Practicality of Sub-Linear Algorithms for Distributed Shortest Paths

Tung Vu

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

June 2025

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Tung Vu**

Entitled: **Practicality of Sub-Linear Algorithms for Distributed Shortest Paths**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair & Examiner
Dr. Sandra Céspedes

_____ Examiner
Dr. Denis Pankratov

_____ Supervisor
Dr. Brigitte Jaumard

_____ Supervisor
Dr. Lata Narayanan

Approved by _____
Joey Paquet, Chair
Department of Computer Science and Software Engineering

_____ 2025

_____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Practicality of Sub-Linear Algorithms for
Distributed Shortest Paths

Tung Vu

The computation of Single-Source Shortest Paths (SSSP) is a fundamental problem in distributed networks. This thesis explores efficient SSSP solutions within the synchronous *CONGEST* model, where communication occurs in rounds in which each node can send $O(\log n)$ information to each of its neighbors, where n is the size of the network. We focus on Elkin's algorithm, notable for being the first algorithm to achieve sub-linear round complexity for the problem. This work makes several contributions: we clarify many details in the implementation of Elkin's algorithm, enhance its performance using pipelining techniques, and introduce two novel variants, Elkin-R (probabilistic virtual node selection) and Elkin-D (distance-based selection). Through extensive simulations on various network topologies, we compare these algorithms against Bellman-Ford based on the number of rounds, message count, and clock time. Our findings reveal that specific configurations of the Elkin variants consistently reduce message overhead compared to Bellman-Ford. Importantly, for larger or denser graphs, these variants can also surpass Bellman-Ford in round complexity and overall computation time, demonstrating their potential for practical distributed SSSP computation.

Acknowledgments

This thesis would not have been possible without the guidance and support of many individuals. First and foremost, I wish to express my deepest appreciation to my supervisors, Dr. Brigitte Jaumard and Dr. Lata Narayanan. Their consistent encouragement, insightful critiques, and expert direction were essential throughout my research on the Practicality of Sub-Linear Algorithms for Distributed Shortest Paths. I am truly grateful for their mentorship and patience. I would also like to extend my thanks to Dr. Sandra Céspedes and Dr. Denis Pankratov for serving on my committee and providing valuable feedback that strengthened this thesis. This research was generously supported by Ciena and Mitacs, I gratefully acknowledge their contribution. I would like to thank Emil Janulewicz, from Ciena, for your continuous support throughout my research. I also want to thank Duong Quang Huy for your support at the beginning of my research. I am thankful for the support and intellectual stimulation provided by my peers in the Large Scale Optimization System lab. Lastly, I wish to thank my family and friends for their endless support and belief in me. To Nguyen Thi Anh, my grandmother, Pham Thi Thuy, my mom, and Vu Huy Hai, my dad, your encouragement during long hours and moments of difficulty meant the world to me. Thank you for everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Shortest path algorithms are demonstrably important and relevant because of their use in real-world applications in which there is a necessity to compute efficient or optimal routes based on criteria like distance or cost, or latency. In telecommunication networks, shortest path algorithms play a crucial role in efficient data packet transmission within different computer networks [28, 30]. The length of the path usually relates to latency in this context. The optimization of route planning within the transportation sector is significantly dependent on shortest path algorithms. Global Positioning Systems (GPS) in vehicles utilize Dijkstra's algorithm to find the shortest routes between locations, considering distance as a factor [38]. Beyond these primary areas, shortest path algorithms find applications in many other domains. In robotics, autonomous robots utilize shortest path algorithms such as Dijkstra's algorithm and A^* algorithm for path planning, enabling them to navigate intricate environments and maneuver around obstacles, as discussed by Karur et al. [25]. In VLSI design, shortest path algorithms are used to find the shortest wiring connection between two metal components that must be connected electrically.

Given the critical role of shortest path algorithms in many domains, the necessity for distributed approaches becomes clear when considering the challenges introduced by large and complex networks. Centralized algorithms, which rely on processing the entire network on a single machine,

often struggle with the computational requirements for large networks. Distributed algorithms offer a solution by dividing the network and the computational workload across multiple interconnected processors. This distribution provides scalability, allowing for the analysis of networks far exceeding the capacity of single-machine systems. Distributed algorithms can process networks with thousands of nodes and edges, making analysis feasible in several domains, including social network analysis, as demonstrated by Khopkar *et al.* [26], who developed a Map-Reduce implementation of an incremental All-Pairs Shortest Path (APSP) algorithm specifically for this context. For 5G networks, where distributed algorithms are advantageous, Botez *et al.* [9] tackles network slicing in the backhaul by presenting a novel approach, incorporating a modified A^* algorithm for shortest path computation, which proves more time efficient than Dijkstra's. In Software-Defined Networking (SDN), Buzachis *et al.* [10] contributed a modified distributed Dijkstra's routing algorithm capable of optimizing complex metrics and leveraging MapReduce for rapid SDN router configuration. Their system routes packets based on various factors, including hops, latency, and energy policies.

Distributed shortest path algorithms also play a vital role in fault detection and recovery. Changes in communication path cost, calculated by distributed algorithms mentioned in Hutson *et al.*[24], occur when network components like links or nodes fail; the optimal paths between other nodes often change. Network management systems leverage this by monitoring shortest path information; a sudden disappearance of a path between previously connected points strongly suggests a network fault. This detected anomaly can then automatically trigger alarms to quickly restore connectivity.

Overall, distributed algorithms offer advantages of scalability, fault tolerance, efficiency, and adaptability to dynamic changes in comparison to centralized algorithms in terms of dealing with the challenges introduced by modern network environments.

1.2 Problem definition

This thesis investigates the *distributed single-source shortest path (SSSP)* problem. The fundamental objective is to determine the path with the minimum cumulative weight from a designated source node to all other reachable nodes in the system. The network is modeled as an undirected

graph having several nodes that represent distributed entities and edges that represent direct communication links between them. Crucially, each edge has a real-valued weight, signifying the *cost or weight* (such as distance or latency) associated with traversing that specific link. In reality, an undirected edge between u and v corresponds to two bidirectional links between u and v with the same weight. Given a weighted graph $G = (V, E, w)$ where V is the set of vertices/nodes in the graph, $E \subseteq V \times V$ is the set of edges, $w : E \to \mathcal{R}$ is the weight function on the edges, and a source node $s \in V$, an algorithm for the SSSP problem computes for each $v \in V$, the cost of the shortest path from v to s, and also the successor of v on such a shortest path.

We use the *CONGEST* model of distributed computing, which is a synchronous model of computing in a network. All processing occurs in discrete, synchronized rounds. Within each round, a node can perform local computations based on its current knowledge and send $O(\log n)$ bits of information to each of its neighbors. More precisely, it can send $O(1)$ node ids or edge weights to each of its neighbors. This reflects possible limitations on edge bandwidth. A key constraint of this distributed setting is that initially, each node knows only its immediate neighbors and the weight of the edges directly incident on it; it lacks global knowledge of the network topology. It is important to note that this framework contrasts with asynchronous models, where delays complicate the algorithm, making the synchronous approach simpler for algorithm design and verification. While this model may not reflect real-world systems with variable latency, it serves as a valuable theoretical framework for understanding distributed computation principles and analyzing algorithms like shortest path finding.

Within this context, this thesis focuses on the efficient computation of exact shortest paths. The standard Distributed Bellman-Ford algorithm's time complexity scales linearly with the network size, that is, it operates in $O(n)$ rounds. In contrast, Elkin's algorithm [14] was the first distributed algorithm to obtain a sub-linear time complexity for the SSSP problem. This thesis centers its investigation on Elkin's algorithm and its applicability in practice to real communication networks. In particular, though Elkin's algorithm has better asymptotic performance than Bellman-Ford's algorithm, is it practical to implement, and what kind of performance gains can we expect to have in real communication networks? Efficiency in distributed algorithms is primarily evaluated by the number of rounds required and the total number of messages exchanged throughout the algorithm's

execution. We also measure the computational time (clock time) of our implementations.

1.3 Our contributions

We make the following key contributions to the field of distributed shortest path algorithms:

(1) **Detailed description and implementation of Elkin's algorithm:** We clarify several details in the implementation of Elkin's algorithm and correct a minor error in the algorithm. To simulate a distributed algorithm, we used a multi-threaded implementation, with each thread representing a node in the network.

(2) **Improved implementation of the Elkin Algorithm:** By using a *pipelined upcast-broadcast*, we obtain a more efficient implementation of Elkin's algorithm in terms of the number of rounds.

(3) **Introduction of Elkin-R and Elkin-D variants:** We introduce two variants of the Elkin algorithm, that we call *Elkin-R* and *Elkin-D*. They use different ways of choosing so-called *virtual nodes* in Elkin's algorithm.

(4) **Comprehensive performance comparison:** We conduct an extensive performance evaluation comparing our *improved Elkin's algorithm* against the well-established Bellman-Ford algorithm in the synchronous distributed setting, as well as our two new variants. We run experiments on graphs modeling telecommunication networks, random Erdos-Renyi graphs, and grids. This comparison analyzes key metrics, including round counts, message complexity, clock time, and the scalability of all approaches with increasing network size. Our results show that certain Elkin variants, depending on their parameter settings, outperform Bellman-Ford in message complexity across all tested graph types. Furthermore, on larger or denser graphs, some variants also surpass Bellman-Ford in terms of number of rounds and computational times.

1.4 Organization of thesis

The subsequent chapters of this thesis are organized as follows. Chapter 2 offers a literature review covering relevant background work. Chapter 3 describes Elkin's algorithm, our optimized implementation of Elkin's algorithm, and the proposed Elkin-R and Elkin-D variants. Following this, Chapter 4 presents the experimental methodology and discusses the performance results obtained from evaluations on various graphs. Finally, Chapter 5 summarizes the core contributions and findings of this research and provides future work.

Chapter 2

Literature Review

This chapter presents a comprehensive literature review of distributed algorithms for shortest-path problems. Section 2.1 introduces classical non-distributed shortest paths algorithms, providing a foundational context. In Section 2.2, we explore various distributed shortest paths algorithms that extend these classical approaches to distributed computing environments. Section 2.3 delves into the quest for sub-linear algorithms, highlighting Elkin's groundbreaking contribution of a sub-linear algorithmic approach. Finally, Section 2.3.2 explores subsequent refinements to Elkin's algorithm, presenting subsequent research that proposed significant improvements in computational complexity.

2.1 Centralized shortest paths algorithms

In this section, we recall a high-level description of the classical non-distributed shortest-paths algorithms, see, e.g., [4] for a detailed description of them and their complexities.

2.1.1 Dijkstra's algorithm

Dijkstra's algorithm [12] is a fundamental graph algorithm used to efficiently compute the shortest paths from a single source node to all other nodes in a weighted graph with non-negative edge weights. The algorithm iteratively explores the graph, maintaining tentative distances to each node. It starts by assigning a distance of zero to the source and infinity to all others. In each step, it selects

the unvisited node with the smallest current distance, marks it as visited, and updates the distances of its neighbors if a shorter path is found through the current node (a process called relaxation). This continues until all reachable nodes are visited, at which point the shortest paths from the source are determined. Depending on the data structure used for implementation, Dijkstra's algorithm shows varying time complexities. Table 2.1 provides a comparative overview of the worst-case time and space complexities for different implementations.

Table 2.1: Comparison of Dijkstra's algorithm implementations

Data Structure	**Time Complexity**	**Space Complexity**
Array/List	$O(V^2)$	$O(V)$
Binary Heap	$O((V+E)\log V)$	$O(V)$
Fibonacci Heap	$O(E+V\log V)$	$O(V)$

2.1.2 Bellman-Ford's algorithm

The Bellman-Ford algorithm [7] is another algorithm used to find the shortest paths from a single source vertex to all other vertices in a weighted graph. A key advantage of the Bellman-Ford algorithm is its ability to handle graphs with negative edge weights, a limitation of Dijkstra's algorithm.

The Bellman-Ford algorithm works by iteratively relaxing all the edges in the graph. It starts by initializing the distance to the source vertex as 0 and all other vertices to infinity (∞). Then, in each of $|V|-1$ iterations (where $|V|$ is the number of vertices), it examines every edge (u, v) in the graph and checks if the shortest path to v can be improved by going through u. This process of updating the distance to v if a shorter path is found is called relaxation, as in Dijkstra's algorithm. After $|V|-1$ iterations, the algorithm guarantees that the shortest paths to all reachable vertices have been found, assuming there are no negative-weight cycles.

Furthermore, the Bellman-Ford algorithm can detect negative-weight cycles in a graph. After performing $|V|-1$ relaxations, if another relaxation step still results in a shorter path to any vertex, it indicates the presence of a negative-weight cycle reachable from the source. In such cases, a shortest path might not exist as traversing the negative cycle repeatedly can infinitely decrease the path cost.

The time complexity of the Bellman-Ford algorithm is $O(|V| \cdot |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.

2.1.3 Floyd-Warshall's algorithm

The Floyd-Warshall algorithm [16] is a dynamic programming algorithm that finds the shortest paths between *all pairs of vertices* in a weighted graph. This algorithm can handle graphs with both positive and negative edge weights. It iteratively considers each vertex as a potential intermediate node on the shortest path between any two other vertices. For every vertex w, the algorithm checks if using w as an intermediate point improves the current shortest path between each pair of vertices u and v. If the path from u to w and then w to v is shorter than the known path from u to v, the algorithm updates the shortest distance between u and v. This process repeats for all possible intermediate vertices, ensuring that the algorithm ultimately finds the shortest paths between all pairs.

The Floyd-Warshall algorithm has a time complexity of $O(|V|^3)$, where $|V|$ is the number of vertices.

2.2 Distributed shortest paths algorithms

In this section, we discuss distributed algorithms for computing shortest paths, which are fundamental to network routing. Common routing protocols provide practical examples of such algorithms. For instance, RIP [28], an early distance-vector protocol, directly employs a distributed Bellman-Ford's approach where routers iteratively exchange routing tables with neighbors to determine shortest paths based on hop counts. RIP [28] stands as a classic case study of the Distributed Bellman-Ford algorithm in action, particularly in smaller, simpler network environments. Its distributed characteristic of each router is that it independently maintains and updates its routing table based on neighbor information, which is central to its operation. OSPF [30], on the other hand, exemplifies a distributed link-state routing protocol inspired by Dijkstra's algorithm, widely deployed in larger enterprise networks. Its distributed nature is shown in the way each router contributes to

building a shared view of the network topology through LSA flooding and then independently computes its shortest path tree. Meng [29] and Zeng *et al.* [42] are surveys that talk about practically implementing distributed shortest path algorithms.

Distributed shortest path algorithms can be categorized as synchronous or asynchronous based on the timing and coordination of the processes. Asynchronous algorithms do not assume the existence of a global clock and are designed to function correctly even with variable and unpredictable communication delays. In contrast, synchronous algorithms operate in discrete rounds, where each node performs a step of computation and communication in each round. While asynchronous algorithms are crucial for handling the complexities of real-world distributed systems, this thesis specifically investigates synchronous algorithms due to their simplicity, which makes them easier to analyze.

We go on to explore specific models and algorithms in more detail. First, Section 2.2.1 will lay the groundwork by introducing the *CONGEST* model, crucial for understanding distributed algorithm performance. Subsequently, Section 2.2.2 will examine the distributed Bellman-Ford algorithm. Finally, we describe the quest for sub-linear algorithms in Section 2.3.

2.2.1 CONGEST model

The *CONGEST* model is a commonly used model for synchronous distributed computing over a network, particularly in scenarios where communication bandwidth is a critical constraint.

In this model, a network is represented in the standard way as a weighted, undirected graph, $G = (V, E, w)$, where V represents the set of nodes (processors) in the network, E represents the set of bidirectional communication links between nodes, and w represents the weight function, assigning a weight to each edge. We assume that edge weights are polynomially bounded in n, the number of nodes in the network. Each node $u \in V$ has a unique identifier (ID) from the range $(\{1, 2, \ldots, n\})$. Initially, each node knows its incident edges and their associated weights.

Computation and communication proceed in synchronized discrete *rounds*, governed by a global clock. In each round, every node performs the following three steps:

- **(A) Receive Messages:** Receive messages sent by neighbors in the previous round.

- **(B) Local Computation:** Perform internal computations based on received messages and local knowledge.
- **(C) Send Messages:** Send (potentially different) messages of size $O(\log n)$ bits to each of its neighbors.

Note that in each round, a node is able to send $O(1)$ node ids or edge weights to each of its neighbors. It is assumed that a message sent in round i by node u to its neighbor v is received by v in round $i + 1$.

The *LOCAL* model of distributed computing is identical to the *CONGEST* model except that the size of messages is not limited to $O(\log n)$ bits.

Observe that the time taken in a round is not restricted. It is a standard assumption in distributed computing that the cost of local computation is much smaller than the cost of communication. Therefore, the standard measures of complexity of an algorithm in the *CONGEST* model are the *number of rounds* and the *number of messages*. We will sometimes refer to these as the *round count* (which serves as an approximation for *round complexity*) and the *message complexity* of an algorithm.

2.2.2 The distributed Bellman-Ford algorithm

2.2.2.1 Description

The Bellman-Ford algorithm stands as a fundamental algorithm for computing shortest paths from a single source vertex to all other vertices in a weighted directed graph. In a distributed setting, the Bellman-Ford algorithm , each node v maintains two key pieces of information: $\delta(v)$ (the shortest known distance from the source s) and parent (the neighboring node that comes before v in this shortest path). At the start, the initial node s has a $\delta(s)$ of 0, while all other nodes v have set $\delta(v)$ to ∞, with parent values undefined. During each round of the algorithm, every node sends its current $\delta(v)$ value to all its neighbors. Then, each node v performs a *relaxation* step where it examines all incoming neighbors u and calculates if there is a shorter path through any of them (by adding the edge weight from u to v to $\delta(u)$). If a shorter path is found, the node updates its $\delta(v)$.

After n rounds (where n is the number of nodes), the algorithm completes, with $\delta(v)$ where $v \in V$ values showing the shortest distances from v to s.

2.2.2.2 Pseudo-code

Algorithm 1: The distributed Bellman-Ford algorithm for each node v

Data: G, source s, node v
Result: $\ell^{\text{SP}}(s, v) = \delta_v$ is computed for $\forall v \in V$.

```
if Initiate_v = TRUE then
    v sends all estimated distance messages to all of its neighbors
    Initiate_v ← FALSE
else
    for each MSG_u received by v from one (say u) of its neighbors do
        δ(u) ← MSG_u.getDistance()
        if δ(u) + ω_G(v, u) < δ(v) then
            δ(v) ← δ(u) + ω_G(v, u)
    v sends all new estimated distance δ(v) messages to all of its neighbors
```

2.3 Quest for sub-linear algorithms

While the Single-Source Shortest Paths (SSSP) problem is fundamental, achieving its solution efficiently in distributed message-passing systems presents significant challenges. The classical distributed Bellman-Ford algorithm provides a baseline solution, but it takes n rounds, where n is the number of network vertices. This bound is the best possible in networks of $\Omega(n)$ diameter, but the question of whether it is possible to do better in graphs of $o(n)$ diameter has led to a substantial body of research. Notably, Peleg and Rubinovich [34] established that any distributed SSSP algorithm must take at least $\tilde{\Omega}(D + \sqrt{n})$ rounds, where D is the graph's hop-diameter. This raised a major open question: could SSSP be solved in sub-linear ($o(n)$) time? Elkin [13] showed that computing exact SSSP requires $\Omega(\sqrt{n})$ rounds, even in graphs of diameter $O(\log n)$.

