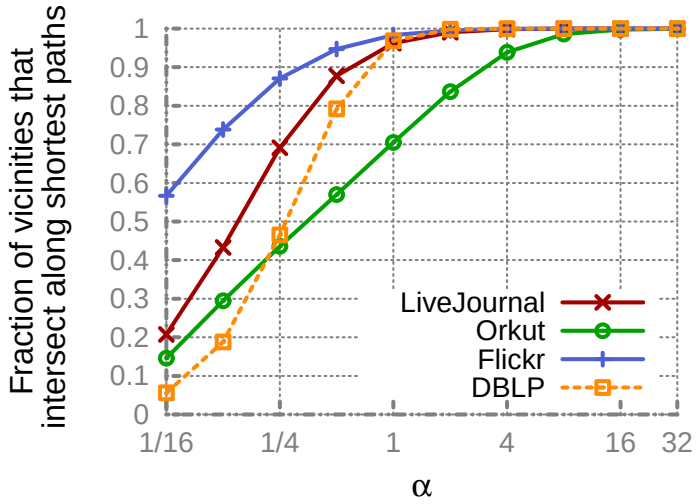
PSPT intersection along shortest paths

Social networks exhibit a structure much stronger than a large fraction of PSPTs merely intersecting. In particular, empirically, for PSPTs of size $4\sqrt{n}$ and larger, not only do the PSPTs of almost all pairs of nodes intersect, the intersection occurs along the shortest path.

It is easy to see that our definition of the PSPT of a node, due to use of tie-breaking, does not guarantee that for any given pair of nodes, the intersection of PSPTs, if any, occurs along the shortest path. Although our definition of a node PSPT does not guarantee intersection along the shortest path, real-world social networks do exhibit this property for PSPTs of size $4\sqrt{n}$ and larger.



(a) For PSPTs of size $4\sqrt{n}$ and larger, any pair of PSPTs intersects with extremely high probability.



(b) For PSPTs of size $4\sqrt{n}$ and larger, PSPTs almost always intersect along the shortest path.

Figure 6.3: Fraction of PSPTs of size $\alpha\sqrt{n}$ that intersect: (a) overall; (b) along the shortest path. For precise values for $\alpha = 2, 4$ and 8, see Table 6.2.

Table 6.2: Precise numbers (approximated to four decimal places) for Figure 6.3 for PSPTs of size $\alpha\sqrt{n}$ for $\alpha = 2, 4$ and 8. For PSPTs of size $4\sqrt{n}$ and larger, almost all PSPT pairs intersect along the shortest path.

Dataset	Fraction of interesting PSPTs					
	$\alpha = 2$		$\alpha = 4$		$\alpha = 8$	
	total	along SP	total	along SP	total	along SP
DBLP	0.9999	0.9986	1	1.0000	1	1
Flickr	1	0.9951	1	0.9993	1	1.0000
Orkut	0.8611	0.8366	0.9530	0.9386	0.9927	0.9859
LiveJournal	0.9967	0.9905	0.9998	0.9983	1.0000	0.9998

Figure 6.3(b) shows that for PSPTs of size $4\sqrt{n}$, most pairs of nodes have PSPTs intersecting along the shortest path. More interestingly, comparing results of Figure 6.3(a) and Figure 6.3(b) (also see Table 6.2), we note that for PSPTs of size $4\sqrt{n}$ and larger, whenever the PSPTs intersect, they almost always intersect along the shortest path.

For node pairs whose PSPTs intersect but not along the shortest path, we will discuss later that the length of the path via node along which the PSPTs intersect is "not too long" when compared to the shortest path. In addition, a non-trivial lower bound can be proved on the distance between node pairs whose PSPTs do not intersect. These observations may be interesting for applications that do not necessarily require computing shortest paths and that require computing shortest path only for pairs of nodes that are "close enough".

Note that similar to the case of PSPT intersection, the results shown in Figure 6.3(b) and Table 6.2 for PSPT intersection along the shortest path apply for each of weighted networks, unweighted networks, single-machine implementation and distributed implementation.

Benefits of Consistent Tie Breaking

Recall that our definition of node PSPT (§6.2.1) requires that ties be broken lexicographically using the unique node identifiers. We now elaborate on the significance of this tie breaking scheme. Figure 6.4 compares the performance of our tie breaking scheme with that of an arbitrary tie breaking scheme as in standard implementations of shortest path algorithms; for our experiments, we used the implementation provided by the Lemon graph library [102].

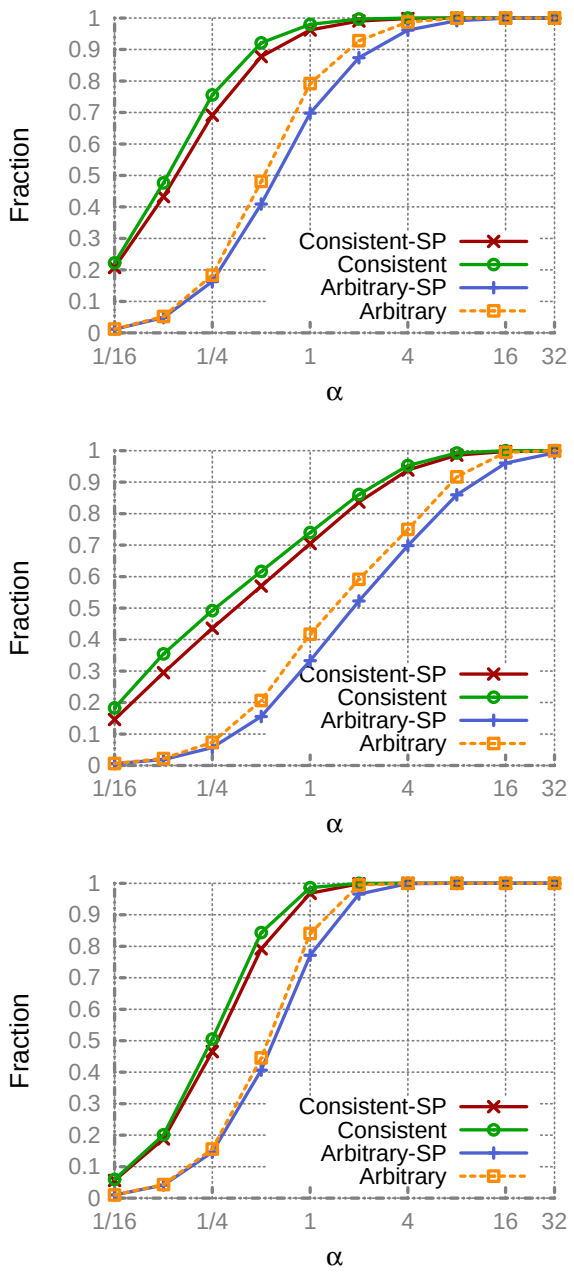


Figure 6.4: Comparison of the tie breaking scheme of §6.2.1 to arbitrary tie breaking (see §6.5.2) for LiveJournal (top), Orkut (middle) and DBLP (bottom). Consistent-SP and Consistent are for PSPT intersections along shortest path and overall for the tie breaking scheme of §6.2.1; Arbitrary-SP and Arbitrary are corresponding ones for the arbitrary tie breaking scheme.

We observe that when the PSPT sizes are rather small, consistent tie breaking can significantly increase the fraction of PSPTs that intersect along the shortest path. For the LiveJournal network and for PSPTs of size $\sqrt{n}/4$, for instance, consistent tie breaking has 57% more PSPT intersections along the shortest path when compared to arbitrary tie breaking. For moderate PSPT size (those of our interests), the consistent tie breaking has smaller but noticeable effect — for PSPTs of size $4\sqrt{n}$, consistent tie breaking leads to an additional fraction 0.24 PSPT intersections along the shortest path for the Orkut network.

6.5.3 Performance of ASAP for weighted networks

We now evaluate ASAP for the case of weighted networks in terms of preprocessing time, memory requirements, accuracy of computing paths and most importantly, the time taken to compute paths. We discuss the results for three specific points of interest — $\alpha = 2, 4$ and 8 (that is, PSPTs of size $2\sqrt{n}, 4\sqrt{n}$ and $8\sqrt{n}$, respectively); these are the values which provide the most interesting trade-offs between memory, latency and accuracy for ASAP and are of practical interest.

Preprocessing and Memory

We start by evaluating ASAP in terms of the time taken to construct the data structure and the memory requirements of the resulting data structure. Note that the preprocessing and memory requirements of ASAP are independent of whether one wants to compute a single shortest path or multiple short paths between a given pair of nodes.

Preprocessing time. The results on time taken to construct our data structure for various networks are shown in Table 6.3. Note that, as expected, the preprocessing time increases with the size of the PSPTs, the size of the network and more importantly, the average degree of the network. **We observe that it is rather easy to distribute the computations required for constructing the data structure across multiple machines** — each machine can compute the PSPT for a fraction of nodes in the network. For instance, using just 10 dual-

Table 6.3: Average preprocessing time & memory requirements for ASAP (approximated to one decimal place).

Dataset	Preprocessing time (ms per node)			Memory Requirements (kB per node)		
	$\alpha = 2$	$\alpha = 4$	$\alpha = 8$	$\alpha = 2$	$\alpha = 4$	$\alpha = 8$
DBLP	6.9	13.6	28.2	10.4	20.9	41.9
Flickr	75.8	149.9	278.7	8.9	18.2	37.4
Orkut	130.7	298.3	638.3	29.4	58.8	117.8
LiveJournal	56.0	113.2	237.1	28.4	57.0	114.7

core machines, we can construct the data structure for the LiveJournal and the Orkut networks in less than 8 hours and 13 hours, respectively. This is comparable or faster than the preprocessing time of recent shortest path computation heuristics [103, 104] and significantly faster than techniques that allow computing approximate distances and paths (based on the evaluations in [94]). However, the former set of techniques are limited to computing a single path between any given pair of nodes, have higher latency compared to ASAP and do not admit efficient distributed implementation.

Memory requirements. ASAP requires storing, for each node with degree greater than 1, an array with $\alpha\sqrt{n}$ entries. Table 6.3 shows the average memory requirements per node for ASAP (if only distances need be retrieved, the memory requirements reduce by roughly 33%). We note that the memory requirements of ASAP, although far from ideal, are much lower than the data structures usually maintained by social networks for answering various user queries. Alternative shortest path computation techniques [103, 104] require slightly lower memory in practice but, unlike ASAP, do not provide any guarantees. In addition, unlike ASAP, these techniques will require significantly higher memory for computing multiple paths between users. Moreover, some of the previous techniques [104] require a hash table based implementation and hence, have extremely high overhead if the data structure is stored in memory; ASAP, on the other hand, employs an array based implementation and hence, has much lower overhead. Our own experiments suggest that arrays require 3-24× less memory than hash tables; hence, an in-memory implementation of ASAP would require memory comparable to that of techniques.