2.3.1 Approximate SSSP

Much of the advancement occurred in the related area of *approximate* shortest paths, where near-optimal, sub-linear time algorithms were developed. The concept of hopsets, a crucial tool in approximate shortest path computations, saw a major advancement with the work of Elkin and Neiman [15]. Briefly, a (β, ε)-hopset is a set of (weighted) edges whose addition to a weighted

graph guarantees that every pair of vertices has a path between them with at most β edges and has weight at most $(1+\varepsilon)$ times the shortest path weight between them in the original graph. Their research provided the first construction of *sparse* hopsets with a *constant* number of hops, significantly improving over the previously polylogarithmic bounds on the hop bound. This breakthrough in hopset construction paved the way for potentially faster approximate shortest path algorithms in different configurations.

The first sub-linear algorithm for approximate shortest path computation was given by Lenzen and Patt-Shamir [27]. In their work on fast routing table construction, they presented a randomized algorithm that computes $O(k \log k)$-approximate distances within time $\tilde{O}(D+n^{1/2+1/k})$. We define $\tilde{O}(g(n))$ as a variant of Big O notation that allows us to ignore poly-logarithmic multiplicative factors. Specifically, $f(n) = \tilde{O}(g(n))$ if $f(n) = O(g(n) \cdot \text{polylog}(n))$, where $\text{polylog}(n)$ represents some polynomial in $\log n$ (or $\log g(n)$).

Nanongkai's work [31] also contributed significantly to the field of approximate shortest paths by achieving a $(1+o(1))$-approximation for SSSP in $\tilde{O}(n^{1/2}D^{1/4}+D)$ time. His approach introduced novel techniques for bounded-hop SSSP and shortest-path diameter reduction, further pushing the boundaries of efficiency for approximate solutions.

Further progress in approximate SSSP was made by Henzinger *et al.* [22], who developed a deterministic $(1+o(1))$-approximation algorithm with a running time of $O(n^{1/2+o(1)}+D^{1+o(1)})$. Their algorithm achieved almost-tight bounds by matching the known lower bound up to subpolynomial factors. They also introduced novel deterministic techniques for path hitting and hopset construction.

The work of Becker *et al.* [6] presented a method for achieving near-optimal approximate solutions by reducing the SSSP problem to the shortest transshipment problem. Their method yielded significant improvements in various computational models, including the Broadcast *CONGEST* model with a $(1+\varepsilon)$-approximate SSSP in $\tilde{O}((\sqrt{n}+D)\varepsilon^{-O(1)})$ rounds. Their approach notably bypassed the traditional use of hopsets, relying instead on sparse spanners.

Rozhoň *et al.* [35] addresses $(1+\varepsilon)$-approximate shortest-path type problems on undirected weighted graphs; the presented algorithms achieve performance guarantees featuring near-linear work complexity. In more detail, the execution time is given by $\tilde{O}(\sqrt{n}+D)\cdot\varepsilon^{-2}$ generally (with

n nodes and hop diameter D). However, under the specific condition that the graph G is free from any $\tilde{O}(1)$-dense minors, a faster time bound of $\tilde{O}(D) \cdot \varepsilon^{-2}$ applies, highlighting the algorithm's adaptation to certain graph structures.

2.3.2 Exact SSSP

Elkin finally solved the long-standing problem of *exact* SSSP [14]. His work presented the first distributed algorithm capable of computing exact SSSP in provably sub-linear time across a broad spectrum of network parameters. The algorithm achieves a runtime of $O((n \log n)^{5/6})$ for $D = O(\sqrt{n \log n})$, and $O(D^{1/3} \cdot (n \log n)^{2/3})$ for larger D with high probability. These complexities are sub-linear whenever $D = o(n/\log^2 n)$, demonstrating that $o(n)$ exact SSSP computation is indeed possible for most network structures.

Following the introduction of Elkin's sub-linear algorithm for exact SSSP, subsequent research has explored ways to refine and build upon its core ideas. This section introduces some such refinements developed since the original work that improve the complexity, or apply under some specific settings. Table 2.2 summarizes different distributed shortest path algorithms.

Table 2.2: Summary of approximate/exact distributed SSSP algorithms in the CONGEST model. In [19], nodes are awake only in polylog(n) rounds.

Algorithm	**Randomized Algorithm (R)**	**Deterministic Algorithm (D)**
Exact (E)	Elkin [14] Ghaffari and Li [18] Forster *et al.* [17] Chechik and Mukhtar [11]	Ghaffari and Trygub [19]
Approximate (A)	Lenzen and Patt-Shamir [27] Nanongkai's [31] Becker *et al.* [6]	Henzinger *et al.* [22] Rozhoň *et al.* [35]

Ghaffari and Li [18] present improved randomized distributed algorithms for the exact shortest

paths problem in the *CONGEST* model. It achieves a time complexity of $\tilde{O}(n^{3/4}D^{1/4})$ for exact SSSP and is the first sublinear-time algorithm for directed graphs. It also provides another algorithm with a time complexity of $\tilde{O}(\max\{n^{3/4+o(1)}, n^{3/4}D^{1/6}\} + D)$.

Forster *et al.* [17] presents new randomized algorithms for the single-source shortest paths (SSSP) problem with non-negative edge weights in the *CONGEST* model. One algorithm achieves time complexity of $\tilde{O}(\sqrt{nD})$, and another achieves $\tilde{O}(\sqrt{n}D^{1/4} + n^{3/5} + D)$. These algorithms also work for directed graphs and in the Broadcast *CONGEST* model.

Chechik and Mukhtar [11] improve the time complexity of the single-source shortest path problem for weighted directed graphs (with non-negative integer weights) in the Broadcast *CONGEST* model. It devises a new randomized algorithm achieving an exact solution with a time complexity of $\tilde{O}(\sqrt{n}D^{1/4} + D)$.

Ghaffari and Trygub [19] present a low-energy deterministic distributed algorithm for computing exact SSSP in near-optimal $\tilde{O}(n)$ rounds, in a model where each node is awake for only polylogarithmic time. It presents a distributed version of Dijkstra's algorithm and also achieves near-optimal time and message complexities for exact SSSP with polylogarithmic messages per edge. This work also has implications for the All Pairs' shortest paths (APSP) problem.

2.3.3 All-Pairs Shortest Paths (APSP)

In the APSP problem, every node in the network needs to know its distance from every other node. In this thesis, we only study the SSSP problem, but we give a brief survey of the work on distributed APSP.

Agarwal *et al.* [3] presents a deterministic, distributed algorithm for computing all-pairs shortest paths (APSP) within edge-weighted graphs, applicable to both directed and undirected cases. Operating in the CONGEST model, their algorithm achieves a round complexity of $\tilde{O}(n^{3/2})$, where n represents the number of nodes. This work represents the first deterministic, distributed approach for the weighted APSP problem that achieves a sub-quadratic ($o(n^2)$) round complexity.

Agarwal and Ramachandran [1] introduce a new pipelined approach for computing all-pairs shortest paths (APSP) in a directed graph with non-negative integer edge weights (including zero weights) in the *CONGEST* model. Their deterministic distributed algorithm computes shortest paths

of distance where values are at most Δ for all pairs of vertices in at most $2n\sqrt{\Delta} + 2n$ rounds.

Agarwal and Ramachandran [2] presents a deterministic distributed APSP algorithm for weighted directed and undirected graphs running in $\tilde{O}(n^{4/3})$ rounds in the *CONGEST* model, an improvement over the previous $\tilde{O}(n^{3/2})$ bound in Agarwal *et al.* [3]. The core contributions are a faster deterministic blocker set construction and a new pipelined distance propagation method, both with potential for wider use in distributed algorithms.

Huang *et al.* [23] introduce a randomized (Las Vegas) algorithm achieving $\tilde{O}(n^{5/4})$ time for exact weighted All-Pairs Shortest Paths (APSP) in the distributed *CONGEST* model. This marks the first improvement over the naive bound for non-sparse networks. Their key innovation is the first application of the classic scaling technique to distributed shortest paths computation. This approach also yields an $\tilde{O}(n^{3/4}k^{1/2} + n)$ time algorithm for the k-source shortest paths problem, outperforming prior work for specific k values.

Bernstein and Nanongkai [8] introduce an $\tilde{O}(n)$-time Las Vegas randomized algorithm that solves the exact weighted all-pairs shortest paths problem for directed graphs (including those with zero or negative edge weights) in the *CONGEST* model. The runtime is near-optimal, matching the lower bound within polylogarithmic factors, and the algorithm employs a novel technique named *Random Filtered Broadcast*.

2.3.4 Discussion

In this thesis, we focus on the SSSP problem. As mentioned earlier, Elkin [14] was the first paper to give a sub-linear algorithm for the exact SSSP problem. Despite the fact that there were subsequent refinements to Elkin's algorithm, we chose in this thesis to implement and work with Elkin's algorithm, both because it is a celebrated and important result, and because the algorithm appears to be simpler than the subsequent refinements in [18, 17, 11], and therefore expected to be more practical.

Chapter 3

Distributed Shortest Path Algorithm of Elkin (2020) and Improvements

In this chapter, we present a comprehensive overview of the Elkin algorithm [14], and propose some variants. We begin in Section 3.1 with a set of preliminary notations and definitions essential for understanding the algorithm. Section 3.2 provides an overview of the algorithm, which can be divided into two main phases. The first phase of the algorithm focuses on computing the so-called k-shortcut hopset, and is detailed in Section 3.3, with an analysis of time complexity. Section 3.4 introduces the second phase of the algorithm. A thorough complexity analysis of the Elkin algorithm is presented in Section 3.5. Finally, Section 3.6 describes a small error in the description of the Elkin algorithm discovered during our implementation, our correction, an optimization in the implementation that reduces the number of rounds, and two variants of the Elkin algorithm that we propose. Our experiments, described in Chapter 4, demonstrate that our variants offer improved performance compared to the original version.

3.1 Preliminary definitions

Let $G = (V, E, w)$ be an undirected weighted graph, where V is a set of nodes, $E \subseteq V \times V$ is a set of edges, and each edge $e \in E$ has a weight of $w_G(e)$ (or w_e for short). For any pair (u, v) of nodes in G, several notations related to paths are as follows:

- $P_G(u, v)$ is the set of all simple paths in the graph between u and v.
- For a path $p \in P_G$, ℓ_p and $|p|$ are its length (i.e., sum of the weights) and number of hops, respectively.
- $\ell^{\text{SP}}(u, v)$ is the minimum length of a path between u and v, i.e.,

$$\ell^{\text{SP}}(u, v) = \min_{p \in P_G(u,v)} \ell_p$$

- A path $p \in P_G$ is a h-limited path if it has at most h hops (edges), i.e., $|p| \leq h$.
- $P_G^{(h)}(u, v)$ is the set of all h-limited paths between u and v in the graph G.
- $\ell^{\text{SP}(h)}(u, v)$ is the minimum length of a path with at most h hops between u and v, i.e.,

$$\ell^{\text{SP}(h)}(u, v) = \min_{p \in P_G^{(h)}(u,v)} \ell_p$$

In the sequel, key steps of the Elkin algorithm are illustrated with the CONUS network topology that has been used in several studies, see, e.g., [37, 41]. The CONUS network topology is a hypothetical fiber-optic backbone network for the continental United States (CONUS) with 75 nodes and 99 links. It was developed for the DARPA CORONET program to study dynamic multi-terabit core optical networks. Each node corresponds to a city location in the US, and weights represent distances between them, see Figure 3.1.

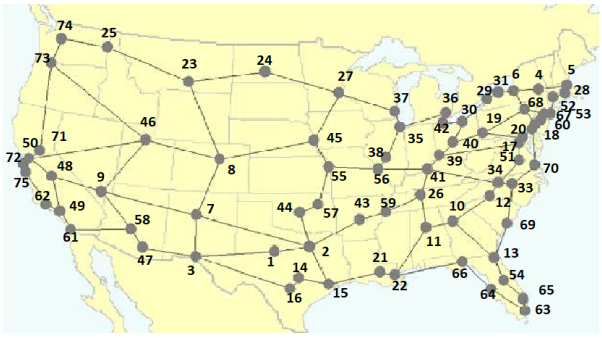


Figure 3.1: CONUS network

In the following examples, we re-draw the CONUS network using Gephi software [5], which

can be viewed in Figure 3.2.

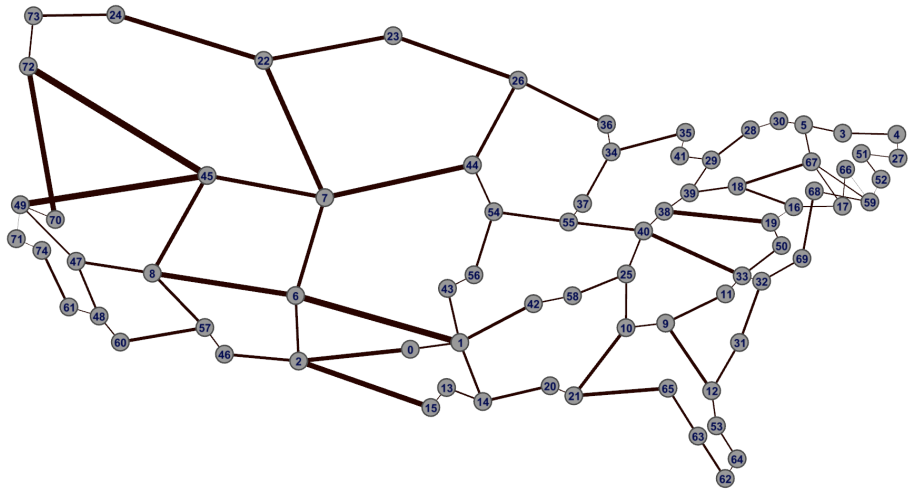


Figure 3.2: CONUS network drawn using Gephi software [5]; the darkness of the links is proportional to their weight.

Definition 1. *[Hop-diameter] The hop-diameter D^{H} of a graph is the maximum over all pairs of nodes in the graph of the number of hops in any min-hop path between them, i.e.,*

$$D^{\mathrm{H}} = \max_{u,v \in V} \Big(\min_{p \in P_G(u,v)} |p| \Big)$$

The diameter of a graph can be found using the All-Pairs-Shortest-Paths algorithm of Floyd-Warshall [16] in $O(|V|^3)$ time.

Example 1. *[Hop-diameter of CONUS network] As illustrated in Figure 3.3, the CONUS network has a hop diameter of 17, measured between nodes 27 and 61, which represents the longest minimum hop shortest path in the network between any two nodes.*

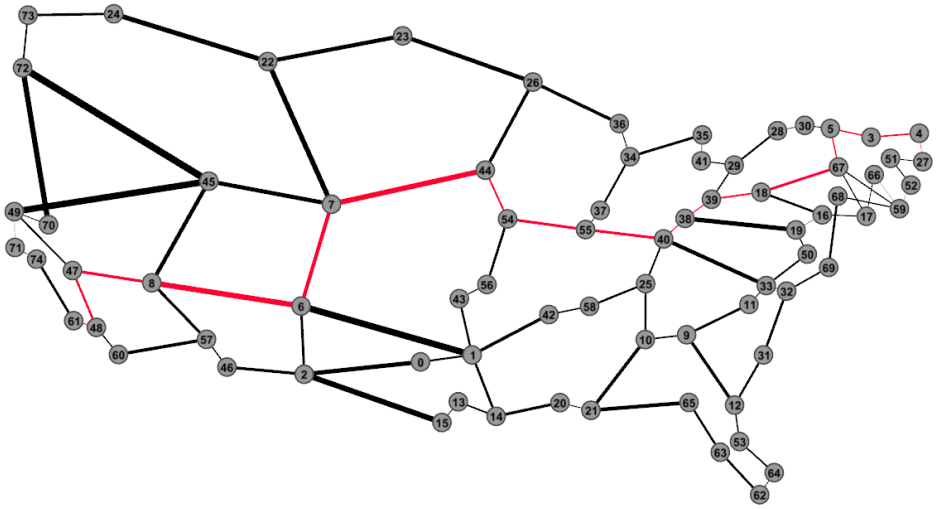


Figure 3.3: Hop-diameter of the CONUS network

Definition 2. *[Weighted diameter] The weighted diameter D^{W} of a graph is the longest minimum weighted-distance between any pair of nodes, i.e.,*

$$D^{\mathrm{W}} = \max_{u,v \in V} \ell^{\mathrm{SP}}(u, v)$$

Note that the weighted diameter may occur along a longer path (number of hops) than the hop diameter. This is a reason that the distributed Bellman-Ford needs to run for n rounds even in graphs of much smaller diameter.

Example 2. *[Weighted diameter] The weighted diameter of the CONUS network is* $D^{\mathrm{W}} = 6472.18$ *connecting node 63 and node 74, see Figure 3.4.*

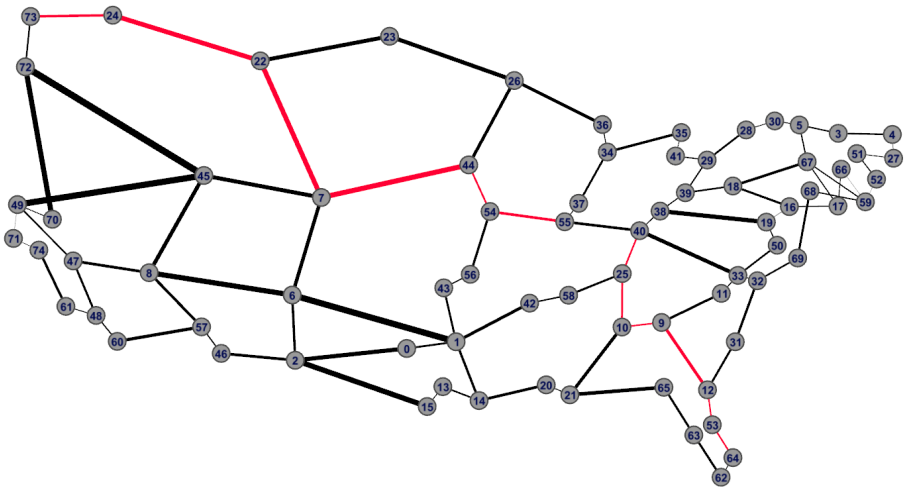


Figure 3.4: Weighted diameter of the CONUS network

We now turn our attention to hopsets. A hopset is a graph-theoretic object that is a set of edges that can be added to a graph to ensure that every pair of vertices has a path with at most a certain number of edges. It was first introduced in the context of approximate shortest paths. As we restrict our attention to exact shortest paths, we go directly to the definition of (exact) h-hopset. Informally, an exact h-hopset for a weighted graph G is a weighted edge set, whose addition to the graph guarantees that every pair of vertices has a path between them with at most h edges (hops) and whose length is exactly the length of the shortest path between the vertices. More formally, the definition is as follows.

Definition 3. *[Exact h-hopset [20]] An (exact) h-hopset for a weighted graph $G = (V, E, w)$ is a set of edges H such that*

$$\ell^{\mathrm{SP}(h)}_{G(H)}(u, v) = \ell^{\mathrm{SP}}_{G}(u, v) \qquad u, v \in V,$$

where $G(H) = (V, E \cup H, w)$, and ℓ_G (resp. $\ell_{G(H)}$) is the length of the shortest path computed over the set E of edges of G (resp. over the $E \cup H$ of edges of $G(H)$).

Weights of the edges are such that:

$$w_{G(H)}(e) = \ell^{\text{SP}}(u, v) \qquad e = (u, v) \in H$$
$$w_{G(H)}(e) = w(e) \qquad e \in E.$$

The parameter h is called the hopbound of the hopset. Edges from set H are called shortcuts in G.