Accuracy

In terms of accuracy, we make three observations. First, since Algorithm 8 iterates through all the nodes in the PSPT of the source to check for PSPT intersection, ASAP returns the shortest path as long as the PSPTs intersect along the shortest path. From Table 6.2, this happens for 99.83% of the node pairs for PSPTs of size $4\sqrt{n}$; of course, a higher accuracy of 99.98% can be achieved by using PSPTs of size $8\sqrt{n}$. Second, out of the remaining 0.171% of the node pairs, ASAP returns at least one path for 0.150% of the pairs since their PSPTs intersect; for these node pairs, ASAP provides the following guarantee: if the distance between the nodes is $d(s,t)$, the distance returned by the algorithm is $d(s,t) + W_{\max}$, where $W_{\max}$ is the weight of the heaviest edge incident on nodes in the PSPT of the source (for networks modeled as unweighted graphs, $W_{\max} = 1$). Finally, for the remaining 0.021% of the node pairs, it is possible to combine ASAP with those for computing exact [91, 92] paths; however, it may just be easier to just store shortest paths between such a small fraction of node pairs (as and when they are computed).

Query latency

Finally, we discuss the results for query time. Our implementation stores the PSPTs of nodes in-memory using a standard C++ array implementation. The implementation runs on a single core of a Core i7-980X, 3.33 GHz processor running Ubuntu 12.10 with Linux kernel 3.5.0-19. Table 6.4 presents the query time of ASAP for shortest path computation (for $\alpha = 2, 4$ and 8); Table 6.5 compares the results in Table 6.4 with that of an optimized implementation of breadth-first search algorithm and bidirectional breadth-first search algorithm [92] using the experimental setup of §6.5.1. Note that we compare the performance of ASAP with breadth-first search simply to demonstrate that *even* for unweighted networks, ASAP provides significant speed-ups; the relative performance of ASAP will be much better in comparison with a shortest path algorithm for weighted networks since the query time of ASAP is independent of whether the network is weighted or not.

We make several observations. First, rather surprisingly, ASAP is at least 4-5 orders of magnitude faster than the current fastest known technique for computing paths of extremely low error (that is, error less than 10%; the

Table 6.4: Query time results (approximated to three decimal places) for ASAP algorithm for weighted networks from §6.2.

Dataset	Our technique Time (in μs)		
	$\alpha = 2$	$\alpha = 4$	$\alpha = 8$
DBLP	9.721	20.325	41.827
Flickr	12.967	26.474	51.974
Orkut	15.686	31.756	64.968
LiveJournal	24.072	48.938	100.197

Table 6.5: Query time results (approximated to three decimal places) for ASAP algorithm for weighted networks from §6.2.

Dataset	BFS Time (in ms)	Bidirectional BFS Time (in ms)	Speed-up (compared to Bidirectional BFS)
DBLP	327.2	18.614	$445 - 1915\times$
Flickr	2090.2	83.956	$1615 - 6475\times$
Orkut	28678.5	760.987	$11713 - 48514\times$
LiveJournal	6887.2	156.443	$1561 - 6499\times$

Table 6.6: Results on query time and number of paths for computing multiple paths using ASAP. LJ refers to the LiveJournal network.

Dataset	$\alpha = 2$		$\alpha = 4$		$\alpha = 8$	
	Time (μs)	#Paths	Time (μs)	#Paths	Time (μs)	#Paths
DBLP	15.080	62	31.568	173	68.841	416
Flickr	23.672	276	51.974	523	83.956	1762
Orkut	22.250	81	64.968	237	760.987	817
LJ	34.462	115	99.197	453	156.443	1141

current fastest implementation [94] for computing low error paths requires at least 1090 ms and can be up to 2751 ms for the Orkut network). Second, the latency of ASAP for single path computation is $2-5\times$ lower than the techniques that compute the exact shortest path [103, 104]. Note, however, that ASAP achieves this speed-up at the cost of slight loss in accuracy; we believe that such loss in accuracy is completely acceptable for most real-world applications as long as we achieve a speed-up for most of the queries.

Not only does ASAP require lower latency for single shortest path computation, its most significant advantage is that it enables computing a large number of paths between any given node pair in less than 100 μs (see Table 6.6). Consider the LiveJournal network for instance. ASAP computes 453 paths, on an average, between a given pair of nodes in roughly 99 μs, hence enabling a plethora of new social network applications. We are not aware of any other technique that can compute multiple paths between node pairs in time comparable to ASAP.

6.5.4 Performance of ASAP for unweighted networks

We now evaluate the performance of ASAP assuming that the underlying network topologies are unweighted. We start with preprocessing time, memory requirements and accuracy. We then discuss results on query latency.

Preprocessing time, Memory requirements and Accuracy. The preprocessing time for the ASAP algorithm for unweighted networks is the same as that for weighted networks — the hash table containing the boundary nodes and the list containing nodes in the PSPT in order of increasing distance can be generated while computing the PSPT.

In terms of memory, we note that the above technique requires at most twice the memory of the technique of §6.2 since in the worst-case, we now need to store all the nodes in the PSPT twice — once in the form of a hash table for storing boundary nodes and another in the form of an array storing nodes in the PSPT in order of increasing distance. However, being a hash-table based implementation, the ASAP technique for unweighted networks may require more space for in-memory implementations.

Table 6.7: Query time results (approximated to three decimal places) for the ASAP algorithm for unweighted networks from §6.3.

Dataset	Our technique		
	Time (in μs)		
	$\alpha = 2$	$\alpha = 4$	$\alpha = 8$
DBLP	15.237	12.139	8.362
Flickr	11.355	10.028	6.488
Orkut	69.2315	94.200	117.761
LiveJournal	40.566	40.830	37.146

Regarding accuracy, we remark that the accuracy of Algorithm 9 is *exactly* the same as that of Algorithm 8; that is, for the LiveJournal network and with PSPTs of size $4\sqrt{n}$, the algorithm returns the shortest path for 99.829%, a path of length $d(s,t)+1$ for 0.150% and no path for the remaining 0.021% of the source-destination pairs. Intuitively, the two conditions used in our stopping criteria ensure that if the PSPTs intersect along the shortest path, then the algorithm indeed finds the shortest path; also, since we stop only after we have found an intersection (if at all the PSPTs intersect), we guarantee to find a path of length at most $d(s,t)+1$ as long as the PSPTs intersect.

Query latency. We now discuss our results for query time. Table 6.8 presents the query time of ASAP for unweighted networks; Table 6.7 compares the results of Table 6.8 with that of an optimized implementation of breadth-first search and bidirectional breadth-first search using exactly the same setup as in §6.2. We make two observations. First, unlike the case of weighted networks, PSPTs of larger size may lead to lower query latency; intuitively, as the size of PSPTs grow, fewer checks are required before the first intersection is found and the algorithm needs to perform only a few more checks after the first intersection is found. The second observation is that by using the two conditions to avoid unnecessary checks, the performance improvement can be significant — for the LiveJournal network, the latency reduces by 4.8× for PSPTs of size $4\sqrt{n}$ and by more than 10× for PSPTs of size $8\sqrt{n}$, without any loss in accuracy; for the Flickr network, the reduction in latency is roughly 30× for PSPTs of size $8\sqrt{n}$.

Table 6.8: Query time results (approximated to three decimal places) for the ASAP algorithm for unweighted networks from §6.3.

Dataset	BFS Time (in ms)	Bidirectional BFS Time (in ms)	Speed-up (compared to Bidirectional BFS)
DBLP	327.2	18.614	$1221 - 2226\times$
Flickr	2090.2	83.956	$7393 - 12940\times$
Orkut	28678.5	760.987	$6462 - 10992\times$
LiveJournal	6887.2	156.443	$3856 - 4211\times$

Table 6.9: Per-query latency in microseconds amortized over all node pairs using the experimental setup of §6.5.1 and an external memory (data residing on HDFS and read by mappers) MapReduce implementation with 20 and 40 mappers and reducers. LJ refers to the LiveJournal social network.

Dataset	20 Mappers & Reducers			40 Mappers & Reducers		
	Map	Shuffle +Reduce	Total	Map	Shuffle +Reduce	Total
DBLP	33.574	34.866	68.440	19.370	25.826	45.196
Flickr	140.051	225.640	365.690	73.916	143.941	217.857
Orkut	134.462	53.785	188.247	68.725	29.880	98.605
LJ	255.796	99.561	355.357	128.664	59.737	188.401

6.5.5 Performance of Distributed ASAP Implementation

Finally, we evaluate the performance of distributed ASAP using an external memory MapReduce implementation, that is, the data is residing on the HDFS and is read when queries are initiated. The implementation runs on a 64 node MapReduce cluster with Hadoop version 0.19.1; each node in the cluster supports up to 4 mappers and 4 reducers. The evaluation results are shown in Table 6.9 (with ASAP running atop 20 and 40 mappers and reducers) and Table 6.10 (with ASAP running atop 80 mappers and reducers).

We make several observations. First, we note that since distributed ASAP requires significantly less memory when compared to a single machine implementation of ASAP, it is entirely feasible to do an in-memory implementation of distributed ASAP (our cluster does not provide support for this); for in-

Table 6.10: Per-query latency in microseconds amortized over all node pairs using the experimental setup of §6.5.1 and an external memory (data residing on HDFS and read by mappers) MapReduce implementation with 80 mappers and reducers.

Dataset	80 Mappers & Reducers		
	Map	Shuffle +Reduce	Total
DBLP	11.622	19.370	30.992
Flickr	38.903	105.038	143.941
Orkut	35.856	20.916	56.772
LiveJournal	67.395	41.356	108.751

stance, for the LiveJournal network, using 40 machines require roughly 6.5 GB of memory which is feasible for most modern desktops. With such an in-memory implementation, the corresponding query latency will simply be the time consumed in the shuffle and reduce phase and hence less than 100 μs for most networks. Second, even with an external memory implementation, the amortized query time is less than 366 μs and most of this is spent reading the input data from HDFS.[2] Finally, we note that the amortized query latency (and memory requirements!) of distributed ASAP reduces almost linearly with increase in the number of machines, which is a highly desirable property of distributed implementations.

6.6 Related Work

Our goals are related to two key areas of related work: algorithms and heuristics for computing shortest paths and for computing approximately shortest paths.

[2]We note that the mappers in our distributed algorithm perform extremely simple tasks; hence, the extremely high time of mapper operations (255 μs for the LiveJournal network with 20 mappers and reducers) is due to slow hard disks and issues with the filesystem. The same files can be read on a single machine using our Ubuntu based implementation in amortized time of less than 12 μs.

TEDI and Pruned Landmark Labeling. TEDI [104] is one of the most closely related works to ASAP. Independently of our work, a recent paper entitled "Pruned Landmark Labeling" [103] also computes shortest paths in large social networks. The preprocessing time of [104] is significantly larger than that of ASAP; [103], on the other hand, requires lower preprocessing time. However, as discussed in §6.5, the preprocessing stage can be easily parallelized across multiple machines and hence, is not a bottleneck for ASAP.[3] In terms of memory and accuracy, the focus of [103, 104] was on providing 100% accuracy (shortest paths for all node pairs) without providing any bounds on memory requirements, although their memory footprint is lower than that of ASAP. ASAP, on the other hand, provides guarantees of the memory requirements with slight loss in accuracy. Either of these trade-offs may be interesting depending on the application. **The main advantage of ASAP is its lower latency, its ability to compute multiple paths and its ability to answer batch queries using a simple distributed implementation**. None of [103, 104] achieve any of the last two properties.

Shortest path algorithms and heuristics. Heuristics like $A^\star$ search [91, 92] and bidirectional search [92] have been proposed to overcome the latency problems with traditional algorithms for computing shortest paths. The approaches in [91, 92], although useful in reducing the query time, still require running a (modified) shortest path algorithm for each query and do not meet the latency requirements. For instance, the experimental results in §6.5 show that bidirectional search can take hundreds of milliseconds to compute the shortest paths even on moderate size networks.

In comparison to [91, 92], our contributions are two-fold: first, we show that empirically, in social networks, PSPT of size $4\sqrt{n}$ nearly always intersect (heuristics in [91, 92] could also exploit this); and second, we argue that the PSPT being a small fraction of the entire network, storing and checking intersection quickly is feasible. This should be substantially faster than traditional bidirectional search [92] because it is just a series of hash table look-ups in a relatively compact data structure with one element per PSPT node — as opposed to running a shortest path algorithm that would require priority queue operations, and may even explore a large fraction of the entire network.

[3] Note that it is not clear how to parallelize the preprocessing stage of [104].

Approximation algorithms. Arguing that the above heuristics [91, 92] are unlikely to meet the stringent latency requirements of social network applications, [25–27, 94, 95, 105] focus on computing *approximate* distances and paths. The body of work can be broadly characterized into two categories.

The first category uses techniques from graph embedding literature [25,26]. The main advantage of these schemes is their low memory footprint; however, these schemes often compute paths of high worst-case stretch (providing a guarantee of $\log(n)$ stretch for a network with n nodes) [25, 26], and are often not able to compute shortest paths [25]. ASAP, on the other hand provides latency similar to the above techniques while providing the benefits of computing *shortest distances and paths* for most source-destination pairs.

The second category uses techniques from distance oracle literature [27,94, 95, 105]. In comparison to these techniques, ASAP differs in several aspects. First, the above techniques are primarily modifications or heuristic improvements on results from theoretical computer science [41]; these results are now known to be far from optimal for real-world networks [48,51,76] which ASAP borrows ideas from. Second, techniques in this category that have lowest latency [95] return paths that have high absolute error (more than 3 hops on an average, even on small networks); in comparison, ASAP computes shortest paths between almost all source-destination pairs. On the other hand, techniques that provide significantly better accuracy require 4-5 orders of magnitude higher query time when compared to ASAP [94, 105]. Finally, similar to graph embedding based techniques, some of these techniques [27] are unable to compute the actual paths.

Chapter 7

ShapeShifter: Shortest Paths on Dynamic Graphs

Computing distances and paths is a fundamental primitive in social network analysis. In Chapter 6, we described the design and implementation of ASAP—a system that computes, on networks with millions of nodes and edges, short paths in tens of microseconds. The design and implementation of ASAP was motivated by several concrete applications in §1.1.2, including social search and ranking, socially-sensitive and location-aware search, and social auctions.

However, an assumption made by ASAP was that the underlying network topology does not change with time. This is rarely the case in real-world social networks where user-user links are inserted and/or deleted over time; ASAP, for such dynamic networks, would require hundreds of milliseconds of update time upon each update in the network. Applications that require distance and path computations on dynamic networks can significantly benefit by techniques that achieve the same performance as ASAP, but also allow handling network updates more efficiently. The problem of efficiently computing paths on dynamic networks is the focus of this chapter.

Scalable distance computation on dynamic graphs is challenging for two reasons. First, the obvious solution of storing just the input graph, and running a shortest path algorithm like $A^\star$ search [91, 92] or bidirectional search [92] for each query does not work either. This is due to the fact that the applications above compute paths in response to a user query and hence have rather stringent latency requirements [38]. As discussed in §6.5, these algorithms require hundreds of milliseconds even on moderate size networks.

On the other hand, techniques based on metric embedding [25, 26], tree decomposition [104], landmark-based distance computation [27, 94, 95] and 2-hop cover [103] do not work either — even a single edge insertion and/or deletion may result in distorting the previous embedding, tree decomposition, landmark distances, and covers computed in these techniques. To the best of our understanding, all these techniques require reconstructing the data struc-

ture from scratch since it is hard to precompute and store the subset of data structure that needs to be updated for each possible choice of edge update.

Note that there is a natural tussle between update time and query time. On the one hand, the obvious solution of precomputing and storing distances to vertices that are "close" (or "famous", or "high-degree") has extremely low query time (simple hash table lookups when distances are stored) but high update time (a single edge insertion and/or deletion may render all distances distorted in unpredictable ways). On the other hand, another obvious solution of just storing the input graph and running a fast shortest path algorithm has extremely low update time (just insertion/deletion of a new edge) but very high query time (running a shortest path algorithm on a massive graph).

7.1 Overview and Contributions

This chapter presents ShapeShifter, a system that theoretically achieves each point on the trade-off space between update time and query time. In practice, however, ShapeShifter uses two techniques to perform significantly better than the trade-off. First, ShapeShifter precomputes and stores an inverted index which directly allows ShapeShifter to find the minimal set of updates required in the data structure upon an edge insertion and/or deletion. Second, ShapeShifter uses a set of heuristics to speed up the update process of the data structure by more than two orders of magnitude. As a result of these techniques, ShapeShifter is not only able to theoretically achieve each point in the above trade-off space, but in practice, allows handling *thousands of edge updates per second* using a single machine while *answering each distance query in less than a millisecond*.

ShapeShifter is designed to achieve three goals: (1) quickly updating the underlying data structures (referred to as *ShapeShifter sketch*, henceforth) upon an edge update; (2) quickly updating the inverse-sketch upon an edge update; and (3) quickly computing a short path between any given pair of nodes in the graph. A parameter α in ShapeShifter sketch allows to achieve, theoretically speaking, the trade-off between sketch size, update time and query time. However, as we will discuss in §7.4, setting $\alpha = 3n^{1/3}$ leads to the best operating point for most real-world applications

We now provide an overview of ShapeShifter performance. All the performance results in the following discussion are for a real-world social network, LiveJournal, that contains roughly 5 million nodes and 69 million edges. All the experiments were run on a 8-core Intel server running Ubuntu and the results are stated for a specific value of $\alpha = 3n^{1/3}$.

ShapeShifter Accuracy. ShapeShifter is not a tool for shortest path computations; there are, in fact, a few corner cases where ShapeShifter does not output the shortest path between a given pair of nodes (as discussed in §7.2). This is intentional by design; the problem of maintaining an index that always outputs the shortest path is, theoretically speaking, as hard as computing a new shortest path tree [106]. However, by avoiding the corner cases that make the problem theoretically hard, ShapeShifter is able to achieve low latency for both answering distance queries and for updating the sketch.

That said, the empirical performance of ShapeShifter suggests that it almost always outputs a shortest path between any given pair of nodes. For most topologies and for most update scenarios, ShapeShifter computes a path of stretch at most 1.1.

ShapeShifter Memory Footprint. The empirical performance of ShapeShifter suggests that for most practical applications (that is, for those requiring millisecond scale updates and query time), ShapeShifter requires storing a sketch that is at most 5× the size of the input graph. This is entirely acceptable as many modern systems keep indexes of hundreds of gigabytes of size in main memory. Moreover, it is possible to reduce the memory footprint of ShapeShifter at the expense of higher query time by choosing any desirable value of α. Finally, it is possible to implement ShapeShifter as a disk-residing (or, SSD-residing) sketch.

ShapeShifter Update Latency. ShapeShifter can handle thousands of edge updates per second using a single machine. Furthermore, ShapeShifter demonstrates a near-linear horizontal scalability with number of machines — using 10 machines for updates allows ShapeShifter to handle roughly 10× more edge insertions and deletions per second.

ShapeShifter Query Latency. ShapeShifter computes a path between any given pair of nodes in less than a millisecond. For most cases, the latency is in lower hundreds of microseconds. This compares favorably to techniques for computing approximate and exact shortest paths on massive graphs [27, 103, 104], which do not handle graph updates. ShapeShifter can also compute multiple paths between any given node pairs at the expense of a small overhead.

ShapeShifter Scalability and Theoretical Bounds. While ShapeShifter has good performance (in terms of memory footprint, update time and query time) on the evaluated topologies, it is natural to question about the expected performance of ShapeShifter for larger graphs. We answer this question upfront in terms of expected increase in memory footprint, update time and query time of ShapeShifter for the case of a billion node graph.

ShapeShifter builds upon a strong theoretical foundation, providing asymptotic bounds on scalability. The memory footprint of ShapeShifter scales as $n^{1/4}$, where n is the number of nodes in the graph. Hence, even with a billion node graph, the memory footprint of ShapeShifter would simply be 3.7× larger than the numbers reported in §7.4. Note that even for a billion node graph, $n^{1/4}$ is 6× larger than $\log_2(n)$; hence, the memory footprint of ShapeShifter compares favorably with techniques for approximate shortest path computation on static graphs that allow computing distances of worst-case stretch $\log_2(n)$ [95].