A procedure for constructing the hopset is as follows:

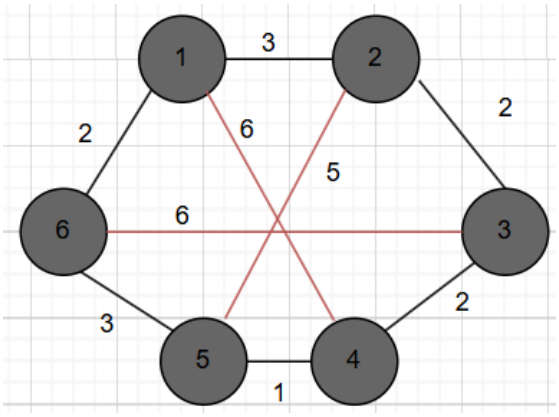
- First, we compute the shortest distances between every pair of nodes in the graph.
- Select the shortest paths with more than h hops.
- For each such path p, create a new edge $e = (u, v)$ to directly connect the two endpoints u and v of the path, with the weight equal to $\ell^{\text{SP}}(u, v)$.

Note that G may contain more than one h-hopset, see Example 3.

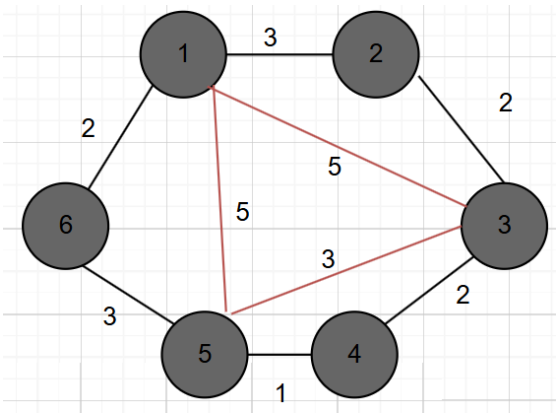
Example 3. *[Exact h-hopset] Consider graph $G = (V, E, w)$ with a set of nodes $V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$, a set of edges $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$, and the weight of the edges as described in Table 3.1. The set $H = \{e_7, e_8, e_9\}$ is a 2-hopset of graph G as shown in Table 3.1 and Figure 3.5a. Weights of the edges of the hopset are equal to: w_7, w_8, w_9 with values as described in Table 3.1. Another hopset is $H' = \{(1, 3), (3, 5), (5, 1)\}$.*

Graph $G = (V, E, w)$			2-hopset H			2-hopset H'		
Edges	**Connected nodes**	**Weights**	**Edges**	**Connected nodes**	**Weights**	**Edges**	**Connected nodes**	**Weights**
e_1	(1, 2)	3	e_7	(1, 4)	8	e_{10}	(1, 3)	5
e_2	(2, 3)	2	e_8	(2, 5)	5	e_{11}	(3, 5)	3
e_3	(3, 4)	2	e_9	(3, 6)	6	e_{12}	(5, 1)	5
e_4	(4, 5)	1						
e_5	(5, 6)	3						
e_6	(6, 1)	2						

Table 3.1: Graph G and two exact 2-hopsets



(a) 2-hopset H



(b) 2-hopset of H'

Figure 3.5: Two different 2-hopsets H and H'

Example 4. *[Exact h-hopset] $H = (V, \{e_1, e_2, e_3, e_4, e_5, \ldots, e_{15}\}, w)$ is a 16-hopset of the CONUS network $G = (V, E, w)$, see Table 3.2 and Figure 3.6 for the details.*

Edge e_i	**Connected nodes**	**Weight** $w(e_i)$
e_1	(3, 48)	5565.36
e_2	(3, 61)	5716.04
e_3	(4, 13)	3713.17
e_4	(4, 15)	3856.72
e_5	(4, 48)	5842.43
e_6	(4, 60)	5618.58
e_7	(4, 61)	5993.10
e_8	(5, 61)	5481.82
e_9	(15, 27)	3776.80
e_{10}	(27, 48)	5792.90
e_{11}	(27, 60)	5569.05
e_{12}	(27, 61)	5943.57
e_{13}	(48, 51)	5667.34
e_{14}	(51, 61)	5818.01
e_{15}	(52, 61)	5631.74

Table 3.2: Edges of the exact 16-hopset of CONUS network

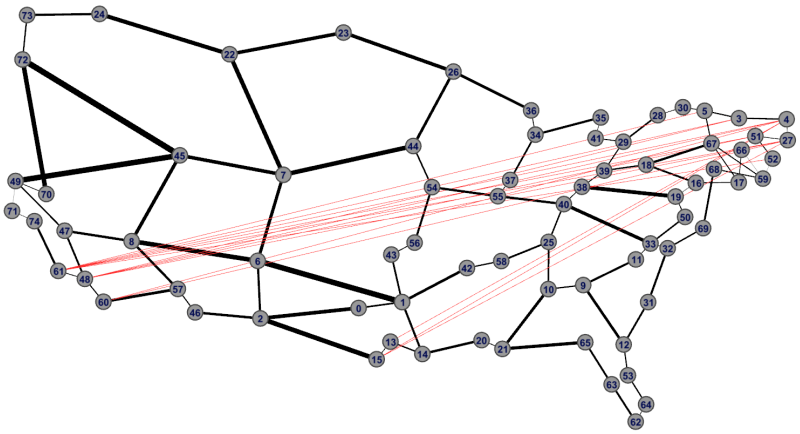


Figure 3.6: h-hopset of G where $h = 16$

Note that if H is an exact h-hopset for G, its weighted diameter is exactly the same as G, while its hop-diameter may be reduced because of the addition of edges.

In Elkin's algorithm, we first choose a random subset of vertices, which are denoted as V^{VIRT}, the set of *virtual nodes*, and then define a graph on these nodes, called the *virtual graph.*

Definition 4. *[Virtual graph - G^{VIRT}, Elkin [14]]*

Let $G = (V, E, w)$ be an undirected weighted graph, and let $n = |V|$. A graph $G^{\text{VIRT}} = (V^{\text{VIRT}}, E^{\text{VIRT}}, w_e)$ is a virtual graph of G if

$$V^{\text{VIRT}} \subseteq V,$$

$$E^{\text{VIRT}} = \{\{u, v\} \in V^{\text{VIRT}} \times V^{\text{VIRT}}:$$

$$\exists \text{ a path } p(u, v) \text{ in } G, \text{ with at most } c\sqrt{n}\ln n \text{ hops for a given constant } c\},$$

$$w_e, e \in E^{\text{VIRT}}, \text{ is the length of the shortest } (c\sqrt{n}\ln n)\text{-limited paths in } G.$$

Example 5. *[Virtual graph of CONUS network] Figure 3.7 depicts G^{VIRT} of the CONUS network, the red nodes are V^{VIRT} and blue edges are E^{VIRT}.*

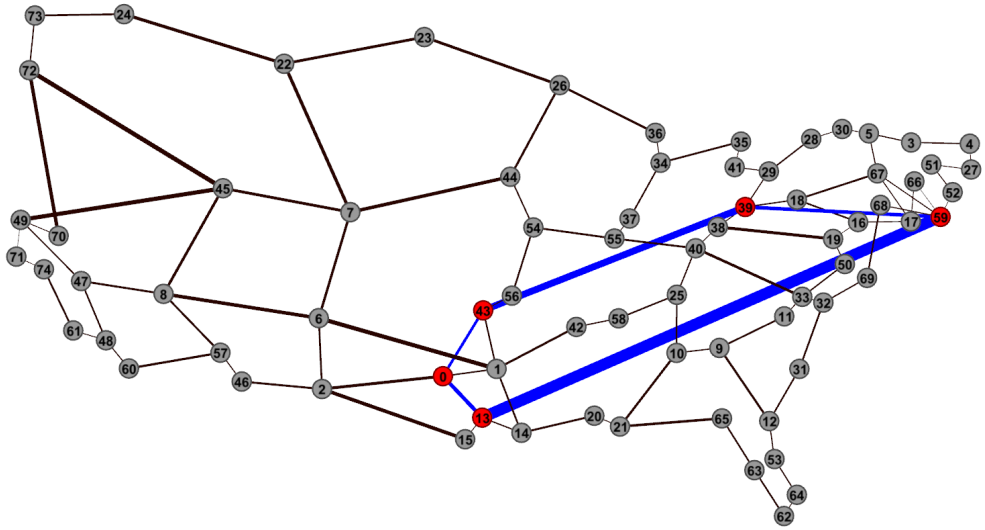


Figure 3.7: Virtual graph G^{VIRT} of CONUS network

We now go on with the definition of k-shortcut-hopset of G, which we define as a set of particular edges (differently from Elkin [14]) in order to be consistent with the previous definitions of hopsets. Moreover, Elkin gives an algorithm to compute the k-shortcut-hopset of the virtual graph G^{VIRT}, which we describe in Section 3.3.

Definition 5. *[k-shortcut-hopset of G] Let $G = (V, E, w)$ be a weighted undirected graph. For a positive integer parameter k, a k-shortcut-hopset (denoted by $H^{[k]}(G)$ or $H^{[k]}$ for short) is a set of edges such that:*

$$H^{[k]} = \{(u, v) \in V \times V : u \in S_G[k](v) \text{ or } v \in S_G[k](u)\},$$

where $S_G[k](v)$ = set of k closest (reachable) nodes to v in G w.r.t. weight, not including v.

Define:

$$G(H^{[k]}) = (V, E \cup H^{[k]}, w_{G(H^{[k]})}) \text{ where weights } w_{G(H^{[k]})} \text{ are defined as follows:}$$

$$w_{G(H^{[k]})}(e) = \ell^{\text{SP}}_G(u, v) \text{ if } e = (u, v) \notin E$$

$$w_{G(H^{[k]})}(e) = w_G(e) \text{ if } e = (u, v) \in E.$$

Example 6. *[k-shortcut-hopset of the virtual graph of CONUS network] Figure 3.8 here depicts the $H^{[k]}(G^{\text{VIRT}})$ of the CONUS network where $k = 6$.*

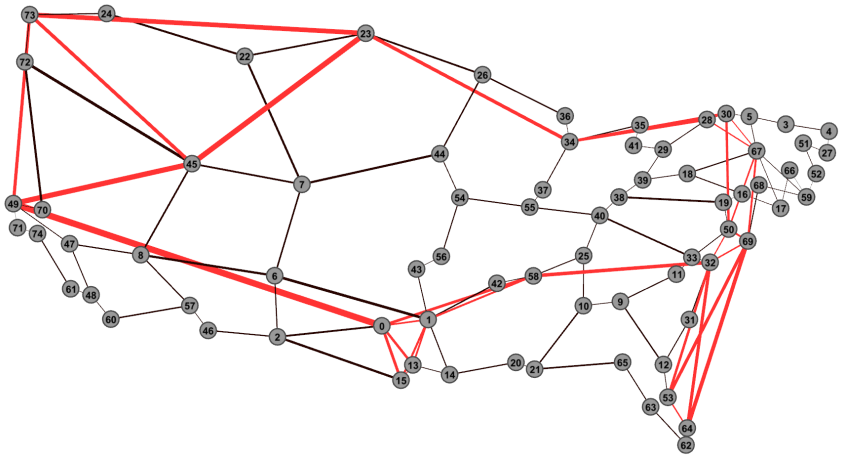


Figure 3.8: k-shortcut hopset $H^{(k)}$ of G^{VIRT} where $k = 6$

Definition 6. *[h-limited k-closest-virtual-node set $Q_G^{(h)}[k](v)$] In graph G, h-limited k-closest-virtual-nodes of a node v is a set that contains k-closest virtual nodes to v (not including v) that are at most h hops from v.*

Example 7. *[h-limited k-closest-virtual-node set $Q_G^{(h)}[k](v)$] Figure 3.9 shows $V^{\text{VIRT}} = \{0, 13, 39, 43, 59\}$, and $Q_G^{(h)}[k](v) = \{0, 13, 43\}$ where $v = 42$, $k = 6$, and $h = 4$. Note that there are only 3 virtual nodes that are within 4 hops of v.*

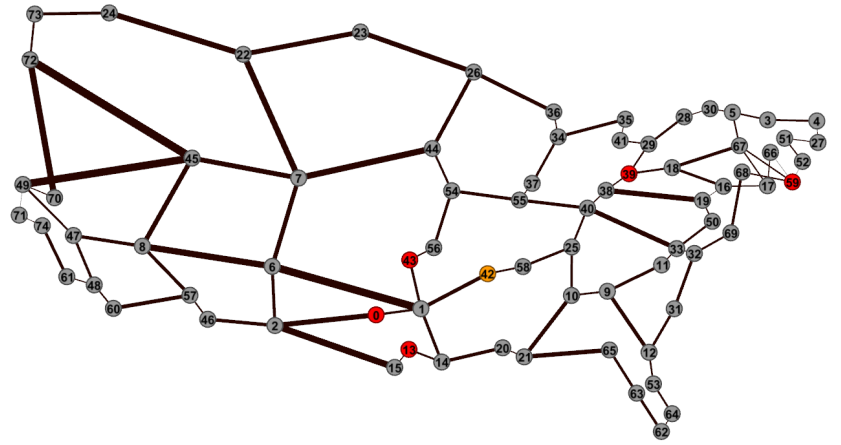


Figure 3.9: CONUS network with $v = 42$ and V^{VIRT} shown in red. The only three virtual nodes within 4 hops of v are $\{0, 13, 43\}$.

Table 3.3 summarizes the key concepts and their corresponding notations:

Table 3.3: Notation

Notation	**Definitions**
$G = (V, E, w)$	undirected weighted graph V set of vertices, E set of edges, w set of edge weights
n	number of nodes in graph G
$p(u, v)$	a path between u and v
$P(u, v)$	set of paths between u and v
$P^{(h)}(u, v)$	set of h-limited paths between u and v
$\|p\|$	number of hops of path $p \in P$
$\ell(u, v)$	estimation of the minimum length between u and v, not necessarily the minimum one
$\ell^h(u, v)$	estimation of the minimum length between u and v, on a path with no more than h hops
$\ell^{\text{SP}}(u, v)$	minimum length (weighted sum of the edges) of $u - v$ paths in G
$\ell^{\text{SP}(h)}(u, v)$	minimum length of h-limited $u - v$ path in G
s	source node (aka root)
$\delta(v)$	$\ell_G^{\text{SP}}(s, v)$ (in order to alleviate the notations when describing Phase 2 of Elkin's algorithm)
V^{VIRT}	a selected subset of V (see Section 3.3.2 for more details)
$G^{\text{VIRT}} = (V^{\text{VIRT}}, E^{\text{VIRT}}, w)$	virtual graph of G where: $E^{\text{VIRT}} = \{e = \{u, v\} \in V^{\text{VIRT}} \times V^{\text{VIRT}} : \exists$ a path $p(u, v)$ in G with at most $c.\sqrt{n}.\ln n$ hops for an appropriate constant $c\}$ $w_e =$ weight of $e \in E^{\text{VIRT}}$
$S_G[k](v)$	set of k closest (reachable) nodes to v in G w.r.t. weight and not including v

Continued on next page

Table 3.3 – continued from previous page

Notation	Definitions
$S_G^{(h-1)}[k](v)$	set of k closest (reachable) nodes to v within $h-1$ hops in G w.r.t. weight and not including v
$H^{(k)}$	a k shortcut-hopset of G where: $H^{(k)} = \{(u,v) \in E : u \in S_G[k](v) \text{ or } v \in S_G[k](u)\}$ $w(u,v) = \ell^{\text{SP}}(u,v)$ for $(u,v) \in H^{(k)}$ $G(H^{(k)}) = (V, E \cup H^{(k)}, w)$
$Q^{\text{MSG}}(v)$	queue containing messages received by v from the set of virtual nodes
$\text{MSG}_u \in Q^{\text{MSG}}(v)$	message received by node v from node u in $Q^{\text{MSG}}(v)$: $\text{MSG}_u \leftarrow (u,\ \ell(u,v),\ \lvert p(u,v)\rvert,\ \text{PARENT}(v))$
$h(u,v)$	the number of hops in a path between node u and v
$h^{(h)}(u,v)$	the number of hops in a path between node u and v within at most h hops
$\text{PTR}(u,v)$	the parent node where node u receive the shortest distance message to node v from
$Initiate_v$	a boolean status indicating whether node v has begun sending distance estimates in the distributed Bellman-Ford algorithm
$Depth(\tau)$	the depth in hops of the BFS tree τ of graph G
deg_{avg}	average node degree of graph G

Table 3.4: Notation of the Upcast, Broadcast, and Pipeline algorithms

Notation	**Definitions**
UPCAST$_v(u)$	TRUE if node v acknowledges that its child u is $Done$ for $Upcast$
UPCAST$_v$	TRUE if status of node v is $Done$ for $Upcast$
BROADCAST$_v(u)$	TRUE if node v acknowledges that its child u is $Done$ for $Broadcast$
BROADCAST$_v$	TRUE if status of node v is $Done$ for $Broadcast$
ALLSENT$_{\text{PARENT}}$	TRUE if a node acknowledges that the parent has sent an ALLSENT message
FROM-CHILDREN-QUEUE$_v$	the queue where node v stores all UPCAST messages received from its children
FROM-PARENT-QUEUE$_v$	the queue where node v stores all BROADCAST messages received from its parent

3.2 An overview of Elkin's algorithm

The basic approach in recent distributed approximate shortest path algorithms involves sampling about $\sqrt{n}$ virtual vertices, forming a *skeleton* graph G^{VIRT} on these vertices. Assume that the source node s is always one of the virtual vertices V^{VIRT}. An edge exists between two virtual vertices if there is a path between them in the original graph G with at most $c\sqrt{n}\log n$ hops, and the edge weight is the (weighted) length of the shortest such path. Once G^{VIRT} is built, the algorithm constructs an approximate hopset $H(\beta, \varepsilon)$ for G^{VIRT}, ensuring approximate distances between virtual vertices. A Bellman-Ford exploration with β iterations is then conducted on $G^{\text{VIRT}} \cup H$, which uses virtual vertices to estimate distances. However, this approach only provides approximate distances since $H(\beta, \varepsilon)$ is an approximate hopset.

The Elkin algorithm aims to produce exact shortest paths, and it bypasses constructing G^{VIRT} from scratch altogether by directly computing the exact hopset H. During the Bellman-Ford exploration, the algorithm computes only the necessary edges of G^{VIRT} on the fly. This method ensures exact distance calculations while maintaining efficiency, leveraging the exact hopset. The exploration proceeds by running Bellman-Ford in parallel from virtual vertices in G, updating distance estimates iteratively. The process is simpler and faster than previous algorithms, which rely on more complex approximate shortest path computations and hopset constructions.

In general, the core idea of the Elkin algorithm involves two phases to compute the shortest path from the source to the destination. The first phase constructs the k-shortcut hopset of the virtual graph denoted by $H^{(k)}(G^{\text{VIRT}})$. The second phase leverages the k-shortcut hopset to compute the shortest distance from the source to all other virtual nodes and later to compute the shortest paths from the necessary virtual nodes to the destination. By necessary virtual nodes, it means that the set of virtual nodes for which the shortest path from the source to the destination has to go through at least one of these nodes. In the next two sections, we describe Elkin's algorithm in detail.

3.3 Phase 1: Computing the k-shortcut hopset

3.3.1 Description

In the first phase of Elkin's algorithm, we compute the k-shortcut hopset of the virtual graph G^{VIRT}, i.e., the sets $S_G^{(h-1)}[k](v)$ for all vertices v in G. Each vertex chooses with a fixed probability q to be a virtual vertex. The values of q and k are functions of n and the hop diameter D of the graph and are described in Table 3.5 in Section 3.3.2.

As mentioned in the overview of the algorithm, Elkin's algorithm does not compute the virtual graph, but directly computes the k-shortcut hopset of the virtual graph, which means computing the k virtual vertices closest to v, and for each vertex v', the distance to v', the number of hops to v', and the next node on the shortest path to v'.

We proceed into three sub-phases for these computations. During the first sub-phase, we initialize the set $S_G^{(0)}[k](v)$ for every vertex $v \in V$. In the second sub-phase, we calculate the set $S_G^{(h-1)}[k](v)$. Finally, in the third sub-phase, we ensure that all virtual nodes contain the appropriate hopset edges of the k-shortcut hopset of G^{VIRT}.