The update and query time of ShapeShifter scale with $\sqrt{n}$; hence, for a billion node graph, the update and query time would increase by 14×. We make two remarks, however. First, the bounds on ShapeShifter scalability are for worst-case graph topologies; as we show in §7.4, ShapeShifter empirically demonstrates significantly better scalability than the theoretical bounds. Second, with larger graphs, it is only natural to assume that an application running ShapeShifter would be provided with more resources (more cores, more machines and/or more memory); given the almost-linear horizontal scalability of ShapeShifter, the resulting increase in update and query time can be made negligible.

7.2 ShapeShifter Sketch

In this section, we describe the ShapeShifter sketch, the suite of data structures maintained by ShapeShifter for quick updates and for quick computation of distances. ShapeShifter maintains two kinds of sketches — a forward sketch and an inverse sketch. The former is used to quickly answer distance queries while the latter is used to quickly update the indexes upon insertion and/or deletion of edges. We describe these sketches in §7.2.1 and §7.2.2, respectively.

We need some more notation; this is described in Table 7.1. In addition, we will assume that the input graph $G = (V, E)$ is a connected, weighted, undirected graph, with V denoting the set of nodes in the graph and E denoting the set of edges in the graph.

Table 7.1: Additional notation used in the chapter.

α	a parameter $1 \leq \alpha \leq n$
$\mathcal{B}$	set of boundary nodes
$\mathcal{B}_s$	$\log(n)$ closest nodes of s in $\mathcal{B}$
$\beta(s)$	farthest boundary node in $\mathcal{B}_s$
PSPT (u)	PSPT of node u
iPSPT (u)	inverse-PSPT of node u

7.2.1 Forward Sketch

To succinctly describe our indexes, we will use the notion of **set of boundary nodes**, denoted by $\mathcal{B}$. Let $\mathcal{B}$ be a set of boundary nodes constructed by sampling each node with probability

$$\frac{1}{\alpha} \times \left\lceil \frac{n \log n * \deg(v)}{2m} \right\rceil$$

This ensures that $|\mathcal{B}|$ is $n \log n/\alpha$, in expectation, and that high degree nodes have a higher probability of being sampled.

Let $\mathcal{B}_s$ denote the set of $\log n$ closest boundary nodes of node s. Furthermore, let $\beta(s)$ be the farthest boundary node in $\mathcal{B}_s$ from s; that is, $d(s, \beta(s)) >= d(s, \beta)$ for every $\beta \in \mathcal{B}_s$.

The most basic component of ShapeShifter sketch is what we call a **Partial Shortest Path Tree**, denoted by **PSPT**. At a high level, the PSPT for ShapeShifter has a similar structure as the PSPT for ASAP from Chapter 6. However, we need some modifications so as to allow faster updates. We define the node PSPT for ShapeShifter as follows. The PSPT of a node s comprises all nodes in $\mathcal{B}_s$ and all nodes at distance strictly less than $d(s, \beta(s))$ from s after removing the degree-1 nodes from the graph. It is rather easy to prove that the size of the PSPT of any node is $(\alpha + \log(n))$, in expectation.

During the preprocessing phase, ShapeShifter computes and stores the following forward sketch:

- For each node u of degree 1, a hash table containing its neighbor $N(u)$ and the weight of the edge $(u, N(u))$
- For each node s of degree greater than 1, a hash table containing the distance to nodes in its PSPT. If paths need be computed, the hash table also stores for each node $w \in$ PSPT, the next-hop node along the shortest path from w to s.
- For each node s of degree greater than 1, a hash table containing $\mathcal{B}_s$ and the corresponding distances.

7.2.2 Inverse Sketch

During the preprocessing phase, ShapeShifter computes and stores the following inverse sketch:

- For each node s of degree 1, a hash table containing its neighbor $N(u)$.
- For each node s of degree greater than 1, a hash table containing the set of nodes w that contain s in their PSPT.

The above data structure is referred to as the **inverse-PSPT** of a node s. Essentially, for nodes with degree greater than 1, an inverse-PSPT contains the set of nodes w that contain s in their PSPT. For degree-1 nodes, the inverse-PSPT simply contains their respective neighbors.

7.3 ShapeShifting for Updates

We describe the update algorithm for ShapeShifter. We assume that an edge (u, v) is updated, where u or v may potentially be a new node or the update might simply be an edge weight update.

7.3.1 Edge Insertions

We describe how ShapeShifter reacts to edge insertions. The formal description of the algorithm is given in Algorithm 11; the algorithm uses two subroutines shown in Algorithm 12 and Algorithm 13, respectively. We informally describe the execution of the algorithm. Throughout the description, we assume that an edge (u, v) with weight w_{uv} is inserted in the graph and that v is the farther of the two nodes from s before the edge insertion, that is, $d(s, v) \geq d(s, u)$ before the edge insertion.

The algorithm exploits the following two observations. First, an edge update can only reduce the distance between any pair of nodes in the graph. The second observation is that if we let $\mathcal{B}_s$ and $\mathcal{B}_s$' be the set of $\log(n)$ closest boundary nodes before and after the edge insertion respectively, and let $\beta(s)$ and $\beta'(s)$ be the farthest nodes from s in $\mathcal{B}_s$ and $\mathcal{B}_s$', then $d(s, \beta'(s)) \leq d(s, \beta(s))$; that is, the radius of the PSPT can only shrink.

We get the following two corollaries from the above two observations. First, consider the case when both u and v belong to the PSPT of some node s before the edge is inserted. Then, if the inserted edge does not change $d(s, v)$ (in other words, if the inserted edge does not change the distance between u and v), the PSPT of s does not need to be updated; this is reflected in lines $7-9$ of Algorithm 11. Second, consider the case when v does not belong to the PSPT of some node s, but u does. In this case, since the radius of the PSPT of s can only shrink after edge insertion, the PSPT of s does not need to be updated if the new distance (after edge insertion) is still greater than $d(s, \beta(s))$; this is reflected in lines $16-18$ of Algorithm 11.

We now describe how ShapeShifter updates the PSPTs of vertex s. First, consider the case when v is not contained in PSPT of s before the edge insertion. The algorithm (lines $20-23$ and subroutine PSPT-Insert-Update) finds the set of nodes whose distance after edge insertion becomes less than $d(s, \beta(s))$ and initially includes all these nodes in the PSPT of s. Some of these nodes may

Algorithm 11 Updating the index upon **insertion** of edge (u, v)

Input: iPSPT(u), iPSPT(v)
Let $\mathcal{U} \leftarrow i\text{PSPT}(u) \cup i\text{PSPT}(v)$

$w_{uv} \leftarrow$ weight of the inserted edge
For each $s \in \mathcal{U}$
    If $u \in$ PSPT(s) *AND* $v \in$ PSPT(s)
        /* Let $d(s, v) > d(s, u)$, else switch u and v */
        $\delta \leftarrow d(s, v) - d(s, u)$
        If $\delta < w_{uv}$
            DO NOTHING
        Else
            $d(s, v) \leftarrow d(s, u) + w_{uv}$
            PSPT-Insert-Update$(v, d(s, v), \beta(s))$
            PSPT-Boundary-Update(s)
    Else
        /* Let $u \in$ PSPT(s), else switch u and v */
        $\delta \leftarrow w_{uv}$
        If $d(s, u) + \delta >= d(s, \beta(s))$
            DO NOTHING
        Else
            Insert $\langle v, d(s, u) + d(u, v)\rangle$ in PSPT(s)
            If $v \in \mathcal{B}$
                Insert $\langle v, d(s, u) + d(u, v)\rangle$ in $\mathcal{B}_s$
            PSPT-Insert-Update$(v, d(s, v), \beta(s))$
            PSPT-Boundary-Update(s)

already be contained in the PSPT of s; for instance, a node w in the PSPT of s may have a shorter path $s \rightsquigarrow v \rightsquigarrow w$ due to the inserted edge reducing its distance to s. A simple corollary is that $d(s, \beta(s))$ may change as well; this can happen for two reasons. First, some of the newly inserted nodes in the PSPT of s may be the sampled boundary nodes in $\mathcal{B}$ and have distance less than $d(s, \beta(s))$; and second, the distance $d(s, \beta(s))$ may reduce due to node $\beta(s)$ having a shorter path to s via v. The algorithm, in the next step, finds the new set of $\log(n)$ closest nodes in $\mathcal{B}_s$ and lets $\beta(s)$ to the farthest node in the new $\mathcal{B}_s$. Finally, all nodes in PSPT of s that are farther than this new $\beta(s)$ are deleted from the PSPT of s. The only difference for the case when v is contained in the PSPT of s before edge insertion is that the distance to v is updated directly.

Algorithm 12 PSPT-Insert-Update Subroutine

```
1: For each neighbor x of w
2:     If d(s,w) + d(w,x) < d(s,β(s))
3:         Insert ⟨x, d(s,w) + d(x,w)⟩ in PSPT(s)
4:         If x ∈ B
5:             Insert ⟨x, d(s,w) + d(x,w)⟩ in B_s
6:         PSPT-Insert-Update(x, d(s,x), β(s))
```

Algorithm 13 PSPT-Boundary-Update Subroutine

```
1: Let β(s) be the farthest boundary in B_s
2: For each node w in PSPT (s)
3:     If d(s,w) >= d(s,β(s))
4:         Delete w from PSPT (s)
```

7.3.2 Edge Deletions

We now describe how ShapeShifter reacts to edge deletions. Throughout the following discussion, we assume that an edge (u, v) is deleted from the graph and that v is the farther of the two nodes, that is, $d(s, v) \geq d(s, u)$.

Similar to the edge insertion case, we start by making the following important observation: an edge deletion can only result in increase of distance between any pair of nodes. An immediate corollary is that the deletion of an edge (u, v) can only affect the PSPT of those nodes s that contain both u and v in their PSPT. To see this, consider any node s and its PSPT. If $v \notin$ PSPT(s), then we know that $d(s, v) \geq d(s, \beta(s))$. Furthermore, since deleting an edge can only increase distances, in the new updated graph, the above inequality holds and hence the PSPT of s remains agnostic to deletion of edge (u, v) irrespective of whether or not u is contained in PSPT of s.

An important design decision made in ShapeShifter for edge deletions is that we let the size of $\mathcal{B}_s$ shrink over time; that is, ShapeShifter does not necessarily keep track of $\log(n)$ closest boundary nodes for any node s whose PSPT undergoes a re-shaping due to edge deletions. This shrinking of $\mathcal{B}_s$ size over time may, after a certain number of deletions, lead to $\mathcal{B}_s$ being empty; in such a case, the PSPT of s is simply recomputed from scratch. We now describe the two cases when shrinking of $\mathcal{B}_s$ occurs and describe how ShapeShifter handles this. The first case is when the path from $\beta(s)$ to s before edge deletion does not contain v (the farther of the nodes in the deleted edge); in this case,

Algorithm 14 PSPT-Delete-Update Subroutine

```
1: For each node w' ∈ N(w) ∩ T(v)
2:     If δ2(s, w') > push-dist + weight of edge(w, w')
3:         δ2(s, w') ← push-dist + weight of edge(w, w')
4:         push-dist ← δ2(s, w')
5:         PSPT-Delete-update-(w', push-dist, v)
```

all nodes in $\mathcal{B}_s$ whose distance to s after edge deletion is greater than $d(s, \beta(s))$ are deleted from $\mathcal{B}_s$, resulting in shrinking of $\mathcal{B}_s$ size. The second case is when the path from $\beta(s)$ to s before edge deletion does contain v. In this case, ShapeShifter finds the farthest node in $\mathcal{B}_s$ whose path to s does not contain v and lets this to be the new $\beta(s)$. The old $\beta(s)$ and all nodes at distance greater than the distance between s and the new $\beta(s)$ are deleted from $\mathcal{B}_s$. Essentially, the above discussion leads to the fact that ShapeShifter always considers $\beta(s)$ to be the farthest node in $\mathcal{B}_s$ whose path to s in PSPT (s) does not contain v.

So, let $\beta(s)$ to be the farthest node in $\mathcal{B}_s$ whose path to s in PSPT (s) does not contain v. We now describe the update process for the other nodes in PSPT (s). As discussed earlier, once the edge (u, v) is deleted, the only nodes whose distance from s changes are the ones in $\mathcal{T}(v)$ — the subtree rooted at v (the farther of the two nodes). To this end, we first set the distance from s to nodes in $\mathcal{T}(v)$ to be infinity (line 14). The next step is to find new paths from s to each node in $\mathcal{T}(v)$. In the first step (line $13-16$), we find for each node in $\mathcal{T}(v)$ whether there is a path via one of its neighbors outside $\mathcal{T}(v)$; since we are only interested in paths of length $d(s, \beta(s))$, we can restrict this search to neighbors that are in PSPT (s).

The next step is to find, for each node in $\mathcal{T}(v)$, candidate paths via its neighbors in $\mathcal{T}(v)$. To do this (see lines $17-20$ and subroutine PSPT-Delete-Update), ShapeShifter uses the results of the first step — it pushes the path information computed in the first step (paths via neighbors not in $\mathcal{T}(v)$) through nodes in $\mathcal{T}(v)$; that is, a node x having path via neighbors not in $\mathcal{T}(v)$ informs each of its neighbors x' in $\mathcal{T}(v)$ that they can have a path of length $\delta_1(s, x) +$ weight of edge(x, x') via x to s; this path information is recursively pushed by x' to all its neighbors in $\mathcal{T}(v)$.

At the end of the previous step, each node in $\mathcal{T}(v)$ has a list of paths to s — one via its neighbors not in $\mathcal{T}(v)$ and few paths via other nodes in $\mathcal{T}(v)$. In the third step (line $21-24$), each node settles on to the least-cost path among all

Algorithm 15 Updating the index upon **deletion** of edge (u, v)

Input: $\mathcal{IB}(u)$, $\mathcal{IB}(v)$
Let $\mathcal{U} \leftarrow \mathcal{IB}(u) \cap \mathcal{IB}(v)$

For each $s \in \mathcal{U}$
    If edge (u, v) not in PSPT(s)
        DO NOTHING
    Else
        /* Let $d(s, v) > d(s, u)$, else switch u and v */
        While path from $\beta(s)$ to s contains v
            If size of $\mathcal{B}_s$ is less than 2
                Recompute ball from scratch
            Remove $\beta(s)$ from $\mathcal{B}_s$
            Let $\beta(s)$ be the farthest node in $\mathcal{B}_s$
        For each node $w \in \mathcal{T}(v)$
            $\delta_1(s, w), \delta_2(s, w) \leftarrow \infty$
            $\mathcal{N}_w \leftarrow N(w) \cap PSPT(s) \setminus \mathcal{T}(v)$
            $\delta_1(s, w) \leftarrow \min_{x \in \mathcal{N}_w} \{d(w, x) + d(s, x)\}$
        For each node $w \in \mathcal{T}(v)$
            If $\delta_1(s, w) < \infty$
                push-dist $\leftarrow \delta_1(s, w)$
                PSPT-Delete-Update$(w,$ push-dist$, v)$
        For each node $w \in \mathcal{T}(v)$
            $\delta(s, w) \leftarrow \min\{\delta_1(s, w), \delta_2(s, w)\}$
            If $\delta(s, w) < d(s, \beta(s))$
                Insert $\langle w, \delta(s, w)\rangle$ in PSPT (s)
        PSPT-Boundary-Update(s)

these paths. It is easy to show that this is, indeed, the shortest path from each node in $\mathcal{T}(v)$ to s via nodes that lie in PSPT (s) before the edge deletion. It is also an easy exercise to show that we need not explore any other paths due to the invariant that $\beta(s)$ is the farthest node in $\mathcal{B}_s$ whose path to s before edge deletion does not contain v.

In the final step, the PSPT of s is cleaned — all nodes at distance $d(s, \beta(s))$ or greater are removed from the PSPT. This completes the description of the ShapeShifting process for edge deletions.

7.4 ShapeShifting for Queries

In this section, we describe how ShapeShifter computes distances and paths between any given pair of nodes s, t. As mentioned previously, the design of ShapeShifter is tailored towards the common case scenario — since most of the application query for nodes that are "close", the algorithm in this section ensures that if s and t are close, distances and paths are computed extremely quickly; for cases when s and t are farther away, ShapeShifter may incur larger, but acceptable, latencies. This is particularly interesting for networks that exhibit a small-world behavior (most node pairs are extremely close to each other), where ShapeShifter is able to answer queries between any pair of vertices in less than a millisecond.

We start the section with a description of the ShapeShifter algorithm for computing the distance between s and t. We then extend the implementation to retrieve the corresponding shortest paths. Finally, we extend the technique for applications where multiple paths need be retrieved between s and t.

Distance Computation. To start with, assume that both nodes are of degree greater than 1. Then, the algorithm returns the exact distance within two hash table lookups if either node is contained in the PSPT of the other node. If such is not the case, the algorithm computes, for both s and t, larger PSPT *on the fly* — these PSPT are stored temporarily during the query. Specifically, when queried for the distance between s and t, the algorithm constructs a new hash table as follows: it iterates through each node w in the PSPT of s and inserts all the nodes in the PSPT of w in the hash table; it does the same for node t.

The algorithm then iterates through each node w in the newly constructed hash table of s and checks if w is contained in the newly constructed hash table of t; if it does, then sum of distance from s to w and from t to w is a candidate shortest distance. In the end, the minimum of all the candidate distances is returned. If either of the nodes is of degree 1, we replace the node by its neighbor and add the corresponding distance in the result.

Extension for multiple paths We now extend the algorithm for computing multiple paths between a given pair of nodes. Prior to that, we note that the previously described algorithm already computes multiple paths; each PSPT intersection corresponds to a path between s and t — the problem is that

Algorithm 16 QuerySPR(s,t) — algorithm for computing the distance between nodes s and t. $H(s)$ and $H(t)$ are hash tables temporarily constructed to enable checking PSPT intersection quickly. See Table 7.1 for the notation.

Input: Nodes s, t, PSPT(s), PSPT(t)

If $\deg(s) = 1$
    return $d(s, N(s)) + \text{Query}(N(s), t)$
If $\deg(t) = 1$
    return $d(t, N(t)) + \text{Query}(s, N(t))$
If $s \in \text{PSPT}(t)$ OR $t \in \text{PSPT}(s)$
    Return $d(s,t)$

doIntersect? $\leftarrow$ false
$H(s) \leftarrow \text{PSPT}(s)$
$H(t) \leftarrow \text{PSPT}(t)$
$\delta \leftarrow \infty$
While (!doIntersect?)
    For each node $w \in H(s)$
        For each node $w' \in \text{PSPT}(w)$
            $H(s) \leftarrow \langle \text{key}, \text{value} \rangle = \langle w', d(t,w) + d(w,w') \rangle$
    For each node w in PSPT of t
        For each node w' in PSPT of w
            $H(t) \leftarrow \langle \text{key}, \text{value} \rangle = \langle w', d(t,w) + d(w,w') \rangle$
    For each $w \in H(s)$
        If $w \in H(t)$
            $\delta = \min\{\delta, d(s,w) + d(t,w)\}$
            doIntersect? $\leftarrow$ true

Delete $H(s), H(t)$
Return δ

some of these paths may be duplicated (if the PSPT intersect along multiple nodes of a path). Hence, the high-level idea is to output a path corresponding to each intersection of the two PSPTs while avoiding duplicate paths.

To achieve this, we maintain a list of "visited" nodes during the execution of the algorithm. Specifically, for any pair of nodes s and t, upon finding a node w that belongs to the PSPT of both s and t, the algorithm first checks if w is marked as visited; if yes, we ignore the node w. If not, we compute subpaths $s \rightsquigarrow w$ and $w \rightsquigarrow t$ and mark each node along these two subpaths as visited and outputs the path $s \rightsquigarrow w \rightsquigarrow t$ as earlier. It is easy to see that by maintaining such a list of visited nodes, the algorithm never outputs duplicate paths.

7.5 Evaluation

In this section, we evaluate the performance of ShapeShifter over several real-world datasets. We start by describing the datasets and experimental setup (§7.5.1). We then discuss the performance of ShapeShifter in terms of preprocessing time (time taken to construct the data structure) and memory requirements, time taken to update the data structure upon an edge update, and the time taken to answer each query.

7.5.1 Datasets and Experimental Setup

We start by describing the experimental setup used to study the performance of ShapeShifter. The datasets used in our experiments are shown in Table 7.2.