3.3.1.1 Sub-phase 1.1. Initialization (lines 1-3 of Algorithm 2)

We set $S_G^{(0)}[k](v)$ as the set of k closest virtual vertices to each $v \in V$, based on 0-hop distance in G. For each node, we set the first element u of the set $S_G^{(0)}[k](v)$, to v, the hop count $h^{(i)}(v,u) = 0$, and the parent pointer $\text{PTR}(v) = \text{NIL}$. The distance $w^{(i)}(u,v)$ is set to 0 if v is virtual, and ∞ otherwise.

3.3.1.2 Sub-phase 1.2. Computation of the sets $S_G^{(h-1)}[k](v)$ (lines 4-22 of Algorithm 2)

The algorithm proceeds in $\sqrt{n}k$ *super-rounds*, (see line 4), each lasting k iterations (see line 5) and some local computation. The algorithm starts with estimates of $S_G^{(h-1)}[k](v)$, which are refined until they are exact. In the super-round h, every node sends its current estimate of $S_G^{(h-1)}[k](v)$ to all its neighbors. Under the *CONGEST* model, a node can only send $O(\log n)$ bits to each of its neighbors in one round. This is why $O(k)$ iterations are needed to send $S_G^{(h-1)}[k](v)$ to all the neighbors. Once a node receives estimates of these sets from its neighbors, it selects its estimates

of its own k closest neighbors that are within h hops and can compute $S_G^{(h)}[k](v)$ using the *tie-breaking* rule mentioned in Elkin [14], Section 3.1. To be more specific, in this rule, for each origin x that v receives, the distance estimate is computed using the minimum value across different paths. In the case of multiple paths with equal distances, v selects the tuple with the smallest hop count. If hop counts are equal, v chooses the tuple with the lexicographically smallest parent node. When comparing tuples from different origins, v applies the same hierarchical comparison rules and selects the k smallest tuples. At the end of $\sqrt{n}k$ super-rounds, each node has computed the set of its k closest virtual vertices that are within $\sqrt{n}k$ hops.

The same idea is also applied in Algorithm 4 where node v compares the distance estimate, denoted by $\ell^{h-1}(x,v)+w_G(v,u)$, including the path between node u and node v, against its current best path, denoted by $\ell^h(x,v)$. This is done in a hierarchical manner: First, checking if the distance is shorter, then if the hop count is lower, and finally, if the lexicographical order of the parent nodes is smaller. If any of these conditions are met, the algorithm returns TRUE, indicating that the path through u is preferred. Otherwise, it returns FALSE, meaning the current best path is still optimal.

3.3.1.3 Sub-phase 1.3. Sending hopsets of the virtual nodes to all virtual nodes (lines 23-33 of Algorithm 2)

The goal of this sub-phase is to make sure that every vertex $u' \in V^{\text{VIRT}}$ that belongs to the set $S_G^{(h-1)}[k](v')$ knows about this hopset edge (u', v'). The way to proceed is first to compute $S_G^{(h-1)}[k](v')$ for all virtual nodes $v' \in V^{\text{VIRT}}$, and then run an upcast and pipelined broadcast of all the edges in $S_G^{(h-1)}[k](v')$ for all $v' \in V^{\text{VIRT}}$ over all the edges of the auxiliary BFS tree τ of G. This is done in three parts.

In the first part, we build a BFS tree τ of G (line 23) described in Algorithm 3. Readers can refer to Peleg [33], Chapter 5, for details of the classical algorithm.

In the second part, each virtual node v' upcasts its $S_G^{(h-1)}[k](v')$ set to the source node s. The pseudocode is given in Algorithm 5. For the upcast algorithm, we keep running the whole algorithm until the source node receives all the messages $\text{MSG}(v')$. In every iteration, each node first checks whether its status is *Done* or not, if yes it sends the *Done* message to its parent. If not, it sends all hopset edges information from its $S_G^{(h-1)}[k](v)$ to its parent. When receiving messages from

its children, if the message is *Done* message, the node marks the message received from this child *Done*, while if the message is the hopset edge information, it adds to its $S_G^{(h-1)}[k](v)$.

In the third part, the source node broadcasts these sets to *all* nodes (virtual and non-virtual) using the BFS tree. The pseudocode is given in Algorithm 5. Message MSG_u^v of node v coming form u is defined as follows:

$$\text{MSG}_u^v \leftarrow (u, \ell(u,v), h(u,v), \text{PTR}(v,u)),$$

where u is one of the k closest virtual node to v, $\ell(u,v)$ is the weighted distance between u and v, $h(u,v)$ is the number of hops between u and v, and $\text{PTR}(v,u)$ is the node from which u received the message information. For the broadcast algorithm, the source node s starts initiating the broadcasting by sending all the messages to its children. Then, for each round, each node, except for the root node, receives the messages sent from its parent. If the message is of type *BroadCastDistance*, and the node v receiving the message itself is a virtual node, it stores the message distance in its $Q^{\text{MSG}}(v)$, i.e., the queue containing all messages of the closest virtual nodes. If the message is of type *AllSent*, the node marks its $\text{ALLSENT}_{\text{PARENT}}$ to *True*. In every round, for every node v, if $Q^{\text{MSG}}(v)$ is not empty, it takes one message from the queue and sends to its children. In this case, line 6, the node sends all the *BroadCastDistance* messages to its children. In another case, line 4, if it has sent all *BroadCastDistance* messages and $Q^{\text{MSG}}(v)$ is empty, and it receives the *AllSent* message from its parent, this node sends the *AllSent* message to its children.

Algorithm 2: Phase 1 - Synchronous Algorithm: Compute and broadcast k-shortcut hopset $H^{(k)}(G^{\text{VIRT}})$ for virtual graph G^{VIRT}

Data: graphs G and G^{VIRT}, source s, k
Result: $\forall v \in V$, outputs k-shortcut hopset $H^{(k)}(G^{\text{VIRT}})$ for virtual graph G^{VIRT}

/* Sub-phase 1.1: Initialization */
for $v \in V$ **do**
  $S_G^{(0)}[k](v) \leftarrow \{v\}$; $h^{(0)}(v,v) \leftarrow 0$; $\text{PTR}(v,v) = \text{NIL}$;
  if $v \in V^{\text{VIRT}}$ **then** $\ell^0(v,v) \leftarrow 0$ **else** $\ell^0(v,v) \leftarrow \infty$;

/* Sub-phase 1.2: Compute the set $S_G^{(h-1)}[k](v)$: k^{th} shortcut hopset construction */
for $h : 1 \dots \sqrt{n}k$ **do**
  for $i = 1 \dots k$ **do**
    for $v \in V$, *in parallel* **do**
      for $u \in$ { *neighbors of* v} *synchronously* **do**
        v sends i^{th} element of $S_G^{(h-1)}[k](v)$ to u, see below for the subsequent updates
  After u receives messages from all its neighbors:
  for $u \in V$, *in parallel* **do**
    $S_G^{(h)}[k](u) \leftarrow \emptyset$
    for $v \in$ *neighbors of* u **do**
      for $x \in S_G^{(h-1)}[k](u)$ **do**
        $h^{(h)}(x,u) \leftarrow h(x,u)$;
        $\ell^h(x,u) \leftarrow \ell(x,u)$; $\text{PTR}^{(h)}(x,u) \leftarrow \text{PTR}^{(h-1)}(x,u)$;
        /* Check whether u is closer to x through node v */
        if *closer-through*(v,u,x) (see Algorithm 4) **then**
          $\ell(x,u) \leftarrow \ell(x,v) + w_G(v,u)$; $\ell^h(x,u) \leftarrow \ell(x,u)$;
          $h(x,u) \leftarrow h^{(h-1)}(x,u) + 1$; $h^{(h)}(x,u) \leftarrow h(x,u)$;
          $\text{PTR}(x,u) \leftarrow v$;
          $\text{MSG}_x^u \leftarrow (x,\ \ell^h(x,u),\ h^{(h)}(x,u),\ \text{PTR}(x,u))$;
          $S_G^{(h)}[k](u) \leftarrow S_G^{(h)}[k](u) \cup \{x\}$
    $S_G^{(h)}[k](u) \leftarrow$ select the k-closest virtual vertices from the list ;

/* Sub-phase 1.3: Sending hopsets of V^{VIRT} */
$\tau \leftarrow$ BFS tree of G rooted at s using Algorithm 3
/* Upcast */
for *each leaf node* v **do**
  if $S_G^{(h)}[k](v) = \emptyset$ **then**
    $\text{UPCAST}_v \leftarrow \text{TRUE}$
while $\text{UPCAST}_s \neq True$ **do**
  for $v \in V^{\text{VIRT}}$, *in parallel* **do**
    v upcasts all computed edges $e = (v,u) : u \in S_G^{(h)}[k](v)$ to s of τ using Algorithm 5 ;

/* Broadcast */
s starts initiating the broadcasts to all of its children
while $\text{BROADCAST}_s \neq True$ **do**
  for $v \in V^{\text{VIRT}}$, *in parallel* **do**
    v broadcasts all computed edges to its children of τ using Algorithm 6

Algorithm 3: Constructing BFS tree τ of G

Data: original graph G, source s
Result: BFS tree τ of G

for $i \leftarrow 0 \dots V-1$ **do**
  $parent[i] \leftarrow NIL$
if $v = s$ **then**
  $initiate_v \leftarrow False$
if $v = s$ & $initiate_v =$ TRUE **then**
  v sends BFS formation message to all of its neighbors
  $initiate_v \leftarrow False$
for *each* MSG$_u$ *received by* v *from one (say* u*) of its neighbors* **do**
  if MSG$_u$ *is* BFS *formation* **then**
    if $parent(v) = NIL$ **then**
      $parent(v) \leftarrow u$
      v sends $YouAreMyParent$ message to u
      v sends BFS formation message to all of its neighbors except u
    else
      v discards MSG$_u$
  if MSG$_u$ *is* $YouAreMyParent$ **then**
    $chilren(v) \leftarrow children(v) \cup u$

Algorithm 4: closer-through(v, u, x): checking whether $\ell^h(x, u)$ needs to be updated by $\ell^{h-1}(x, v) + w_G(v, u)$, i.e., the distance from node u to x through v

Data: $\ell^{h-1}(x, v)$, $w_G(v, u)$, $\ell^h(x, u)$, $h^{(h-1)}(x, v)$, $h^{(h)}(x, u)$, PTR(x, v), PTR(x, u)
Result: TRUE or FALSE

if $\ell^{h-1}(x, v) + w_G(v, u) < \ell^h(x, u)$ **then**
  return TRUE
else if $\ell^{h-1}(x, v) + w_G(v, u) = \ell^h(x, u)$ **then**
  if $h^{(h-1)}(x, v) + 1 < h^{(h)}(x, u)$ **then**
    return TRUE
  else if $h^{(h-1)}(x, v) + 1 = h^{(h)}(x, u)$ **then**
    if PTR$(x, v) <$ PTR(x, u) **then**
      return TRUE
else
  return FALSE

Algorithm 5: Upcast algorithm for node v

Data: original graph G, BFS tree τ, node v
Result: source s contains all $\text{MSG}_v, \forall v \in V^{\text{VIRT}}$, terminate when root s is done

if $\text{UPCAST}_v(u)$ = TRUE *for all children* u *of* v **then**
  if $v = s$ **then**
    $\text{UPCAST}_v \leftarrow$ TRUE
    return
  else
    if $Q_v^{\text{MSG}} = \emptyset$ **then**
      $\text{UPCAST}_v \leftarrow$ TRUE
      v sends $UpcastDone$ message to its parent
      return
if $Q_v^{\text{MSG}} \neq \emptyset$ **then**
  Dequeue MSG_u from $Q^{\text{MSG}}(v)$
  v sends MSG_u to its parent
for *each* MSG_u *received by* v *from one (say* u*) of its children* **do**
  if MSG_u *is* $UpcastDone$ **then**
    /* v acknowledges that it receives a Done message from u */
    $\text{UPCAST}_v(u)$ = TRUE
  else
    Enqueue MSG_u in $Q^{\text{MSG}}(v)$

Algorithm 6: Broadcast algorithm for node v

Data: original graph G, source s, BFS tree τ, node v
Result: v' contains all $\text{MSG}(v'), \forall v' \in V^{\text{VIRT}}$; terminate when root s is done

$\text{BROADCAST}_v \leftarrow IsBroadCastDone()$
if $\text{BROADCAST}_v = \text{TRUE}$ **then**
  v sends $BroadcastDone$ message to its parent
if $Q^{\text{MSG}}(v) = \emptyset$ & $\text{ALLSENT}_{\text{PARENT}} = \text{TRUE}$ **then**
  v sends $AllSent$ message to all of its children
else if $Q^{\text{MSG}}(v) \neq \emptyset$ **then**
  Dequeue MSG from $Q^{\text{MSG}}(v)$
  v sends MSG to all of its children
for *each* MSG_u *received by* v *from one (say* u*) of its neighbors* **do**
  if MSG_u *is* $BroadcastDone$ **then**
    /* v acknowledges that it receives a Done message from u */
    $\text{BROADCAST}_v(u) \leftarrow \text{TRUE}$
  else if MSG_u *is AllSent* **then**
    /* v acknowledges that it receives All sent from its parent which is u */
    $\text{ALLSENT}_{\text{PARENT}} \leftarrow \text{TRUE}$
  else
    if $v \in V^{\text{VIRT}}$ **then**
      if $v \in S_G^{(h)}[k](u)$ **then**
        $S_G^{(h)}[k](v) \leftarrow S_G^{(h)}[k](v) \cup \{\text{MSG}_u\}$
    Enqueue MSG_u in Q_v^{MSG}

Algorithm 7: Is broadcast done: Checking the status whether node v is *Done* for *Broadcast* or not

Data: Node v
Result: TRUE or FALSE for status BROADCAST_v

if v *is a leaf node* **then**
  if v *receives all sent from parent* **then**
    return TRUE
else if $v = s$ **then**
  if $\text{BROADCAST}_v(u)$ = TRUE *for all children* u *of* v & $Q^{\text{MSG}}(v) = \emptyset$ **then**
    return True
else
  if $\text{BROADCAST}_v(u)$ = TRUE *for all children* u *of* v & v *receives all sent from parent* **then**
    return TRUE
return FALSE

3.3.2 Elkin's virtual node selection algorithm

As mentioned in the overview of the Elkin algorithm, a preliminary step consists of sampling about $\sqrt{n}$ virtual nodes. In Elkin [14], the virtual nodes are selected randomly based on a value q selected to satisfy some conditions related to the complexity, see Elkin [14], as described in Algorithm 8.

Algorithm 8: Get V^{VIRT} using probability q value

Data: original graph G, source s, probability q
Result: set of virtual nodes V^{VIRT}

```
V^VIRT ← {s}
for v ← 1 to n − 1 do
    q_1 ← random()
    if q_1 < q then
        V^VIRT ← V^VIRT ∪ {v}
return V^VIRT
```

The definition of q depends on two other constants c and k, which are mentioned in Elkin [14], pp. 17-18, suggests choosing as follows.

For parameter c, the suggestion is to select c to let $1 - \frac{1}{n^{c-3}}$ have high probability which can be chosen as 0.9. For k, the suggested value depends on the diameter value. If the diameter of the graph $D = \Omega(\sqrt{n}\log n))$, then $k = (\frac{n \ln n}{D})^{1/3}$, otherwise $k = n^{1/6} \ln^{1/6} n$. Similarly, parameter q is also selected based on the graph diameter D which is mentioned in Table 3.5. Since we can see q depends on q in the table, The algorithm first calculates q based on these two cases. It then compares the actual diameter D to the thresholds involving q which are $O(\max\{nq, \frac{\ln n}{q}\})$ and $\Omega(\sqrt{n}\log n))$ to determine which case applies and select the appropriate formula to compute q.

Table 3.5: Selection of parameters in Elkin's algorithm

Parameters	**Formula / Value**
c	c should be selected to let $1 - \frac{1}{n^{c-3}}$ have high probability, e.g., 0.9
k	$k = \begin{cases} n^{1/6} \ln^{1/6} n & \text{with small diameter value} \\ (\frac{n \ln n}{D})^{1/3} & \text{with larger diameter values, i.e., } D = \Omega(\sqrt{n} \log n)) \end{cases}$
q	probability to select an original node as a virtual node $q = \begin{cases} \sqrt{\frac{\ln n}{n}} & \text{with } D = O(\max\{nq, \frac{\ln n}{q}\}) \\ \frac{\ln n}{D} & \text{with } D \geq \max\{nq, \frac{\ln n}{q}\} \end{cases}$

3.4 Phase 2: Constructing shortest distances

The second phase of the Elkin algorithm has two sub-phases, the first sub-phase computes the shortest path from s to all virtual nodes of G^{VIRT} (lines 1 - 23), while the second sub-phase (lines 24 - 30) leverages the result from the first sub-phase to compute the shortest paths from s to all nodes in G (Line 24 - 30). Note that we assume that s is a virtual node regardless of the selection process, and the BFS tree τ of G rooted at the vertex s constructed in the first part of the algorithm (Algorithm 2) is available.

Sub-phase 2.1. Bellman-Ford exploration in $G^{\text{VIRT}}(H^k)$ (lines 1 - 23) of Algorithm 9

We now describe the first sub-phase in more detail. The idea of the algorithm is to run a Bellman-Ford exploration in $G^{\text{VIRT}}(H^k)$ starting at s. Note that we assume that s is artificially marked as a virtual node regardless of the virtual-node selection process. Thus, one can run the Bellman-Ford algorithm on G^{VIRT} starting from s. The time complexity of this solution will be $O(D)$ because the communication of G^{VIRT} is indeed done by the original graph G. However, leveraging the computed k-shortcut hopset of G^{VIRT}, the first part can reduce this complexity to $O(|V^{\text{VIRT}}|/k)$ thanks to Theorem 3.1 in [14]. To this end, the idea of the algorithm's first part is to run a Bellman-Ford exploration in $G^{\text{VIRT}} \cup H^k(G^{\text{VIRT}})$ starting at s.

To run a round of Bellman-Ford exploration in $G^{\text{VIRT}}(H^k)$, the algorithm breaks it into two sub-rounds to handle the two different types of edges in $G^{\text{VIRT}}(H^k)$: the hopset edges from $H^k(G^{\text{VIRT}})$ and the edges from the virtual graph G^{VIRT} itself. This strategy is key to achieving a sublinear time complexity without explicitly constructing the potentially dense virtual graph G^{VIRT}. To be more specific, the algorithm breaks one round into two sub-rounds as follows. To recall, each virtual node in $G^{\text{VIRT}}(H^k)$ has two types of neighbors. The first type consists of neighbors connected with an edge of $H^k(G^{\text{VIRT}})$ while the second one consists of neighbors connected with an edge in $E(G^{\text{VIRT}})$. Thus, in one round, a virtual node receives information from these two types of neighbors.

The information of the first neighbor set (neighbors connected with an edge in $H^k(G^{\text{VIRT}})$) is processed in the first part, i.e., lines 8 - 10. This part also leverages an auxiliary Breath-First-Search (BFS) tree τ of G rooted at a vertex s to transfer information between two neighbors in $H^k(G^{\text{VIRT}})$. The BFS tree is constructed in Algorithm 2, demonstrated in 3, and details of the algorithm is discussed in [33], pp. 49–68 for more details. $\text{MSG}^I(v)$ stores the best path via this first neighbor set of a node v.

In the second part of the iteration, the algorithm transfers information from G^{VIRT} nodes by sending information to a maximum number of $k\sqrt{n}$ hops from a virtual node, as this is the maximum number of hops between two virtual nodes. At this point, $\delta(v)$ stores the best path via this second neighbor set of a node v.

At the end of each super round, the algorithm selects the minimum value between two auxiliary distances and store this value as the current minimum shortest distance between the source (s) and all the virtual nodes which is $\delta(v')$ where $v' \in V^{\text{VIRT}}$ (line 21). In the next super round, all virtual nodes broadcast these distances to the root node of the BFS tree to make a comparison between the shortest distances in the current super round and the shortest distances in the previous super round. If all the virtual nodes do not achieve a better distance compared to the previous super round, the algorithm stops updating the shortest paths from the source to the virtual nodes.

Sub-phase 2.2. $\sqrt{n}$ rounds of the B-F from every vertex (lines 24 - 30) of Algorithm 9

Once the super rounds of the algorithm finish, the algorithm transitions to the second phase, which begins on line 24. In this phase, the algorithm runs a series of Bellman-Ford (B-F) computations, performing $\sqrt{n}$ rounds of B-F from every virtual node v' in the set V^{VIRT}. The purpose of these

B-F computations is to calculate the final shortest distances from the source node to all other nodes in the graph. The B-F algorithm is a classic technique for finding shortest paths, and by running it repeatedly from each virtual node, the algorithm can propagate distance information throughout the entire graph. By running this process for each virtual node, the algorithm can propagate distance information from the source to all other nodes in the graph. The $\sqrt{n}$ rounds are necessary to ensure that the shortest paths are correctly computed, as well as to guarantee the time-complexity of the algorithm. Once this procedure completes, the algorithm has successfully calculated the final shortest distances from the source to all nodes, and the computation is finished.

Algorithm 9: Synchronous Algorithm: Compute Distances

Data: G, k-shortcut hopset $H^{(k)}(G^{\text{VIRT}})$, source s, BFS tree of G rooted at s τ
Result: $\ell^{\text{SP}}(s,v) = \delta_v$ is computed for $\forall v \in V$.

/* Sub-phase 2.1 */
Assume that $s \in V^{\text{VIRT}}$;
$\delta_s \leftarrow 0$ and $\delta_v \leftarrow \infty$ for all $v \in V \setminus \{s\}$;
for $h \leftarrow 0 \ldots \frac{\sqrt{n}\log n}{k}$ **do**
  /* First part */
  for $v' \in V^{\text{VIRT}}$, *in parallel* **do**
    $\delta_{old}(v') \leftarrow \delta^I(v') \leftarrow \delta(v')$;
    v' upcasts its estimate δ to s on τ ;
  s broadcasts δ_v for all $v \in V^{\text{VIRT}}$ to the entire graph G;
  for $v' \in V^{\text{VIRT}}$, *in parallel* **do**
    /* v' hears an estimate $\delta(u')$ of its neighbor u' in the $H^{(k)}(G^{\text{VIRT}})$ */
    for $u' \in S_G^{(\sqrt{n}k)}[k](v')$ **do**
      $\delta^I(v') \leftarrow \min(\delta^I(v'), \delta(u') + w^{(\sqrt{n}k)}(v', u'))$;
  /* Second part */
  for $v' \in V^{\text{VIRT}}$, *in parallel* **do**
    for $u \in$ *neighbors of* v' **do**
      v' sends $\delta(v')$ to u
  for $j \leftarrow 1 \ldots \sqrt{n}$ **do**
    for $v \in V$ **do**
      for $x \in$ *neighbors of* v *in* G **do**
        if $\delta(x) + \omega_G(v,x) < \delta(v)$ **then**
          $p(v) \leftarrow x$
          $\delta(v) \leftarrow \delta(x) + \omega_G(v,x)$
      v sends $\delta(v)$ to all of its neighbors
  $\delta(v') = \min\{\delta^I(v'), \delta(v')\}$ for all $v' \in V^{\text{VIRT}}$;
  if *for all virtual nodes* $\delta(v') = \delta_{old}(v')$ **then**
    break;
/* Sub-phase 2.2 */
for $j \leftarrow 1 \ldots \sqrt{n}$ **do**
  for $v \in V$ **do**
    for $x \in$ *neighbors of* v *in* G **do**
      if $\delta(x) + \omega_G(v,x) < \delta(v)$ **then**
        $p(v) \leftarrow x$
        $\delta(v) \leftarrow \delta(x) + \omega_G(v,x)$
    v selects and sends the smallest $\delta(v)$ to all of its neighbors

3.5 Complexity analysis

We will dive deep into the complexity analysis of Elkin's algorithm, which comprises Algorithm 2 and Algorithm 9 (which in turn use other algorithms as sub-routines). We refer to our pseudocode, and sometimes give the analysis in terms of the number of virtual nodes, so that our analysis also holds for the Elkin variants we introduce in Sections 3.6.3 and 3.6.4 in which the number of virtual nodes may be smaller than the $\sqrt{n}\log n$ virtual nodes specified by Elkin.

Algorithm 2:

Sub-Phases 1.1 and 1.2 constitute the k-shortcut hopset computation. Clearly, Sub-Phase 1.1 takes 1 round and sends 0 messages. In Sub-phase 1.2, from lines 4-22 we can see that after $O(\sqrt{n}k)$ super-rounds, the $H^k(G^{\text{VIRT}})$ is constructed and every virtual node v' knows its $Q_G^{(\sqrt{n}k)}[k](v)$. Each super-round contains k rounds of communication (lines 5-8). The rest of Sub-Phase 1.1 is local computation. Therefore, this sub-phase uses $O(\sqrt{n}k^2)$ rounds. We see that in each round, every node sends a message to each of its neighbors, so the total number of messages sent is upper bounded by the number of rounds multiplied by $|E|$, the number of edges in the graph. To get a more precise bound that also applies to our Elkin variants, we see that if the number of virtual nodes is less than k, then the number of rounds in a super-round is $min\{|V^{\text{VIRT}}|, k\}$, leading to a bound of $\sqrt{n}k \times \min\{|V^{\text{VIRT}}, k\} \times |E|$ on the number of messages sent in this sub-phase.

In Sub-phase 1.3, before the upcast and broadcast part occurs, the algorithm constructs the BFS tree τ rooted at s on line 23; this step takes $O(D)$ rounds and sends $\Theta(E)$ messages. After that, from lines 24-33, the algorithm conducts the upcast and broadcast of all the computed edges of the $H^k(G^{\text{VIRT}})$ over the BFS tree. It is known that upcasting or broadcasting ℓ items on a tree of depth d needs $O(d) + \ell$ messages [33]. Since each virtual node has at most k items to be upcast, the total number of items involved in the upcast and broadcast is $|V^{\text{VIRT}}| \cdot k$, and the number of rounds for the upcast and broadcast is $O(Depth(\tau) + |V^{\text{VIRT}}| \times k)$. In each round, messages are sent along the edges of the tree τ, so the total number of messages is upper bounded by the number of rounds multiplied by n. Since in Elkin's algorithm, the number of virtual nodes is $\sqrt{n}\log n$, Sub-Phase 3 uses $O(Depth(\tau) + \sqrt{n}k\log n) = O(D + \sqrt{n}k\log n)$ rounds since $O(Depth(\tau)) = O(D)$.

Therefore, Algorithm (2) takes $O(D + \sqrt{n}k^2 + \sqrt{n}k\log n)$ rounds in all.

Algorithm 9

From lines 1-2, the algorithm initialized δ_v for all $v \in V$, this part takes 1 round and 0 messages. Next, from lines 3-23 Sub-phase 3.4 performs $O(\frac{\sqrt{n}\log n}{k})$ super rounds. In each super round, in the first part, lines (4-7) consist of an upcast and broadcast of a single estimate $\delta(v')$ for each virtual node v'. Therefore, the total number of items to be upcast and broadcast on the tree τ is $|V^{\text{VIRT}}|$. As mentioned earlier, this needs $|V^{\text{VIRT}}| + Depth(\tau)$ rounds, and the number of messages is upper bounded by the number of rounds multiplied by n. The rest of the first part is local computation.

In the second part, lines 11-20 constitute $\sqrt{n}+1$ rounds with each node sending a message to its neighbors in each round, for a total of $|E|$ messages being sent in each round.

In total, Sub-Phase 2.1 uses $O(|V^{\text{VIRT}}|+Depth(\tau)+\sqrt{n})\times O(\frac{\sqrt{n}\log n}{k})$ rounds and $O((|V^{\text{VIRT}}|+Depth(\tau))\times n+\sqrt{n}\times|E|)\times O(\frac{\sqrt{n}\log n}{k})$ messages. Sub-Phase 2.2 (lines 24-30) of Algorithm 9, takes $O(\sqrt{n})$ rounds and in each round, every node sends a message to each of its neighbors for a total of $\sqrt{n}|E|$ messages in all.

Since in Elkin's algorithm, the number of virtual nodes is $\sqrt{n}\log n$, and noting that $O(\sqrt{n}\log n)$ asymptotically dominates $O(\sqrt{n})$, and $Depth(\tau) = O(D)$, the time complexity for this segment of the algorithm becomes $O(D+\sqrt{n}\log n)\times O\left(\frac{\sqrt{n}\log n}{k}\right)$. Therefore, the complexity of Algorithm 9 is $O(D+\sqrt{n}\log n)\times O\left(\frac{\sqrt{n}\log n}{k}\right)$.

Sub-linear time complexity

The total time complexity of Elkin's algorithm (Algorithms 2 and 9) then is:

$$O\Big(D+\sqrt{n}k^2+\sqrt{n}k\log n\Big)+O\Big(D+\sqrt{n}\log n\Big)\times O\Big(\frac{\sqrt{n}\log n}{k}\Big)$$
$$= O\Big(D.\frac{\sqrt{n}\log n}{k}+\sqrt{n}.k^2+\frac{n.(\log n)^2}{k}+\sqrt{n}.k.\log n\Big).$$

Elkin [14] describes how to choose the parameter k to achieve sub-linear time complexity. When we set $k = n^{1/6}\log^{2/3} n$ with smaller diameter value $D = O(\sqrt{n}.\log n)$, we achieve the sublinear time of the algorithm: $\tilde{O}(n^{5/6}.(\log n)^{4/3})$.

For larger diameter, $D = \Omega(\sqrt{n}\log n)$ and we set $k = (D.\log n)^{1/3}$, the running time of the algorithm is $\tilde{O}(D^{2/3}.\sqrt{n}.(\log n)^{2/3})$ which is also sublinear when $D = o(\frac{n^{\frac{3}{4}}}{\log n})$.

Therefore, the algorithm overcomes the linear time complexity of traditional methods like Distributed Bellman-Ford through careful parameter selection to achieve a sublinear time complexity in a broad range of graph diameters.

The time and message complexity for the main parts of Algorithm (2) and Algorithm (9) are summarized in Table 3.6.

Table 3.6: Complexity analysis of distributed Elkin's algorithm components

Algorithm component	**Round complexity**	**Message complexity**
k-shortcut hopset construction (lines 1-22 Algorithm 2)	$O(\sqrt{n} \cdot k^2)$	$\sqrt{n} \cdot k \cdot \min(\lvert V^{\text{VIRT}}\rvert, k) \cdot \lvert E\rvert$
BFS tree construction (lines 23 Algorithm 2)	$O(D)$	$\Theta(\lvert E\rvert)$
Upcast & Broadcast (lines 24-33 Algorithm 2)	$O(Depth(\tau) + \lvert V^{\text{VIRT}}\rvert \cdot k)$	$O(Depth(\tau) + \lvert V^{\text{VIRT}}\rvert \cdot k) \cdot n$
Computing shortest distances (Algorithm 9): Initialization (lines 1-2) Upcast-Broadcast (lines 4-10) $\sqrt{n}$ rounds Bellman-Ford (lines 11-20) Update $\delta(v')$ (lines 21-23)	$\begin{pmatrix} O(1) \\ +O(Depth(\tau) + \lvert V^{\text{VIRT}}\rvert) \\ +O(\sqrt{n}) \\ +O(1) \end{pmatrix} \times O(\frac{\sqrt{n}\log n}{k})$	$\begin{pmatrix} 0+ \\ (Depth(\tau) + \lvert V^{\text{VIRT}}\rvert) \cdot n \\ +\sqrt{n} \cdot \lvert E\rvert \\ +0 \end{pmatrix} \times O(\frac{\sqrt{n}\log n}{k})$
$\sqrt{n}$ rounds Bellman-Ford (lines 24-30 Algorithm 9)	$O(\sqrt{n})$	$O(\sqrt{n} \cdot \lvert E\rvert)$

3.6 Improvements to Elkin's algorithm

In this section, we describe four different improvements we made to the Elkin algorithm [14]. The first one, Section 3.6.1, is indeed a correction to a small error we found in the algorithm. Second, we optimize the upcast and broadcast processes through our *Pipelined Upcast-Broadcast* technique (Section 3.6.2). Third, we introduce Elkin-R (Section 3.6.3), which employs different probability distributions for virtual node selection. Finally, we present Elkin-D (Section 3.6.4), implementing a distributed selection mechanism where virtual nodes are separated by a number of hops.

3.6.1 A small error in the Elkin algorithm and the correction

In Elkin's algorithm, we identified a small but subtle error in the description of the construction of the k-shortcut hopset, which leads in some situations to incorrect shortest paths being stored.

Elkin starts this description as follows:

> *For a vertex $v \in V$, let $S_G[k](v)$ (or, shortly, $S[k](v)$, when G can be understood from the context) denote the set of k closest (reachable) vertices to G, not including v.*

Then later in describing how to compute the k-shortcut hopset, Elkin writes:

> *At the beginning of each super-round i, $i = 0, 1, \ldots$, every vertex v maintains the set $S^{(i)}(v) = S'^{(i)}_G[k](v)$, defined as the set of k closest selected (aka, virtual) vertices $v' \in V'$ to v, closest with respect to i-limited distance in G. In other words, if one orders virtual vertices $v'_1, v'_2 \ldots, v'_N$ in order of non-decreasing i-limited distance from v, then $S^{(i)}(v)$ contains the first k elements in this ordering. It is also required that these distances be $< \infty$.*
>
> *In super-round i, every vertex v sends to all its neighbors its entire set $S^{(i)}(v)$. Then v selects the k smallest estimates among those that it received, using appropriate tie-breaking rules. As a result, the vertex v computes its set $S^{(i+1)}(v)$.*

We note that $S^{(i)}(v)$ is exactly the set $Q^i_G[k](v)$ in our pseudocode. Together, the above statements imply that the sets $S^{(i)}(v)$ never contain v, as otherwise, they would always contain v, but then we would have only $k-1$ vertices that are different from v. We also point out that leaving v out of $S^{(i)}(v)$ means that we cannot start with $i = 0$, and must start with $i = 1$.

However, the above algorithm creates a problem in the correct computation of the k-shortcut hopset in some cases. In particular, suppose the virtual node x is among the k closest nodes to virtual node v with respect to h-limited distance, and assume u is the node immediately after v on the h-limited shortest path from node v to x. The Elkin algorithm for the hopset construction is based on the idea that then x must be among the k closest nodes to u with respect to $(h-1)$-limited distance. Thus when u sends its set $S^{h-1}_G[k](u)$ to its neighbor v, the node v would find out about this h-limited shortest path to x. However, it may be that v itself is in the set $S^{h-1}_G[k](u)$ and the cost of the (u, v) edge is less than the cost of the $(h-1)$-limited path from u to x. Thus it is possible that u includes v in $S^{h-1}_G[k](u)$ but not x. So v does not find out about this $v - x$ path via u from u.

Note that this situation can happen only if the following three conditions are met:

(i) v is one of the k closest nodes to u with respect to $(h-1)$-limited distance

(ii) x is the $k+1^{st}$ closest node to u, and

(iii) u is a non-virtual node. Indeed, if there are $\geq k+1$ nodes ahead of x in the ordering of closest nodes to u with respect to $(h-1)$-limited distance, since only one of them is v, there are k nodes ahead of x in the ordering of closest nodes to v with respect to h-limited distance, a contradiction to our assumption that x is among the k closest nodes to virtual node v. On the other hand, if there are $\leq k$ nodes ahead of x in the ordering of closest nodes to u within $h-1$ hops, then $x \in S_G^{h-1}[k](u)$ and the omission error by u does not occur. This shows that (1) and (2) must hold for the error to occur.

Next, suppose u is a virtual node. Then $u \notin S_G^{h-1}[k](u)$, but u is closer to v than x, so even though v may only get $k-1$ nodes different from itself (and from u) from this set, it cannot be that x is among the k closest nodes to v. This shows that (3) must also hold for the error to occur.

Thus, this error will be exhibited only in some very specific situations. We illustrate how this error can occur in an example graph shown in Figure 3.10, which is part of the CONUS network. Note that the virtual nodes are marked red. Table 3.7 shows the S sets computed and propagated to neighbors by each node. For example, $S_G^{(1)}[3](3)$ includes 16^3 which means that for node 3, the (at most) 3 closest virtual nodes within 1 hop are nodes 16 and 1, and the next hop on the shortest 1-hop path from 3 to 16 is 16. Similarly, $S_G^{(3)}[3](3)$ is the set $\{1, 16, 2\}$, which are the 3 closest virtual vertices of node 3, and the next hop on the shortest 3-hop path from 3 to 2 is 1.

With $k = 3$ on the graph shown in Figure 3.10, node 1 is one of the 3 closest nodes within 3 hops to node 14, via the path 14-15-2-1. However, node 14 incorrectly receives the shortest distance from node 1 via node 16, rather than the correct path through node 15. This error occurs because (1) node 14 is one of the 3 closest virtual nodes within 3 hops to node 15 (2) Node 1 is the 4th closest node within 2 hops to node 15, while (3) node 15 is not a virtual node. So node 15 does not include node 1 in its set $S_G^{(2)}[3](15)$, and node 14 does not find out about the path 14-15-2-1. As a result, when constructing the k-shortcut hopset, node 14 calculates an incorrect shortest path: it uses the path 14-16-3-1 instead of the actual shortest path 14-15-2-1.

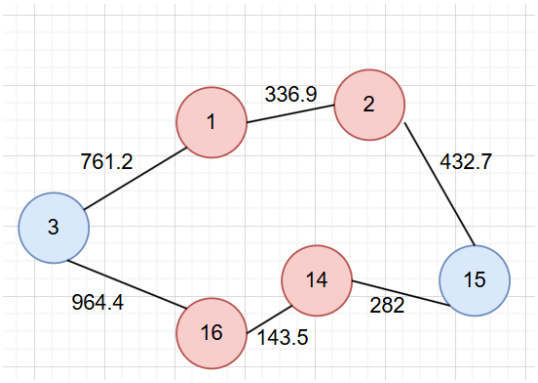


Figure 3.10: Part of the CONUS network graph

Table 3.7: The sets $S_G^{(h)}[3](v)$ for the graph in Figure 3.10 using Elkin's description.

Node	$S_G^{(1)}[3](v)$	$S_G^{(2)}[3](v)$	$S_G^{(3)}[3](v)$
1	2^2	$2^2, 16^3$	$2^2, 14^2, 16^3$
2	1^1	$1^1, 16^3$	$1^1, 14^{15}, 16^1$
3	$1^1, 16^{16}$	$1^1, 16^{16}, 2^1$	$1^1, 16^{16}, 2^1$
14	16^{16}	$16^{16}, 2^{15}$	$16^{16}, 2^{15}, 1^{16}$
15	$2^2, 14^{14}$	$2^2, 14^{14}, 16^{14}$	$2^2, 14^{14}, 16^{14}$
16	14^{14}	$14^{14}, 1^3$	$14^{14}, 1^3, 2^{14}$

3.6.1.1 A correction to the error

We now describe a simple fix to the error. The set $S_G^{(h)}[k](v)$ must always include v and we initialize $S_G^{(0)}[k](v) = \{v\}$. However, when we want to find a k-shortcut hopset, we must look for and maintain $k+1$ closest vertices, and only omit v at the end of the computation of the k-shortcut hopset. Table 3.8 shows the computation of the 3-shortcut hopset for the graph in Figure 3.10; we store at most 4 closest vertices at every stage of the computation.