For each dataset, we did the following set of experiments. We constructed PSPT of size $\alpha n^{1/4}$ for $\alpha = 2, 3, 4, 5$. We then sampled 1000 nodes uniform randomly resulting in one million pairs of nodes per experiment. For each dataset, we repeated the experiment 10 times, resulting in $10,000$ unbiased samples for nodes and 10 million unbiased samples for node pairs.

We discuss the results for $\alpha = 3$ which is the most interesting operating point in terms of the trade-offs between memory, latency and accuracy — smaller values of α lead to significantly lower accuracy and larger values of α lead to larger query time.

Table 7.2: Social network datasets used in our experiments. The DBLP dataset is from [100]; the LiveJournal dataset is from [101] and the rest of the datasets are from [12].

Topologies	# Nodes	% Nodes with degree ≤ 1	# Undirected links
DBLP	$710,332$	13.77%	$2,509,945$
Flickr	$1,715,255$	50.95%	$15,555,041$
Orkut	$3,072,441$	2.21%	$117,185,083$
LiveJournal	$4,846,609$	21.83%	$42,851,237$

Table 7.3: Average preprocessing time and average memory requirements for ShapeShifter.

Dataset	Preprocessing time (ms/node)	Memory requirements (kB per node)
DBLP	0.4	0.48
Flickr	4.3	0.18
Orkut	5.7	1.03
LiveJournal	1.9	0.72

7.5.2 Preprocessing time

The preprocessing time for constructing the data structure is shown in Table 7.3. We note that the Orkut network requires more preprocessing time due to the high average degree of the network; the Flickr network, on the other hand, has large average degree after removing the degree-1 nodes. For the DBLP and for the LiveJournal network, the preprocessing time is extremely small ranging from 0.5 ms to less than 2 ms per node.

7.5.3 Memory Requirements

In terms of memory, ShapeShifter requires storing $6n^{1/4}$ entries corresponding to each node with degree greater than 1 — $3n^{1/4}$ for storing the PSPT and the other $3n^{1/4}$ for storing the set of nodes that contain the node in their PSPT. Table 7.3 shows the average memory requirements per node for the data structure that allows retrieving paths. Consider the LiveJournal network, which requires 1.01 GB of space on disk; our data structure, on the other hand, requires 3.33 GB of space while allowing us to quickly compute short paths and quickly update the data structure upon each edge insertion and/or deletion. This memory requirement is comparable to the schemes that compute approximate distances and paths as opposed to shortest paths in our technique, have orders of magnitude higher query time and work only for the case of static graphs.

Table 7.4: Accuracy, Query time and Update time results for ShapeShifter. For vertex pairs whose PSPT intersect along the shortest path, ShapeShifter returns the shortest path; otherwise, ShapeShifter returns a low stretch path.

Dataset	Fraction of intersecting PSPTs		Query time (in μs)	Average Update time (in ms/update)
	Total	Shortest Path		
DBLP	1	0.98	414.2	1.5
Flickr	1	0.93	521.0	1.7
Orkut	1.00	0.91	922.1	2.1
LiveJournal	1.00	0.96	997.1	2.4

7.5.4 Update time

The results for the *average* update time are shown in Table 7.4 for the case when we have equal number of edge insertions and deletions. We make two observations. First, the average time required to update the data structure is less than 2.5 ms for each network. Second, the update time scales very well with increase in the size of the network. For instance, the LiveJournal social network is roughly 6.8× larger than the DBLP network; the update time, on the other hand, is just 1.6× larger.

7.5.5 Accuracy

For the LiveJournal network and with PSPT of size $3n^{1/4}$, ShapeShifter returns the shortest path for 96.380% of the source-destination pairs, which is roughly 3.5% less when compared to ASAP from Chapter 6. For the case when PSPTs intersect but not along the shortest paths, ShapeShifter returns a path of length $d(s,t)+3W_{\max}$ for 3.162% of the source-destination pairs; in comparison, ASAP from Chapter 6 returned a path of length $d(s,t)+W_{\max}$ for 0.150% of the source-destination pairs. Finally, while ASAP from Chapter 6 could not return a path for 0.021% of the source-destination pairs, ShapeShifter always returns a path between any given pair of nodes albeit at the cost of higher query time. However, this slight reduction in accuracy comes with the advantage of extremely low memory footprint (roughly 22.5× lower than that of ASAP from Chapter 6) and efficient handling of dynamic networks.

7.5.6 Query latency

Regarding query latency, we note that ShapeShifter requires significantly higher time to compute the shortest paths — roughly 5× in comparison to that of ASAP from Chapter 6. However, for all practical purposes, a latency of less than a millisecond is quite acceptable for most applications; in addition, this latency is still 2-3 orders of magnitude lower than that required by techniques that compute paths with extremely low error and do not allow incremental updates in the data structure.

7.6 Related Work

Our goals are related to two key areas of related work:

Shortest path algorithms and heuristics. Heuristics like $A^\star$ search [91, 92] and bidirectional search [92] have been proposed to overcome the latency problems with traditional algorithms for computing shortest paths. The approaches in [91, 92], although useful in reducing the query time, still require running a (modified) shortest path algorithm for each query and do not meet the latency requirements. For instance, the experimental results in §6.5 show that bidirectional search can take hundreds of milliseconds to compute the shortest paths even on moderate size networks.

ASAP from Chapter 6, TEDI [104] and Pruned Landmark Labeling [103] further reduce the latency of shortest path heuristics at the cost of higher memory requirements. All of these techniques compute shortest paths in tens of microseconds, but are limited to static graphs. To the best of our knowledge, both TEDI and Pruned Landmark Labeling require re-computing the entire data structure from scratch upon each update. ASAP does not require the entire data structure to be recomputed from scratch (by way of keeping an inverted index like ShapeShifter); however, it does require hundreds of milliseconds to update the data structure upon each edge insertion and/or deletion.

In comparison to [53, 91, 92, 103, 104], ShapeShifter has lower memory requirements and higher query latency; however, ShapeShifter solves the more practical problem of handling thousands of edge updates per second.

Approximation algorithms. Arguing that the above heuristics [91, 92] are unlikely to meet the stringent latency requirements of social network applications, [25–27, 94, 95, 105] focus on computing *approximate* distances and paths. The body of work can be broadly characterized into two categories.

The first category uses techniques from graph embedding literature [25,26]. The main advantage of these schemes is their low memory footprint; however, these schemes often compute paths of high worst-case stretch (providing a guarantee of $\log(n)$ stretch for a network with n nodes) [25, 26], are often limited to distance computations [25], and require reconstructing the entire data structure from scratch in case of network updates [25,26]. ShapeShifter, on the other hand, provides latency similar to the above techniques while providing the benefits of almost always computing the shortest distances and paths and efficient update of the data structure upon network updates.

The second category uses techniques from distance oracle literature [27,94, 95, 105]. In comparison to these techniques, ShapeShifter differs in several aspects. First, techniques in this category that have lowest latency [95] return paths that have high absolute error (more than 3 hops on an average, even on small networks); in comparison, ShapeShifter computes shortest paths between almost all source-destination pairs. On the other hand, techniques that provide significantly better accuracy require 3-4 orders of magnitude higher query time when compared to ShapeShifter [94,105]. Third, similar to graph embedding based techniques, some of these techniques [27] are unable to compute the actual paths, while ShapeShifter can. Finally, distance oracle based techniques are known to not admit efficient algorithms for updating the data structure requiring a large number of single-source shortest path computations upon each update and each such computation takes time in the order of tens to hundreds of seconds; in contract, ShapeShifter can update the data structure in a couple of milliseconds on a single machine.

Chapter 8

Conclusions and Future Work

This dissertation developed techniques, algorithms and systems for quickly computing short paths on (static and dynamic) graphs while maintaining feasible memory requirements. The techniques and algorithms developed in the dissertation substantially break a decade-old lower bound barrier from the theory community, which is shown to hold only for the obscure case of extremely dense graphs. By exploiting *graph sparsity*, a property almost always encountered in big graph data, our techniques and algorithms are able to achieve results that are significantly better than what was previously thought possible. Building upon these theoretical advancements, the second part of the dissertation presents two systems — ASAP and ShapeShifter — for quickly computing short paths on massive static and dynamic graphs, respectively.

8.1 Contributions and Key Results

The contributions of this dissertation are three-fold.

Dense versus Sparse graphs. Our first contribution is to formally establish a separation between the sparse and the dense cases for the distance oracle problem. For the realistic case of sparse graphs, our oracles exhibit a smooth three-way trade-off between space, stretch and query time — a phenomenon that does not occur in dense graphs.

Such a separation between dense and sparse graphs is both interesting and important. First, far from being a narrow special case of the problem, sparse graphs are the most relevant case. Nearly all large real-world networks are sparse, including social networks [11], the Internet graph [46] and networks like expander graphs that are important in many settings. Our results show that the classic results in distance oracles do not apply to the realistic case of

sparse graphs.

The second reason sparse graphs are interesting is that the mathematical structure of the question changes dramatically in the case of sparse graphs. In the dense case the key is to *compress* the graph while ensuring that sufficient information remains to return low-stretch distances. In the sparse case the graph need not be compressed, but the trade-off with *query time* becomes critical. This observation leads to new insights into several longstanding open problems in theoretical computer science.

Theoretical Contributions: Space-Stretch-Time Trade-off. For the realistic case of sparse graphs, our oracles exhibit a smooth three-way trade-off between space, stretch and query time — a phenomenon that does not occur in dense graphs. We summarize a few interesting operating points on this space-stretch-time trade-off.

First, this dissertation presents linear-space, constant-stretch distance oracles for stretch 2 and larger; this is achieved at the cost of a small query time, which depends on the desired stretch and graph sparsity. For the realistic case of graphs with $m = \widetilde{O}(n)$ edges, the best known oracles prior to this dissertation required $\Theta(n^{1.5})$ and $\Theta(n^2)$ space for computing stretch 3 and stretch 2 distances, respectively; our oracles require just $\widetilde{O}(n)$ space.

Second, our oracles are the first to allow computing distances of stretch less than 2 for general weighted undirected graphs. The problem of computing distances of stretch less than 2 has deep connections with several other problems, including all-pair shortest paths (ASAP) and combinatorial Boolean Matrix Multiplication (BMM). We show that improving either the query time or the construction time of our oracles for stretch less than 2 will lead to the first $o(mn)$-time combinatorial BMM algorithm, a longstanding open problem in theoretical computer science; this, in turn, will lead to a faster ASAP algorithm, another longstanding open problem.