Table 3.8: The sets $S_G^{(h)}[4](v)$ for the graph in Figure 3.10 using our correction to the algorithm

Node	$S_G^{(1)}[4](v)$	$S_G^{(2)}[4](v)$	$S_G^{(3)}[4](v)$
1	$1^1, 2^2$	$1^1, 2^2, 16^3$	$1^1, 2^2, 14^2, 16^3$
2	$1^1, 2^2$	$1^1, 2^2, 16^3$	$2^2, 1^1, 14^{15}, 16^1$
3	$1^1, 16^{16}$	$1^1, 16^{16}, 2^1, 14^{16}$	$1^1, 16^{16}, 2^1, 14^{16}$
14	$14^{14}, 16^{16}$	$14^{14}, 16^{16}, 2^{15}$	$16^{16}, 2^{15}, 1^{15}, 14^{14}$
15	$2^2, 14^{14}$	$2^2, 14^{14}, 16^{14}, 1^2$	$2^2, 14^{14}, 16^{14}, 1^2$
16	$16^{16}, 14^{14}$	$16^{16}, 14^{14}, 1^3$	$16^{16}, 14^{14}, 1^3, 2^{14}$

We now prove that the above correction leads to the correct computation of the k-shortcut hopset.

Theorem 3.1. *Suppose x is one of the k-closest virtual nodes to v, with $x \neq v$ such that there is an h-limited shortest path from v to x via a neighbor of v, namely u. Then x is one of the $k+1$ closest virtual nodes to u that are within $h-1$ hops to u.*

Proof. Let the $k+1$ closest virtual vertices within $h-1$ hops to u be $x_1, x_2, ..., x_{k+1}$. Suppose for the purpose of contradiction that $x \notin x_1, x_2, ..., x_{k+1}$. Note that there is a $(h-1)$-limited path from u to x, since we assume there is an h-limited path from v to x via u. Therefore, it must be that $\forall i \in \{1, 2, ..., k+1\}$:

$$w(u, x_i) < w(u, x)$$

Since there is a shortest path from v to x via u by assumption, we know:

$$w(v, x) = w(v, u) + w(u, x)$$

Also $\forall i \in \{1, 2, ..., k+1\}$, we have:

$$d(v, x_i) \leq w(v, u) + w(u, x_i) < w(v, u) + w(u, x) = w(v, x)$$

That is, $x_1, ..., x_{k+1}$ are all closer to v than x. Of these $k+1$ nodes, at most one is v itself.

We conclude that there are at least k virtual nodes different from v that are closer to v than x with respect to h-limited distance. This contradicts the assumption that x is one of the k closest virtual nodes within h hops to v with $x \neq v$. □

3.6.2 Pipelined upcast-broadcast

To speed up the implementation, we employ pipelining, allowing the *Upcast* and *Broadcast* phases to overlap and execute concurrently[1], as detailed in Algorithm 12. The key optimization begins at the source node s, which initiates the *Broadcast* phase incrementally as it receives initial *Upcast* messages, rather than inefficiently waiting for the entire *Upcast* from all subtrees to complete. This early start ensures the *Broadcast* message spreads rapidly downwards, because internal nodes accelerate the pipeline: upon receiving a message from their parent, they instantly forward it to their children (Algorithm 12, lines 21-22), without needing to wait for their Upcast processing to be complete. This parallel execution creates overlapping waves of *Upcast* data moving up and *Broadcast* data moving down, significantly reducing the communication rounds. An example of this pipelined algorithm is shown in Section A.1.

Algorithm 10: Is upcast done for Upcast-Broadcast

```
   Data: Node v
   Result: TRUE or FALSE for status UPCAST_v
1  if UPCAST_v(u) = TRUE for all children u of v then
2      if v = s then
3          return TRUE
4      else
5          if FROM-CHILDREN-QUEUE_v is empty then
6              return TRUE
7  if v is leaf node  &  FROM-CHILDREN-QUEUE_v = ∅ then
8      return TRUE
9  return FALSE
```

[1]This assumes that links are full-duplex, which may not be the case in some wireless networks.

Algorithm 11: Is broadcast done for Upcast-Broadcast

Data: Node v
Result: TRUE or FALSE for status BROADCAST_v

```
if v is leaf node then
    if v receives all sent from parent then
        return TRUE
else if v = s then
    if BROADCAST_v(u) = TRUE for all children u of v & FROM-PARENT-QUEUE_v = ∅ then
        return TRUE
else
    if BROADCAST_v(u) = TRUE for all children u of v & v receives all sent from parent then
        return TRUE
return FALSE
```

Algorithm 12: Upcast-Broadcast algorithm

Data: original graph G, source s, BFS tree τ

Result: Terminate when root s is done for both UPCAST and BROADCAST

UPCAST$_v \leftarrow IsUpCastDone()$

BROADCAST$_v \leftarrow IsBroadCastDone()$

switch v **do**

  case s **do**

    if UPCAST$_v$ = TRUE **then**

      if FROM-PARENT-QUEUE$_v = \emptyset$ ***and*** v *hasn't sent* ALLSENT *message before* **then**

        v sends ALLSENT messages to all of its children

    if BROADCAST$_v$ = TRUE **then**

      return

    else if FROM-PARENT-QUEUE$_v \neq \emptyset$ **then**

      Dequeue MSG$_u$ from FROM-PARENT-QUEUE$_v$ v sends MSG$_u$ to all of its children with type $BroadcastDistance$

    if ALLSENT$_{\text{PARENT}}$ = TRUE ***and*** v *hasn't sent* ALLSENT *message before* **then**

      v sends ALLSENT messages to all of its children

  case $v \neq s$ ***and*** v *is not a leaf node* **do**

    if UPCAST$_v$ = TRUE ***and*** v *hasn't sent* $DoneUpcast$ *message before* **then**

      v sends $DoneUpcast$ message to its parent

    else if FROM-CHILDREN-QUEUE$_v \neq \emptyset$ **then**

      Dequeue MSG$_u$ from FROM-CHILDREN-QUEUE$_v$ v sends MSG$_u$ to its parent with type $UpcastDistance$

    if BROADCAST$_v$ = TRUE ***and*** v *hasn't sent* $DoneBroadcast$ *message before* **then**

      v sends $DoneBroadcast$ message to its parent

    else if FROM-PARENT-QUEUE$_v \neq \emptyset$ **then**

      Dequeue MSG$_u$ from FROM-PARENT-QUEUE$_v$ v sends MSG$_u$ to all of its children with type $BroadcastDistance$

    if ALLSENT$_{\text{PARENT}}$ = TRUE ***and*** v *hasn't sent* ALLSENT *message before* **then**

      v sends $AllSent$ messages to all of its children

  case v *is a leaf node* **do**

    if UPCAST$_v$ = TRUE ***and*** v *hasn't sent* $DoneUpcast$ *message before* **then**

      v sends $DoneUpcast$ message to its parent

    else if FROM-CHILDREN-QUEUE$_v \neq \emptyset$ **then**

      Dequeue MSG$_u$ from FROM-CHILDREN-QUEUE$_v$ v sends MSG$_u$ to its parent with type $UpcastDistance$

    if BROADCAST$_v$ = TRUE ***and*** v *hasn't sent* $DoneBroadcast$ *message before* **then**

      v sends $DoneBroadcast$ message to its parent

`/* Received messages from v's neighbors */`

$ProcessMessagesUpcastBroadCast()$

Algorithm 13: Process messages for Upcast-Broadcast

Data: Node v

Result: Node v receives and update all MSG_u from its neighbors

```
for each MSG_u received by v from one (say u) of its neighbors do
    if u is one of v's children then
        if MSG_u is DoneUpcast then
            UPCAST_v(u) ← TRUE
        if MSG_u is DoneBroadcast then
            BROADCAST_v(u) ← True
        if MSG_u is UpcastDistance then
            Enqueue MSG_u in FROM-CHILDREN-QUEUE_v
            if v = s then
                Enqueue MSG_u in FROM-PARENT-QUEUE_v
    else if u is parent of v then
        if MSG_u is AllSent then
            ALLSENT_PARENT ← TRUE
        if MSG_u is BroadcastDistance then
            Enqueue MSG_u in FROM-PARENT-QUEUE_v
            if v ∈ V^VIRT then
                if v ∈ S_G^(h)[k](u) then
                    S_G^(h)[k](v) ← S_G^(h)[k](v) ∪ MSG_u
```

3.6.3 Elkin-R algorithm: Probabilistic virtual node selection

In this variant of Elkin's algorithm, denoted by the Elkin-R algorithm, we explore the impact of varying the probability value used to determine the number of virtual nodes. Suggestion of Elkin [14] is to randomly select q nodes with the value of q as indicated in are calculated Table 3.5.

However, our research investigates the potential for optimizing computational time by employing alternative probability distributions. To this end, we randomly select the set of virtual nodes from the complete set of nodes, using our modified probability distribution. Our experimental results demonstrate a significant improvement in the algorithm's performance compared to the probabilities proposed in the original paper by Elkin. The observed improvements suggest that there may be scope for further optimization of the virtual node selection process in distributed shortest-path algorithms.

3.6.4 Elkin-D algorithm: Distance-based virtual node selection

In this variant of the Elkin algorithm, denoted by the Elkin-D algorithm, we propose to Select the virtual nodes so that they are pairwise at least n_{HOPS} hops away. The details of the algorithm can be viewed in Algorithm 14.

Algorithm 14: Elkin-D: Construct V^{VIRT} (at least n_{HOPS} hops away between each node v')

Data: Graph $G = (V, E)$, min hop distance n_{HOPS}
Result: Set of virtual nodes V^{VIRT}

$V^{\text{VIRT}} \leftarrow \emptyset$
$\tilde{V} \leftarrow V$
while $\tilde{V} \neq \emptyset$ **do**
    Select random $v \in \tilde{V}$
    $V^{\text{VIRT}} \leftarrow V^{\text{VIRT}} \cup \{v\}$
    $\tilde{V} \leftarrow \tilde{V} \setminus \{v\}$
    $Q \leftarrow \{v\}$
    $visited \leftarrow \{v\}$
    for $i \leftarrow 0$ **to** $n_{\text{HOPS}} - 1$ **do**
        if $Q = \emptyset$ **then**
            break
        $size \leftarrow |Q|$
        for $j \leftarrow 0$ **to** $size - 1$ **do**
            $w \leftarrow \text{Dequeue}(Q)$
            foreach $u \in$ *neighbors of* w **do**
                if $u \notin visited$ **then**
                    $Q \leftarrow Q \cup \{u\}$
                    $visited \leftarrow visited \cup \{u\}$
                    $\tilde{V} \leftarrow \tilde{V} \setminus \{u\}$
return V^{VIRT}

Chapter 4

Experimental Results

In this chapter, we describe our implementations and experimental results for the algorithms described in Chapter 3. We conducted extensive experiments on the different algorithms discussed in the previous chapters: Elkin, Distributed Bellman-Ford, Elkin-R, and Elkin-D, on random graphs of different densities, grids of various sizes, the CONUS telecommunication network [41], and the graphs from the Graph Neural Networking Challenge [39].

4.1 Experimental setup

4.1.1 Multi-threaded implementation of distributed algorithms

Since we wanted to experiment on large graphs, conducting experiments on real-world networks was not feasible. To simulate synchronous distributed algorithms, we created a multi-threaded programming environment in Java [32].

Our simulation framework introduces a direct mapping where each node in the distributed algorithm is instantiated as a distinct Java `Thread`. The computational tasks and message processing logic pertaining to a node are executed within the context of its assigned thread. To emulate the progression of synchronous rounds, a crucial synchronization mechanism is implemented. Each node thread utilizes an *executed round* variable to track its progress. This mechanism enforces synchronization by ensuring that a node thread only proceeds to computations and message handling for round $k+1$ after all participating threads have signaled completion of round k. Such coordination guarantees that operations within a specific round (e.g., message exchanges, state updates based on received messages) are globally finalized across all simulated nodes before the next round starts. The total duration of the simulation, in terms of the number of rounds executed by

the distributed algorithm, is governed by a global *number of rounds* parameter. This parameter is configured externally in the main program before initiating the multi-threaded simulation run.

This implementation, while enabling the simulation of synchronous rounds, presents limitations regarding the measurement of computational time. We measure the total execution time by recording the *wall-clock time* from the moment before the first round begins until the algorithm completes its final round. However, this single duration does not accurately reflect the execution time that would be observed in a genuinely distributed system running on physically separate machines. Some factors contribute to this discrepancy:

(1) **Shared Computing Resources:** In our simulation, multiple threads (representing nodes) *compete* for processing time on a limited number of CPU cores available on the single machine running the simulation. The operating system's thread scheduler constantly switches between these threads (*time-slicing*). This introduces scheduling overhead and resource contention that is fundamentally different from a scenario where each node runs on its own dedicated hardware. The parallelism achieved is *concurrent execution* on shared resources, not *true parallel processing* across independent nodes.

(2) **Synchronization Overhead:** The mechanism used to enforce synchronous rounds (involving condition variables coordinating the *executed round* variable across threads) introduces *synchronization overhead* specific to the Java threading model and the host operating system. While real distributed systems also incur synchronization costs, the nature and magnitude of these costs (often involving message exchanges and timeouts) are different.

(3) **Performance Insight:** Measuring the total *wall-clock time* captures everything: actual computation within threads, time spent waiting for the scheduler, time waiting for synchronization barriers, JVM overhead (like garbage collection), and the fast inter-thread communication. It does not isolate the *compute time* per node in a way that maps directly to a real system's performance profile.

Therefore, while the measured total time provides a useful metric for comparing the *relative performance* of different algorithm variations or parameter settings *within this specific simulation environment*, and it reflects the simulated round counts, it should not be interpreted as a precise prediction of the *absolute execution time* on a real distributed network. It remains, however, the most feasible performance indicator we can obtain given the constraints of simulating large-scale synchronous algorithms without access to a dedicated distributed testing system.

4.1.2 Hardware

Simulations were performed on a high-performance computing (HPC) cluster managed by the Slurm workload manager [36] (Node Version 24.05.4). Each simulation program was allocated resources corresponding to 5 CPU cores and permitted a maximum memory usage of 100 GB of RAM. The underlying hardware of the nodes executing these jobs consists of dual-socket servers with hyperthreading enabled. The cluster nodes run Ubuntu Linux [40] (OS=Linux 5.15.0-138-generic) and are equipped with 1 TB of physical RAM each.

4.1.3 Input graphs

Our experimental input graphs were generated and configured as follows. For Erdős-Rényi random graphs, we utilized the NetworkX library [21] to produce 100 distinct graphs, each having 1000 nodes. Edge weights were integers chosen uniformly at random from the range [1, 1000]. To ensure graph connectivity, a prerequisite for our algorithms, graphs were repeatedly generated until this connectivity requirement was met. Across these 100 random graphs, which varied in edge probability p to represent sparse, medium-density, and dense scenarios, node 0 was consistently designated as the source.

For square grid graphs, we also used NetworkX to generate 100 graphs. Similar to the Erdős-Rényi graphs, if weights were applied, they were integers chosen randomly from [1, 1000]. A different source node was randomly selected for each of the 100 generated grid graphs.

In contrast, for the *CONUS* network and datasets from the Graph Neural Networking Challenge [39], we utilized existing graph data given edge weights; the source node for experiments on these graphs was chosen randomly in each experimental run.

4.1.4 Performance metrics

We use three performance metrics to measure the performance of our algorithms: the total number of rounds, the total number of messages, and the computational (clock) time. As already mentioned, the clock time is not a true representation of the time taken by a distributed algorithm, but it gives some idea of the constants involved in the implementation of the algorithm. All results are averaged over 100 graph runs. To confirm their accuracy, we validate these averaged results against the Floyd-Warshall algorithm. To assess the consistency of our results, we also calculate the standard deviation based on the performance observed for each individual graph.

4.2 Performance of non-pipelined Elkin and pipelined Elkin

In this section, we examine the benefits of using the pipelined *Upcast-Broadcast* approach, described in Section 3.6.2. Table 4.1 shows the results on the CONUS network, and shows 18.5% savings for the optimized version of Elkin compared to the original version. We also ran experiments on randomly generated sparse graphs, medium-density graphs, and dense graphs, as well as square grids of different sizes. Figure 4.1 shows the results (averages over 100 graphs are reported), and show that using the pipelined *Upcast-Broadcast* results in savings of 20%, 8.3%, and 13.1%, respectively, in sparse, medium density, and dense in terms of the number of rounds. Figure 4.1-(d) shows the results on the square grid graphs, and shows 35.7% savings for the number of rounds. These percentage improvements are summarized in Table 4.1.

Therefore, in the subsequent sections, we only perform experiments with the optimized version of Elkin's algorithm.

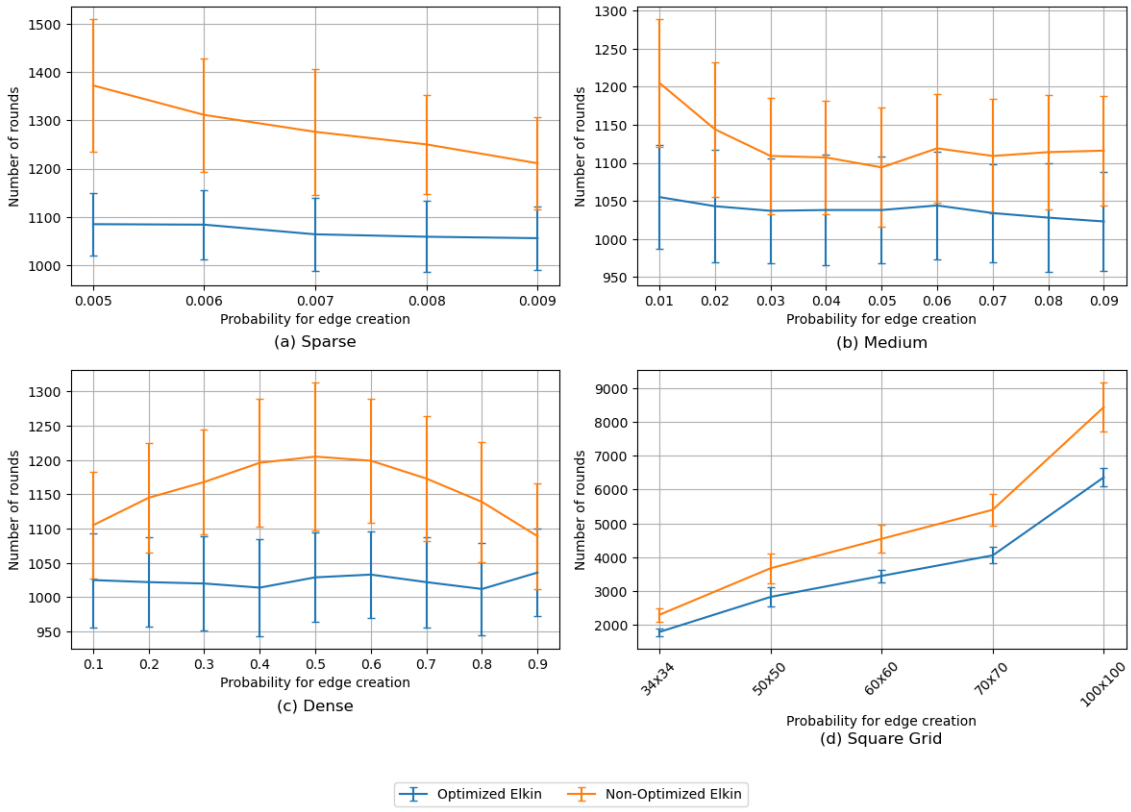


Figure 4.1: Number of rounds between optimized and non-optimized Elkin with computed q

Table 4.1: Percentage improvement in the number of rounds.

Graph type	**Number of Nodes**	**% Improvement in # Rounds**
CONUS	75	19.8%
GNN 50	50	24.9%
GNN 75	75	26.3%
GNN 100	100	25.0%
GNN 130	130	23.7%
GNN 170	170	23.7%
GNN 200	200	22.9%
GNN 240	240	35.6%
GNN 260	260	26.9%
GNN 280	280	21.8%
GNN 300	300	21.7%
Sparse-random	1000	20.0%
Medium-density-random	1000	8.3%
Dense-random	1000	13.1%
Square grid 34×34	1056	28.4%
Square grid 50×50	2500	30.0%
Square grid 60×60	3600	31.8%
Square grid 70×70	4900	33.3%
Square grid 100×100	10000	32.6%

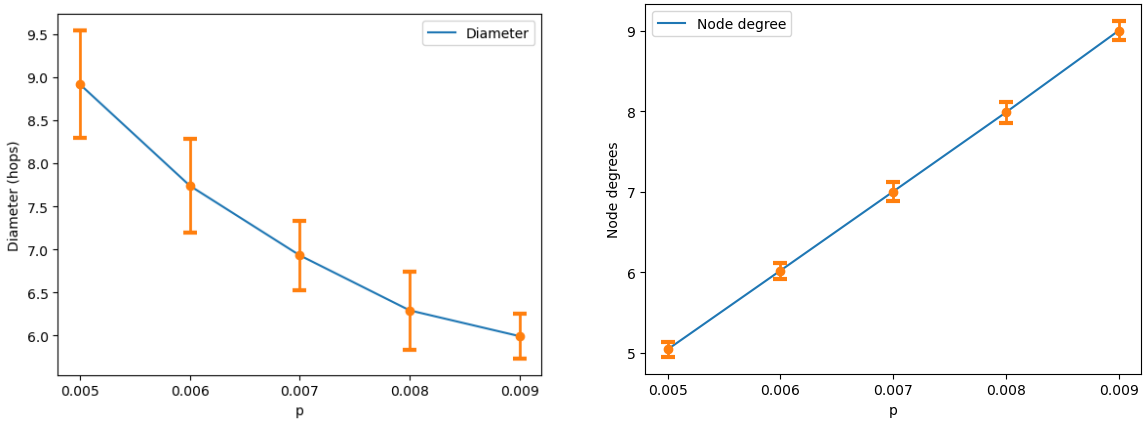
4.3 Experiments on Erdős-Rényi graphs

We conducted our experiments on Erdős-Rényi random graphs using NetworkX [21], generating 100 distinct connected graphs, each containing 1000 nodes. To ensure graph connectivity, we repeatedly generated graphs until each met the connectivity criterion. The edge creation probability varies across different graph densities: for sparse graphs, it ranges from 0.005 to 0.009; for medium graphs, the range is 0.01 to 0.09; and for dense graphs, the probability spans from 0.1 to 0.9. In the following subsections, we describe the results obtained on sparse, medium, and dense graphs separately. In each case, we first describe the graph characteristics (degree and diameter), the performance of Elkin-R for different values of q^*, the performance of Elkin-D for different values of d, and finally a comparison of the best versions of Elkin-R and Elkin-D as determined empirically, with the original Elkin algorithm and Bellman-Ford.

4.3.1 Sparse graphs

4.3.1.1 Graph characteristics

In this section, we present a brief analysis of the random graphs on which experiments were conducted, for edge creation probability in the range 0.005 to 0.009. In particular, we plot the average degree and hop diameter for these graphs.



(a) Diameter

(b) Node degree

Figure 4.2: Comparison of hop diameter and node degree for sparse graphs.

Figure 4.2a illustrates the relationship between edge creation probability (p) and network hop-diameter (D^{H}) in sparse graphs, based on an average of 100 generated graphs. The x-axis shows edge creation probabilities ranging from 0.005 to 0.009, while the y-axis represents the diameter measured in hops. As the edge creation probability increases, there is a clear downward trend in the average hop diameter values, decreasing from approximately 9 hops at $p = 0.005$ to about 6 hops at $p = 0.009$. This is to be expected, as the graph becomes denser as p increases, leading to the hop diameter of the graph becoming smaller.

Figure 4.2b shows the linear relationship between edge creation probability and average node degree across sparse graphs. The expected degree of nodes is np which is confirmed by our experiments.

4.3.1.2 Performance of Elkin-R

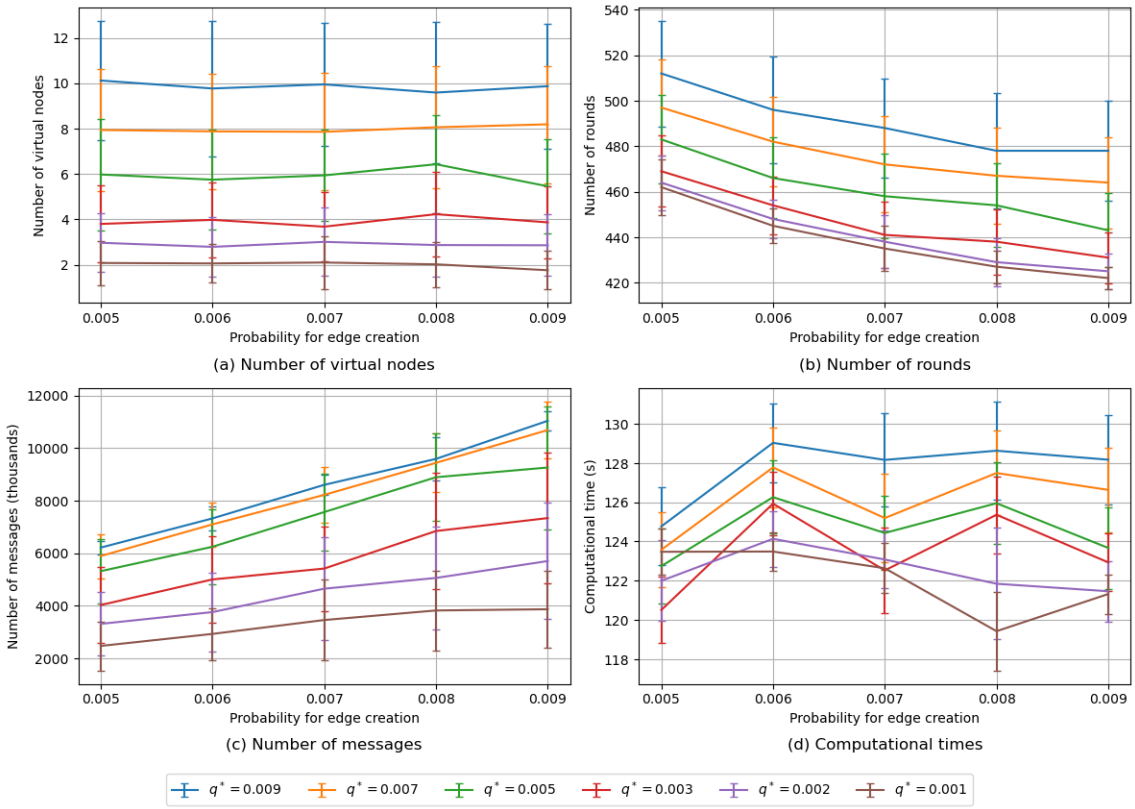


Figure 4.3: Elkin-R performance for sparse graphs

Recall that Elkin-R uses a fixed (empirically derived) probability q for each node to be a virtual node. We experimented with $q^* = 0.001, 0.002, 0.003, 0.005, 0.007, 0.009$. Figure 4.3 plots the number of virtual nodes, the number of rounds, the number of messages, and the computational time for Elkin-R for each of these values of q^* for sparse graphs created with different edge creation probabilities.

Number of virtual nodes

As shown in Figure 4.3-(a), the number of virtual nodes increases with q^* and is independent of p. Since s is always chosen as a virtual node, and all other nodes are chosen with probability q^*. The expected number of virtual nodes is $(n-1)q^* + 1 = 999q^* + 1$, which is confirmed by our experiments.

Number of rounds

Figure 4.3-(b) compares the number of rounds required for different virtual node selection probabilities (q^*) in sparse graphs.

First, we consider the effect of p for a fixed value of q^*. Observe that as p increases, the number of edges E increases, and the hop diameter of the graph decreases, as does the depth of the BFS tree built by

the algorithm. From Table 3.6, it can be seen that the number of edges does not play a role in the number of rounds, and the only part of the algorithm that is affected by the depth of the tree is the *Upcast-Broadcast*. This is also confirmed by our experimental results. The BFS tree construction is affected by the hop diameter, but needs very few rounds in comparison. To summarize, the number of rounds in the *Upcast-Broadcast* algorithm, and therefore the entire algorithm, decreases with increasing p.

Next, we consider the effect of q^* for a fixed value of p. For example, for $p = 0.005$, the plot shows that lower q^* values (0.001 to 0.003) achieve the most efficient performance, requiring approximately 460-470 rounds. In contrast, higher q^* values result in an increased number of rounds, with $q^* = 0.007$ needing 498 rounds and $q^* = 0.009$ requiring 513 rounds. Our analysis (see Figure 4.4-(b)) shows that this increase in the number of rounds is almost entirely in the *Upcast-Broadcast* process.

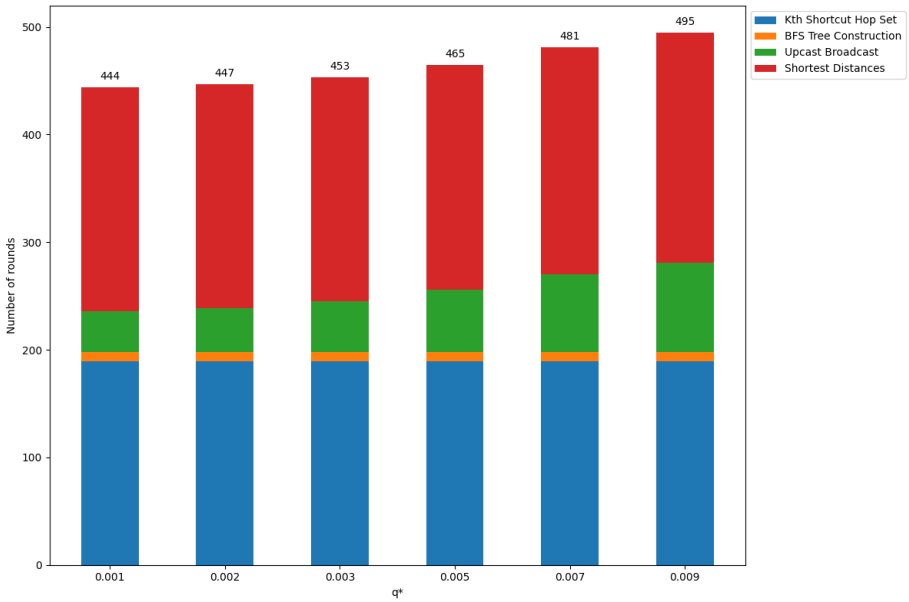


Figure 4.4: Number of rounds across different q^* values for p = 0.006.

Table 4.2 shows the number of rounds for the *Upcast-Broadcast* process in our experiments. Theoretically, the number of rounds needed for the *Upcast-Broadcast* algorithm, which is described in Algorithm 12 can be described as follows:

$$\text{number of rounds} = O\big(Depth(\tau)\big) + (|V^{\text{VIRT}}| - 1) \cdot \min(|V^{\text{VIRT}}|, k) + c \tag{1}$$

As mentioned earlier, the number of virtual nodes increases with increasing q^*, while $Depth(\tau)$ does not depend on q^*. However, the constant in the $O(Depth(\tau))$ depends on the exact location of the virtual nodes in the BFS tree. Also the number $\min(|V^{\text{VIRT}}|, k)$ term is not a linear function of the number of virtual

nodes. We found empirically that Equation (2) is a good approximation of the actual number of rounds in the *Upcast-Broadcast*, as shown in Table 4.2.

$$\text{number of rounds} = (k-1)\cdot Depth(\tau) + (|V^{\text{VIRT}}|-1)\cdot \min\{|V^{\text{VIRT}}|, k\} + \begin{cases} 5, & \text{if } |V^{\text{VIRT}}| < k \\ 1, & \text{if } |V^{\text{VIRT}}| \approx k \\ 0, & \text{if } |V^{\text{VIRT}}| > k \end{cases} \quad (2)$$

Table 4.2: The number of rounds for *Upcast-Broadcast* part from experiment and Equation (2) with $p = 0.007$

q^*	$\lvert V^{\text{VIRT}}\rvert$	**Eq. (2)**	**Real rounds**	**% difference**
0.001	1.8	34.4	35	1.71%
0.002	2.98	38.9	39	0.25%
0.003	4.05	44.9	44	2.05%
0.005	5.56	54.3	54	0.55%
0.007	7.63	67.7	67	1.04%
0.009	9.87	81.2	81	0.24%

Number of messages

Figure 4.3-(c) shows the number of messages exchanged for different virtual node selection probabilities (q^*) in sparse graphs. Higher q^* values (0.005 to 0.009) result in more messages exchanged, requiring between 5.3-6.1 million messages at $p = 0.005$ and increasing to 9.8-10.9 million messages at $p = 0.009$.

We see that for every q^*, the number of messages increases with increasing edge creation probability p. From Table 3.6, we see that the total number of messages increases with the total number of edges. As p increases, the number of edges E increases. Put another way, each node of the graph has more neighbors and exchanges more messages.

We shift our attention to the effect of q^* for a fixed p. Our analysis reveals that the process of constructing the k-shortcut hopset structure is responsible for the majority of the message exchanges within the algorithm. Recall that the number of messages needed for constructing the k-shortcut hopset algorithm, which is described in Algorithm 2, is Equation (3).

$$\text{number of messages} = \sqrt{n}\cdot k \cdot \min(|V^{\text{VIRT}}|, k)\cdot |E| \quad (3)$$

As mentioned earlier, $\min(|V^{\text{VIRT}}|, k)$ is a linear function of $|V^{\text{VIRT}}|$ only up to the value $k = 6$. For

higher values, it stays constant. This is seen in the results shown in Table 4.3, which shows that the number of rounds increases with increasing q^* until the number of virtual nodes is higher than $k = 6$, and stays relatively constant after that. This explains why the curves for $q^* = 0.001, 0.002, 0.003$ are further apart than those for higher values of q^* in Figure 4.3.

Table 4.3: Number of messages when constructing k-shortcut hopset for $p = 0.005$

q^*	$\lvert V^{\mathbf{VIRT}}\rvert$	**Eq. (3)**	**Real number of messages**	**% difference**
0.001	2.08	1,990,460.2	1,972,484	0.90%
0.002	2.97	2,842,147.5	2,806,554	1.25%
0.003	3.8	3,636,417.7	3,515,848	3.31%
0.005	5.56	5,320,653.2	4,801,365	9.75%
0.007	7.63	5,741,712.1	5,347,768	6.86%
0.009	9.87	5,741,712.1	5,651,251	1.57%
0.082	82.77	5,741,712.1	5,689,576	0.9%

This analysis demonstrates that lower virtual node probabilities can reduce the absolute number of messages required.

Computational time

Figure 4.3-(d) compares computational times for different q^* values across various edge creation probabilities (p) in sparse graphs. While noting again that the computational time plotted here for the multi-threaded simulation of a distributed algorithm is not a correct representation of the time taken by a truly distributed and synchronized algorithm, we observe that the computational time depends on the number of rounds, the computational time taken per round, as well as the number of messages sent. We see that the time taken for a fixed q^* is relatively stable across different values of p.

Overall, our experiments demonstrate that selecting lower virtual node probabilities (q^*) leads to better computational efficiency. To be more specific, the results show that lower q^* values (0.001 to 0.002) achieve the most efficient computational times, around 120-122 seconds. In contrast, higher q^* values (0.005 to 0.009) result in longer computational times, requiring approximately 123-128 seconds. As seen in Figure 4.4, the increased time is largely due to the *Upcast-Broadcast* algorithm. As mentioned earlier, as q^* increases, the number of virtual nodes selected is higher, making the *Upcast-Broadcast* part (Algorithm 12) process take a longer time.

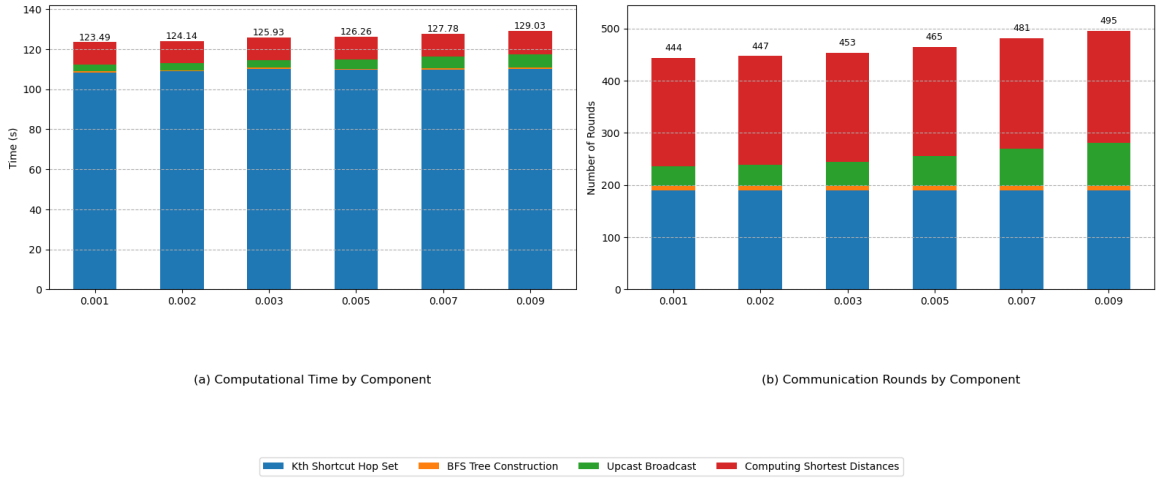


Figure 4.5: Computational times and number of rounds with different q^* values for p = 0.006.

Comparing Figure 4.4-(a) and Figure 4.4-(b), we observe that although the k-th shortcut hopset construction accounts for a large proportion of the total computational time, it does not account for the majority of the total number of rounds. This indicates that the average computational time per round is higher during the k-th shortcut hopset construction process compared to other parts of the algorithm. It is therefore possible that a tighter implementation of the algorithm for k-short cut hopset would result in improving the time taken by the algorithm.