ASAP and ShapeShifter: Shortest Paths in Microseconds. This dissertation presented two systems — ASAP and ShapeShifter — for computing short paths on big graph data. These systems build upon strong theoretical foundations, have feasible (bounded) memory requirements, and on graphs with millions of nodes and edges, can compute almost-exact paths in microseconds.

ASAP builds upon the idea of Partial Shortest Path Trees (PSPT) — a carefully defined data structure with the property that for most pairs of nodes, the shortest path between the nodes is entirely contained within the PSPTs of the two nodes. ASAP demonstrates and exploits the observation that the structure of social networks enables the PSPT of each node to be an extremely small fraction of the entire network; hence, PSPTs can be stored efficiently and each shortest path can be computed extremely quickly. ASAP admits efficient distributed implementation and can be easily mapped on distributed programming frameworks like MapReduce.

ShapeShifter extends ASAP for the case of dynamic graphs. In particular, it shows how to maintain PSPTs when new edges are being inserted and/or deleted in the graph. ShapeShifter can update the node PSPTs, upon each edge insertion and/or deletion, within tens of microseconds while answering each user query in hundreds of microseconds.

In summary, this dissertation presented techniques, algorithms and systems that fundamentally advance our understanding of the problem of computing distances on massive graphs. By exploiting graph sparsity, a property almost always encountered in big graph data, this dissertation is able to achieve results that are significantly better than what was previously thought possible.

8.2 Future Work

Finally, we present a number of problems that this dissertation leaves open.

- Is it possible to reduce the query time of our stretch-3 distance oracle from Chapter 3? In other words, can one design an oracle of size $O(m + n^2/\alpha^2)$ that returns stretch-3 paths in $o(\alpha\mu)$ time? A more challenging problem is to design oracles that have size $O(m\alpha + n^2/\alpha^2)$ and return stretch-3 paths in $o(\alpha)$ time.

- Is it possible to reduce the query time of our stretch-2 distance oracle from Chapter 4? In other words, can one design an oracle of size $O(m + n^2/\alpha)$ that returns stretch-2 paths in $o(\alpha\mu)$ time? A more challenging problem is to design oracles that have size $O(m\alpha + n^2/\alpha)$ and return stretch-2 paths in $o(\alpha)$ time. There are two interesting sub-problems in this direction:

  - Is it possible to reduce the query time at the expense of higher preprocessing time (but fixed space)? Such a result would allow to improve the space-time trade-off for stretch-2 oracles and most likely lead to improvements in space-time trade-off for oracles for stretch less than 2.
  - Is it possible to reduce the query time without any increase in the preprocessing time and space? Such a result would be very significant — it would directly lead to the first improvement on a decade-old result on combinatorial algorithms for computing *all-pair* stretch-2 distances [107].

- Is it possible to reduce the query time of our stretch-1.666... distance oracles from Chapter 5 without increasing the preprocessing time? This would be a significant breakthrough — as discussed in the dissertation, this will lead to the first $o(mn)$-time combinatorial algorithm for Boolean Matrix Multiplication and most likely, to an asymptotically faster combinatorial algorithm for all-pairs shortest path problem.

- For sake of simplicity, we ignored the lower order terms in our results. There is, in fact, an interesting problem related to these lower order terms. Specifically, if one can reduce the query time of our algorithm of Theorem 7 (for $k = 1$, stretch-$(1 + 1/(k + 0.5))$) by $\log^c(n)$ for some large enough c, we would get a combinatorial algorithm for BMM that is asymptotically faster than the state-of-the-art [87]. Is it possible?

- Is it possible to reduce the space-time trade-off for our stretch-3/2 oracle from Chapter 5? There are absolutely no reasons to suggest that stretch-3/2 oracles require more space or query time in comparison to stretch-5/3 oracles. In this context, it would be interesting to prove or disprove a separation between oracles with stretch-k and stretch-less-than-k for $1 \leq k < 2$.

- Is it possible to design compact routing schemes for our linear-space distance oracles from Chapter 3 and from Chapter 4? We presented a distributed implementation of our stretch 2 compact routing scheme only for oracles that have an aggregate memory requirement of $O(m\alpha + n^2/\alpha)$, but not for our linear space oracles (both for stretch 2 and stretch

3 schemes). While it seems significantly more challenging, a distributed version of our linear-space oracles would have significant implications in practice — one could achieve stretch 3 with constant amount of storage at nodes in the network.

- The most intriguing problem is to compute lower bounds for oracles that take $\Omega(\log n)$ query time and return constant stretch paths.

Finally, let us mention a problem that remains at the core of the distance oracle problem. The holy grail of the distance oracle problem for sparse graphs is whether one can design an oracle of size $O(m \text{ polylog}(n))$ that yields constant stretch paths in $O(\text{polylog}(n))$ time. This would be a very significant result.

References

[1] http://tinyurl.com/26lb8p7, "Techcrunch," 2008.

[2] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at Facebook," in *ACM International Conference on Management of Data (SIGMOD)*, 2010, pp. 1013–1020.

[3] http://www.informationweek.com/news/software/bi/207800705, 2008.

[4] D. Duellmann, "Examples of future large scale scientific databases," in *Extremely Large Databases Workshop*, 2010.

[5] E. M. Marcotte and S. V. Date, "Exploiting big biology: Integrating large-scale biological data for function inference," *Briefings in Bioinformatics*, vol. 2, no. 4, pp. 363–374, 2001.

[6] J. Faulon and A. Bender, *Handbook of Chemoinformatics Algorithms*. Chapman and Hall/CRC Mathematical and Computational Biology, 2010.

[7] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *ACM International Conference on World Wide Web (WWW)*, 1998, pp. 107–117.

[8] http://en.wikipedia.org/wiki/PageRank.

[9] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener, "Graph structure in the web," *Computer Networks*, vol. 33, no. 1-6, pp. 309–320, 2000.

[10] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *ACM Workshop on Online Social Networks (WOSN)*, 2009.

[11] Y.-Y. Ahn, S. Han, H. Kwak, S. Moon, and H. Jeong, "Analysis of topological characteristics of huge online social networking services," in *ACM International Conference on World Wide Web (WWW)*, 2007, pp. 835–844.

[12] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *ACM Internet Measurement Conference (IMC)*, 2007, pp. 29–42.

[13] D. J. Watts, P. S. Dodds, and M. E. J. Newman, "Identity and search in social networks," *Science*, vol. 296, no. 5571, pp. 1302–1305, 2002.

[14] M. Cha, A. Mislove, and K. P. Gummadi, "A measurement-driven analysis of information propagation in the Flickr social network," in *ACM International Conference on World Wide Web (WWW)*, 2009, pp. 721–730.

[15] A. Mislove, A. Post, K. P. Gummadi, and P. Druschel, "Ostra: Leverging trust to thwart unwanted communication," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008.

[16] B. Viswanath, A. Post, K. P. Gummadi, and A. Mislove, "An analysis of social network-based sybil defenses," in *ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM)*, 2010, pp. 363–374.

[17] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman, "Sybilguard: defending against sybil attacks via social networks," in *ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM)*, 2006, pp. 267–278.

[18] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. d. C. Reis, and B. Ribeiro-Neto, "Efficient search ranking in social networks," in *ACM Conference on Information and Knowledge Management (CIKM)*, 2007, pp. 563–572.

[19] A. Ukkonen, C. Castillo, D. Donato, and A. Gionis, "Searching the wikipedia with contextual information," in *ACM Conference on Information and Knowledge Management (CIKM)*, 2008, pp. 1351–1352.

[20] P. Singla and M. Richardson, "Yes, there is a correlation: from social networks to personal behavior on the web," in *ACM International Conference on World Wide Web (WWW)*, 2008, pp. 655–664.

[21] S. Amer-Yahia, M. Benedikt, L. V. Lakshmanan, and J. Stoyanovic, "Efficient network-aware search in collaborative tagging sites," in *International Conference on Very Large Databases (VLDB)*, 2008, pp. 710–721.

[22] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós, "Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments," *Internet Mathematics*, vol. 2, no. 3, pp. 333–358, 2005.

[23] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt, "Hierarchical graph embedding for efficient query processing in very large traffic networks," in *International Conference on Scientific and Statistical Database Management (SSDBM)*, 2008, pp. 150–167.

[24] I. Pramudiono, T. Shintani, K. Takahashi, and M. Kitsuregawa, "User behavior analysis of location aware search engine," in *IEEE International Conference on Mobile Data Management (MDM)*, 2002, pp. 139–145.

[25] X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Y. Zhao, "Orion: Shortest path estimation for large social graphs," in *ACM Workshop on Online Social Networks (WOSN)*, 2010.

[26] X. Zhao, A. Sala, H. Zheng, and B. Y. Zhao, "Efficient shortest paths on massive social graphs," in *IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2011, pp. 77–86.

[27] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *ACM Conference on Information and Knowledge Management (CIKM)*, 2009, pp. 867–876.

[28] W. Chen, Y. Wang, and S. Yang, "Efficient influence maximization in social networks," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2009, pp. 199–208.

[29] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2003, pp. 137–146.

[30] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3–5, pp. 75–174, 2010.

[31] C. R. Palmer, P. B. Gibbons, and C. Faloutsos, "ANF: A fast and scalable tool for data mining in massive graphs," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2002, pp. 81–90.

[32] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna, "Four degrees of separation," http://arxiv.org/abs/1111.4570, 2011.

[33] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the facebook social graph," http://arxiv.org/abs/1111.4503, 2011.

[34] E. Karpilovsky and J. Rexford, "Using forgetful routing to control BGP table size," in *ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, 2006, pp. 2:1–2:12.

[35] D. Chang, R. Govindan, and J. Heidemann, "An empirical study of router response to large BGP routing table load," in *ACM SIGCOMM Workshop on Internet Measurement (IMW)*, 2002, pp. 203–208.

[36] M. Yu, A. Fabrikant, and J. Rexford, "Buffalo: bloom filter forwarding architecture for large organizations," in *ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, 2009, pp. 313–324.

[37] N. Feamster, H. Balakrishnan, and J. Rexford, "Some foundational problems in interdomain routing," in *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2004.

[38] J. Brutlag, "Speed matters for Google web search," 2009. [Online]. Available: http://code.google.com/speed/files/delayexp.pdf

[39] www.facebook.com/press/info.php?statistics.

[40] LinkedIn Media Information: http://press.linkedin.com/about, December 2012.

[41] M. Thorup and U. Zwick, "Approximate distance oracles," *Journal of the ACM*, vol. 52, no. 1, pp. 1–24, 2005.

[42] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg, "Near-linear time construction of sparse neighborhood covers," *SIAM Journal on Computing*, vol. 28, no. 1, pp. 263–277, 1998.

[43] E. Cohen, "Fast algorithms for constructing t-spanners and paths with stretch t," *SIAM Journal on Computing*, vol. 28, no. 1, pp. 210–236, 1998.

[44] J. Matoušek, "On the distortion required for embedding finite metric spaces into normed spaces," *Israel Journal of Mathematics*, vol. 93, no. 1, pp. 333–344, 1996.

[45] D. Schultes, *Route Planning in Road Networks*. University of Karlsruhe: PhD Thesis, February, 2008.

[46] The Cooperative Association for Internet Data Analysis, 2009. [Online]. Available: http://www.caida.org/home/

[47] http://www.facebook.com/press/info.php?statistics.

[48] R. Agarwal, P. B. Godfrey, and S. Har-Peled, "Approximate distance queries and compact routing in sparse graphs," in *IEEE International Conference on Computer Communications (INFOCOM)*, 2011, pp. 1754–1762.

[49] R. Agarwal, P. B. Godfrey, and S. Har-Peled, "Faster approximate distance queries and compact routing in sparse graphs," *ArXiv*, 2012.

[50] R. Agarwal and P. B. Godfrey, "Brief announcement: A simple stretch 2 distance oracle," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 2013.

[51] R. Agarwal and P. B. Godfrey, "Distance oracles for stretch less than 2," in *ACM-SIAM Symposium on Discrete Algorithm (SODA)*, 2013.

[52] R. Agarwal, "The space-stretch-time trade-off in distance oracles," Technical Report, 2013.

[53] R. Agarwal, M. Caesar, P. B. Godfrey, and B. Y. Zhao, "Shortest paths in less than a millisecond," in *ACM SIGCOMM Workshop on Online Social Networks (WOSN)*, 2012.

[54] R. Agarwal, M. Caesar, P. B. Godfrey, and B. Y. Zhao, "Shortest paths in microseconds," Technical Report, 2013.

[55] R. Agarwal, C. Zhu, M. Caesar, and P. B. Godfrey, "Shapeshifter: Fast and accurate shortest path computation on dynamic graphs," Technical Report, 2013.

[56] Wikipedia, http://en.wikipedia.org/wiki/Metric_space.

[57] N. Alon and J. H. Spencer, *The probabilistic method*. Wiley-Interscience, 1992, vol. 57.

[58] M. Thorup and U. Zwick, "Compact routing schemes," in *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001, pp. 1–10.

[59] L. Roditty, M. Thorup, and U. Zwick, "Deterministic constructions of approximate distance oracles and spanners," in *Automata, languages and programming*. Springer, 2005, pp. 261–272.

[60] S. Baswana and S. Sen, "Approximate distance oracles for unweighted graphs in expected $O(n^2)$ time," *ACM Transactions on Algorithms*, vol. 2, no. 4, pp. 557–577, 2006.

[61] S. Baswana and T. Kavitha, "Faster algorithms for approximate distance oracles and all-pair small stretch paths," in *Proc. IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 2006.

[62] S. Baswana, A. Gaur, S. Sen, and J. Upadhyay, "Distance oracles for unweighted graphs: Breaking the quadratic barrier with constant additive error," in *International Colloquium on Automata, Languages and Programming (ICALP)*, July 2008.

[63] C. Wulff-Nilsen, "Approximate distance oracles with improved preprocessing time," in *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2012, pp. 202–208.

[64] M. Mendel and A. Naor, "Ramsey partitions and proximity data structures," *Journal of European Mathematical Society*, vol. 2, no. 9, pp. 253–275, 2007.

[65] C. Wulff-Nilsen, "Approximate distance oracles with improved preprocessing time," in *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2013.

[66] A. Naor and T. Tao, "Scale-oblivious metric fragmentation and the nonlinear dvoretzky theorem," *Israel Journal of Mathematics*, vol. 192, no. 1, pp. 489–504, 2012.

[67] M. Mendel and C. Schwob, "C-K-R partitions of sparse graphs," ArXiv, abs/0809.1902, 2008.

[68] S. Chechik, "Approximate distance oracle with constant query time," *ArXiv*, vol. abs/1305.3314, 2013.

[69] M. Thorup, "Compact oracles for reachability and approximate distances in planar digraphs," *Journal of the ACM*, vol. 51, no. 6, pp. 993–1024, 2004.

[70] K. Kawarabayashi, C. Sommer, and M. Thorup, "More compact oracles for approximate distances in undirected planar graphs," in *24th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2013, pp. 550–563.

[71] K. Kawarabayashi, P. N. Klein, and C. Sommer, "Linear-space approximate distance oracles for planar, bounded-genus and minor-free graphs," in *38th International Colloquium on Automata, Languages and Programming (ICALP)*, 2011, pp. 135–146.

[72] W. Chen, C. Sommer, S.-H. Teng, and Y. Wang, "A compact routing scheme and approximate distance oracle for power-law graphs," *ACM Transactions on Algorithms*, vol. 9, no. 1, pp. 4:1–26, 2012, announced at DISC 2009.

[73] M. Enachescu, M. Wang, and A. Goel, "Reducing maximum stretch in compact routing," in *IEEE INFOCOM*, 2008.

[74] C. Sommer, E. Verbin, and W. Yu, "Distance oracles for sparse graphs," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2009, pp. 703–712.

[75] M. Pătraşcu, L. Roditty, and M. Thorup, "A new infinity of distance oracles for sparse graphs," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2012.

[76] M. Pătraşcu and L. Roditty, "Distance oracles beyond the Thorup-Zwick bound," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2010, pp. 815–823.

[77] C. Gavoille and S. Perennes, "Memory requirement for routing in distributed networks," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 1996, pp. 125–133.

[78] P. Fraigniaud and C. Gavoille, "Local memory requirement of universal routing schemes," in *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Padua, Italy, June 1996, pp. 183–188.

[79] P. Fraigniaud and C. Gavoille, "Memory requirement for universal routing schemes," in *ACM Symposium on Principles of Distributed Computing (PODC)*, 1995, pp. 223–230.

[80] I. Abraham and C. Gavoille, "On approximate distance labels and routing schemes with affine stretch," in *International Symposium on Distributed Computing (DISC)*, 2011, pp. 404–415.

[81] R. Agarwal, P. B. Godfrey, and S. Har-Peled, "Approximate distance queries and compact routing in sparse graphs," University of Illinois at Urbana-Champaign, Tech. Rep., April 2010.

[82] Y. Mao, F. Wang, L. Qiu, S. Lam, and J. Smith, "S4: Small state and small stretch routing protocol for large wireless sensor networks," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007.

[83] A. Singla, P. B. Godfrey, K. Fall, G. Iannaccone, and S. Ratnasamy, "Scalable routing on flat names," in *ACM Conference on Emerging Networking Experiments and Technology (CoNEXT)*, 2010.

[84] B. A. Ford, *UIA: A Global Connectivity Architecture for Mobile Personal Devices*. Massachusetts Institute of Technology: PhD Thesis, 2008.

[85] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of internet flow rates," in *ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communications (SIGCOMM)*, August 2002, pp. 309–322.

[86] D. Dor, S. Halperin, and U. Zwick, "All pairs almost shortest paths," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1996.

[87] G. E. Blelloch, V. Vassilevska, and R. Williams, "A new combinatorial approach for sparse graph problems," in *International Colloquium on Automata, Languages and Programming (ICALP)*, 2008, pp. 108–120.

[88] N. Bansal and R. Williams, "Regularity lemmas and combinatorial algorithms," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2009, pp. 745–754.

[89] T. A. J. Nicholson, "Finding the shortest route between two points in a network," *The Computer Journal*, vol. 9, no. 3, pp. 275–280, 1966.

[90] S. E. Dreyfus, "An appraisal of some shortest-path algorithms," *Operations Research*, vol. 17, no. 3, pp. 395–412, 1969.

[91] A. Goldberg and C. Harrelson, "Computing the shortest path: A* meets graph theory," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2005, pp. 156–165.

[92] A. Goldberg, H. Kaplan, and R. Werneck, "Reach for A*: Efficient point-to-point shortest path algorithms," in *ACM-SIAM Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2006.

[93] E. Porat and L. Roditty, "Preprocess, set, query!" in *European Conference on Algorithms (ESA)*, 2011.

[94] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum, "Fast and accurate estimation of shortest paths in large graphs," in *ACM Conference on Information and Knowledge Management (CIKM)*, 2010, pp. 499–508.

[95] A. D. Sarma, S. Gollapudi, M. Najork, and R. Panigrahy, "A sketch-based distance oracle for web-scale graphs," in *ACM International Conference on Web Search and Web Data Mining (WSDM)*, 2010, pp. 401–410.

[96] S. Arikati, D. Chen, L. Chew, G. Das, M. Smid, and C. Zaroliagis, "Planar spanners and approximate shortest path queries among obstacles in the plane," *European Symposium on Algorithms (ESA)*, pp. 514–528, 1996.

[97] Lexicographical order, http://tinyurl.com/3bdqmx.

[98] J. Dean and J. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004, pp. 137–150.

[99] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *ACM International Conference on Management of Data (SIGMOD)*, 2010, pp. 135–146.

[100] DBLP: http://dblp.uni-trier.de/xml/, 2012.

[101] Stanford Large Network Dataset Collection. [Online]. Available: http://snap.stanford.edu/data/index.html

[102] Lemon Graph Library: https://lemon.cs.elte.hu/trac/lemon.

[103] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *ACM International Conference on Management of Data (SIGMOD)*, 2013.

[104] F. Wei, "TEDI: efficient shortest path query answering on graphs," in *ACM International Conference on Management of Data (SIGMOD)*, 2010, pp. 99–110.

[105] K. Tretyakov, A. Armas-Cervantes, L. Garcia-Banuelos, J. Vilo, and M. Dumas, "Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs," in *ACM Conference on Information and Knowledge Management (CIKM)*, 2011, pp. 1785–1794.

[106] L. Roditty and U. Zwick, "On dynamic shortest paths problems," in *Algorithms–ESA 2004*. Springer, 2004, pp. 580–591.

[107] E. Cohen and U. Zwick, "All-pairs small-stretch paths," *Journal of Algorithms*, vol. 38, no. 2, pp. 335–353, 2001.