4.3.1.3 Performance of ELKIN-D

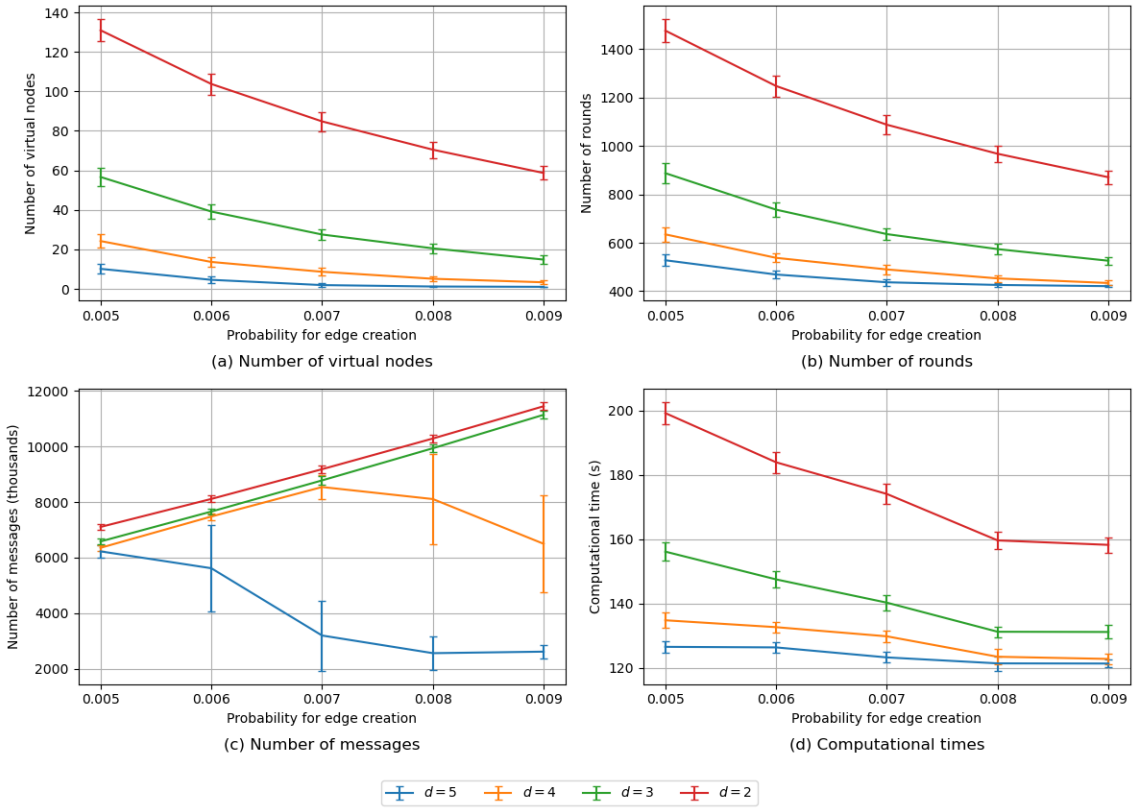


Figure 4.6: Elkin-D performance for sparse graphs

In this section, we investigate the performance of Elkin-D. Recall that in this algorithm, we choose virtual nodes that are at a distance of at least d hops apart. We study the effect of the parameter d, ranging from 2 to 5 hops, on the performance of the algorithm. Comparing Figures 4.3 and 4.6, we see that their behavior is different for all parameters studied.

Number of virtual nodes

Figure 4.6-(a) compares the number of virtual nodes across different hop counts for sparse graphs. Unlike in Elkin-R, we see that the number of virtual nodes decreases with increasing p for every value of d. This can be explained by the virtual node selection algorithm in Algorithm 14; since the graph becomes denser when p increases, the distance between nodes decreases, which leads to fewer virtual nodes being chosen when they have to be d hops apart, for any fixed d. Next, for any fixed p, as d increases, the number of virtual nodes chosen decreases, again since virtual nodes are required to be d hops apart in Elkin-D.

Number of rounds

Figure 4.6-(b) compares the number of rounds required across different values of d for Elkin-D in sparse

graphs. First, we note that with increasing p and a fixed d, the number of rounds decreases. As with Elkin-R, this is partly because as p increases, $Depth(\tau)$ decreases, making the number of rounds for *Upcast-Broadcast* decrease. But unlike Elkin-R, we also have the fact that as p increases, the number of virtual nodes decreases, which also leads to a reduced number of rounds in *Upcast-Broadcast*, as well as in *Computing-Shortest-Distance*, as seen in Figure 4.7.

Examining the effect of different d on a specific value of p, we find that $d = 4$ and $d = 5$ yield the most efficient performance, requiring approximately 528–635 rounds at $p = 0.006$. In contrast, lower values require a higher round: $d = 3$ costs 888 rounds, and $d = 2$ requires 1477 rounds. As illustrated in Figure 4.7-(b), this difference primarily comes from the *Upcast-Broadcast* and computing shortest distances phase. Those parts, which also include the *Upcast-Broadcast* process, decrease because the number of rounds is a function of the $|V^{\text{VIRT}}|$. Our empirical results, shown in Table 4.4, validate Equation (2) as a suitable approximation for the number of rounds in the *Upcast-Broadcast* phase.

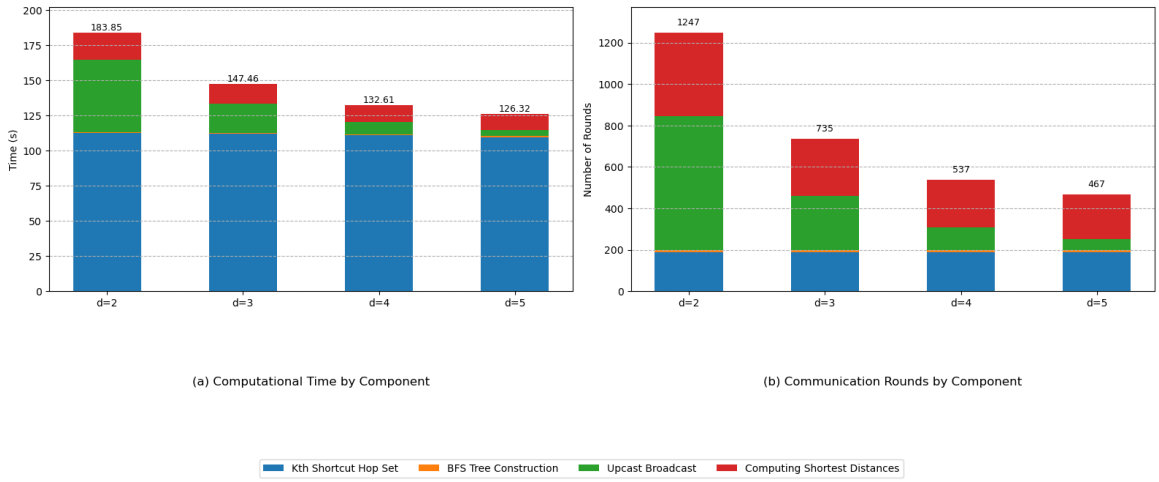


Figure 4.7: Computational times and number of rounds with different d values for p = 0.006.

Table 4.4: The number of rounds for *Upcast-Broadcast* part from experiment and equation with $p = 0.006$

d	$\lvert V^{\text{VIRT}} \rvert$	**Eq.(2)**	**Real rounds**	**% difference**
2	103.77	647.57	648	0.06%
3	39.15	259.85	262	0.82%
4	13.62	106.67	111	3.9%
5	4.56	52.1836	55	5.12%

Number of messages

Table 4.5: Number of messages when constructing k-shortcut hopset for $p = 0.005$

d	$\lvert V^{\mathbf{VIRT}}\rvert$	**Eq. (3)**	**Real number of messages**	**% difference**
2	130.99	5,741,712.13	5,689,620	0.91%
3	56.67	5,741,712.13	5,689,325	0.91%
4	23.5	5,741,712.13	5,689,217	0.91%
5	10.33	5,741,712.13	5,660,283	1.42%

Table 4.6: Number of messages when constructing k-shortcut hopset for $p = 0.009$

d	$\lvert V^{\mathbf{VIRT}}\rvert$	**Eq. (3)**	**Real number of messages**	**% difference**
2	58.68	10,249,422.6	10,156,006	0.91%
3	14.81	10,249,422.6	10,155,694	0.91%
4	3.31	5,654,264.8	5,596,901	1.01%
5	1.02	1,742,401.8	1,726,751	0.90%

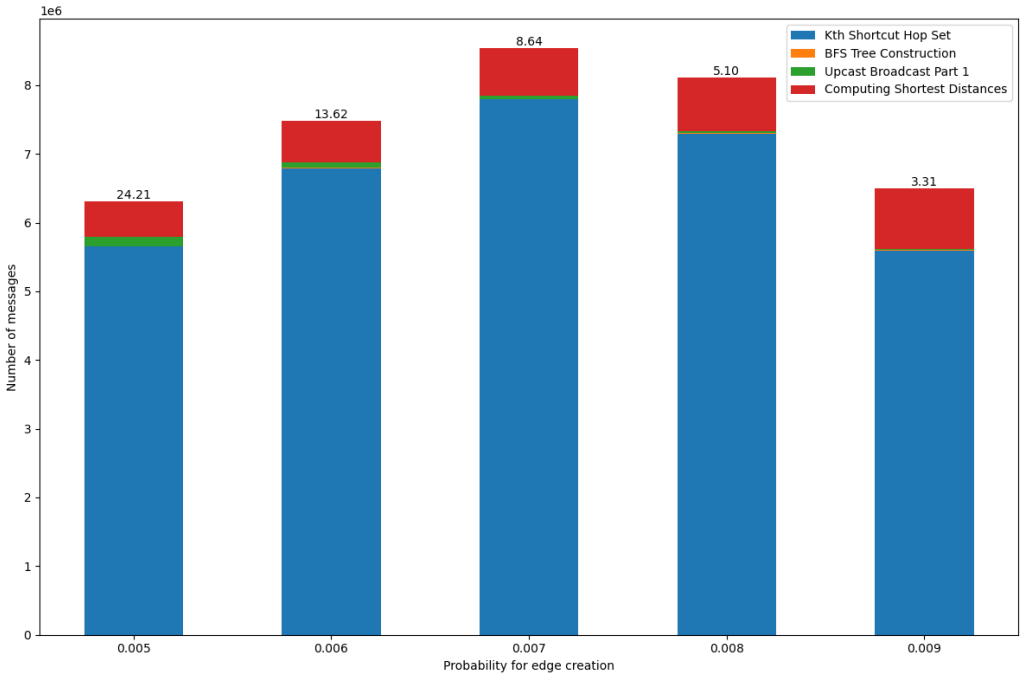


Figure 4.8: Number of messages in k-shortcut hopset for Elkin-D with $d = 4$

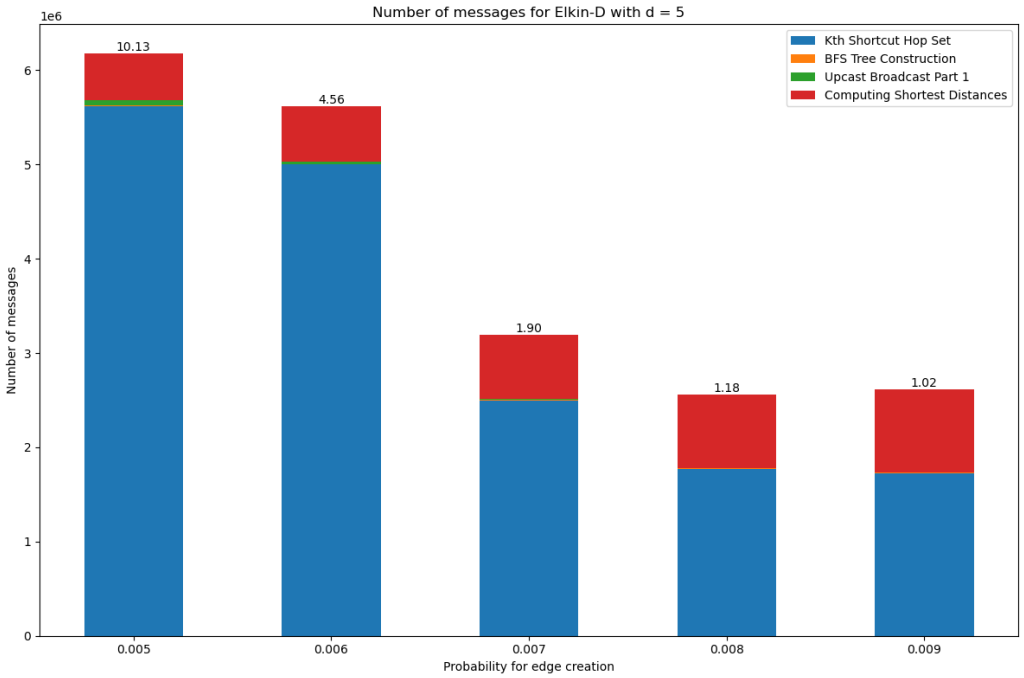


Figure 4.9: Number of messages in k-shortcut hopset for Elkin-D with $d = 5$

Table 4.7: Comparison of formula vs. experimental message counts in k-th shortcut hopset for Elkin-D with $d = 4$

p	**Number of virtual nodes**	**Eq. (3)**	**Experiments**	**% Difference**
0.005	23.5	5,747,882	5,695,331	1%
0.006	14.4	6,834,314	6,761,273	1%
0.007	7.857	7,958,318	7,683,427	3.5%
0.008	4.857	7,359,810	7,294,903	0.9%
0.009	3.857	6,599,259	6,515,720	1.2%

Table 4.8: Comparison of formula vs. experimental message counts in k-th shortcut hopset for Elkin-D with $d = 5$

p	**Number of virtual nodes**	**Eq. (3)**	**Experiments**	**% Difference**
0.005	10.33	5,747,882.4	5,695,294	1%
0.006	5.14	5,854,729.1	5,475,110	6.5%
0.007	1.14	1,512,080.5	1,496,433	1%
0.008	1.14	1,727,441.5	1,718,490	0.5%
0.009	1	1,710,982.6	1,691,363	1.2%

Figure 4.6-(c) compares the number of messages (in millions) exchanged across different maximum numbers of hop distances (2-5 hops) in sparse graphs, averaged over 100 graphs. The results show dramatically different patterns depending on hop distance.

We first examine the effect of d for a fixed p. According to our analysis (see, for example, Figure 4.8, the k-shortcut hopset construction phase is the most message-intensive part of the overall algorithm. As described in Table 3.6, the number of messages required specifically for this construction is determined by Equation (3). At $p = 0.005$, since the number of virtual nodes when using 5, 4, 3, and 2 hops are 10.33, 23.5, 56.67, 130.99 respectively which are both greater than $k = 6$, so the number of messages to be exchanged when constructing the k-th shortcut hopset are approximately the same for all cases of d at almost 5.7 million messages as demonstrated in Table 4.5. At $p = 0.009$, since the number of virtual nodes when using 2 and 3 hops are 56.68 and 14.81 respectively which are both greater than $k = 6$, so the number of messages in the k-th shortcut hopset construction process are approximately the same at about 10.1 million messages as demonstrated in Table 4.6. In contrast, $d = 4$ and $d = 5$, the $|V^{\text{VIRT}}|$ is 3.31 and 1.02 respectively, which are smaller than k, according to Equation (3), it can explain the difference in number of messages between $d = 4$ and $d = 5$, which is described in Table 4.6.

We shift our attention to the effect of p for a fixed d, and focus our attention on the k-shortcut hopset construction part of the algorithm, since it sends the most number of messages. From Equation (3), we see that the number of messages sent is directly proportional to the number of edges in the graph. Therefore, as p increases, since the number of edges in the graph (E) increases, we expect that the number of messages should increase. Indeed, this is the case for $d = 2, 3$. We note that the number of virtual nodes for these two values of d is always greater than $k = 6$, and therefore does not play a role in the number of messages.

However, for $d = 4, 5$, the number of virtual nodes varies greatly for different values of p. When $|V^{\text{VIRT}}| > 6$, as it is for $p < 0.008$ for $d = 4$, it does not play a role in the number of messages, which increases with increasing p. But when $|V^{\text{VIRT}}| < 6$, as for all values of p for $d = 5$, and for $p \geq 0.008$ for $d = 4$, the number of messages is *inversely proportional* to p. So the number of messages starts decreasing. Indeed, Table 4.7 and Figure 4.8 validate Equation (3).

Computational time

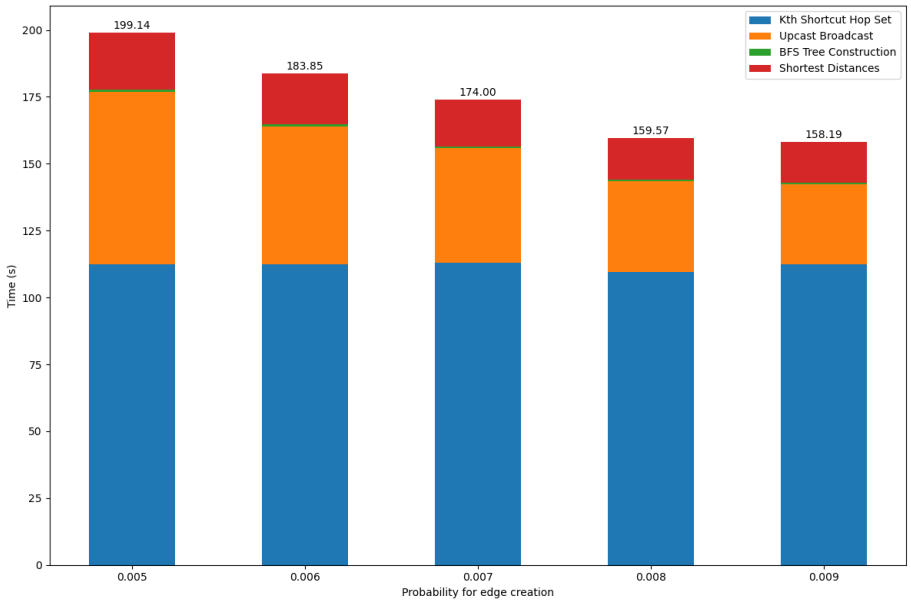


Figure 4.10: Computational times over edge create probabilities for Elkin-D with $d = 2$.

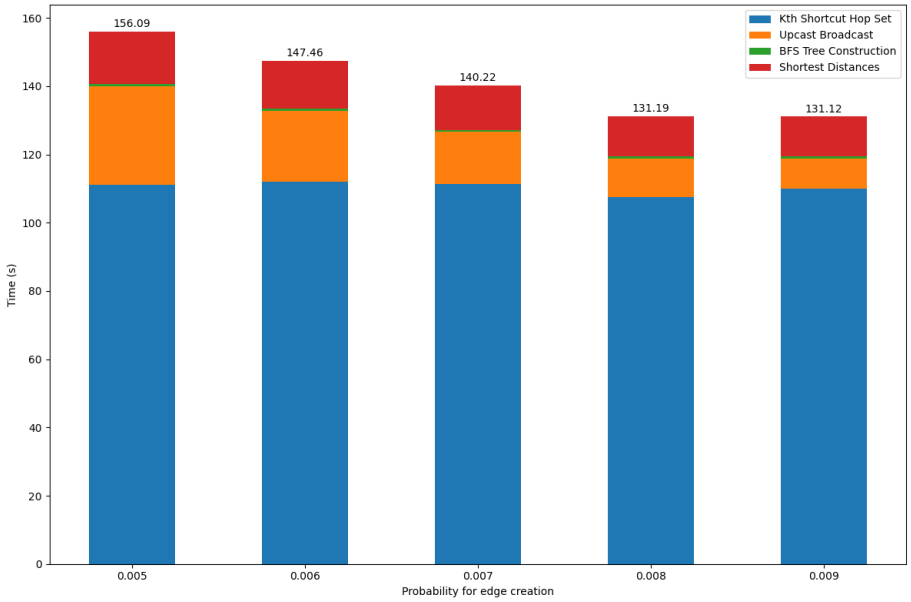


Figure 4.11: Computational times over edge create probabilities for Elkin-D with $d = 3$.

Figure 4.6-(d) compares computational times across different maximum hop distances (d) (2-5 hops) in sparse graphs. The results show that 5 hops achieves the most efficient performance, requiring approximately 126 seconds at $p = 0.005$ and maintaining notably stable performance across different p (0.005 to 0.009). Overall, for all d, all curves show a consistent downward trend as p increases, most pronounced for shorter hop distances (2-3 hops). To be more specific, while 4 hops requires around 122-134 seconds, 3 hops needs

approximately 131-156 seconds, and 2 hops demonstrates the longest computational time, ranging from about 200 seconds at $p = 0.005$ to 158 seconds at $p = 0.009$.

Next, we consider the effect of d for a fixed p. Our experiments demonstrate that selecting a higher d leads to better computational efficiency. To be more specific, the results show that higher d values (4 and 5) achieve the most efficient computational times, around 126 and 134 seconds at $p = 0.005$, respectively.

In contrast, lower d values (2 and 3) result in longer computational times, requiring approximately 156 and 199 seconds, respectively. This could be explained in Figure 4.6-(a) and Figure 4.5, with the lower number of d, the number of virtual nodes selected is higher, making the *Upcast-Broadcast* part (Algorithm 12) process take a longer time.

4.3.1.4 Comparison between Elkin, Elkin-R, Elkin-D, and Bellman-Ford

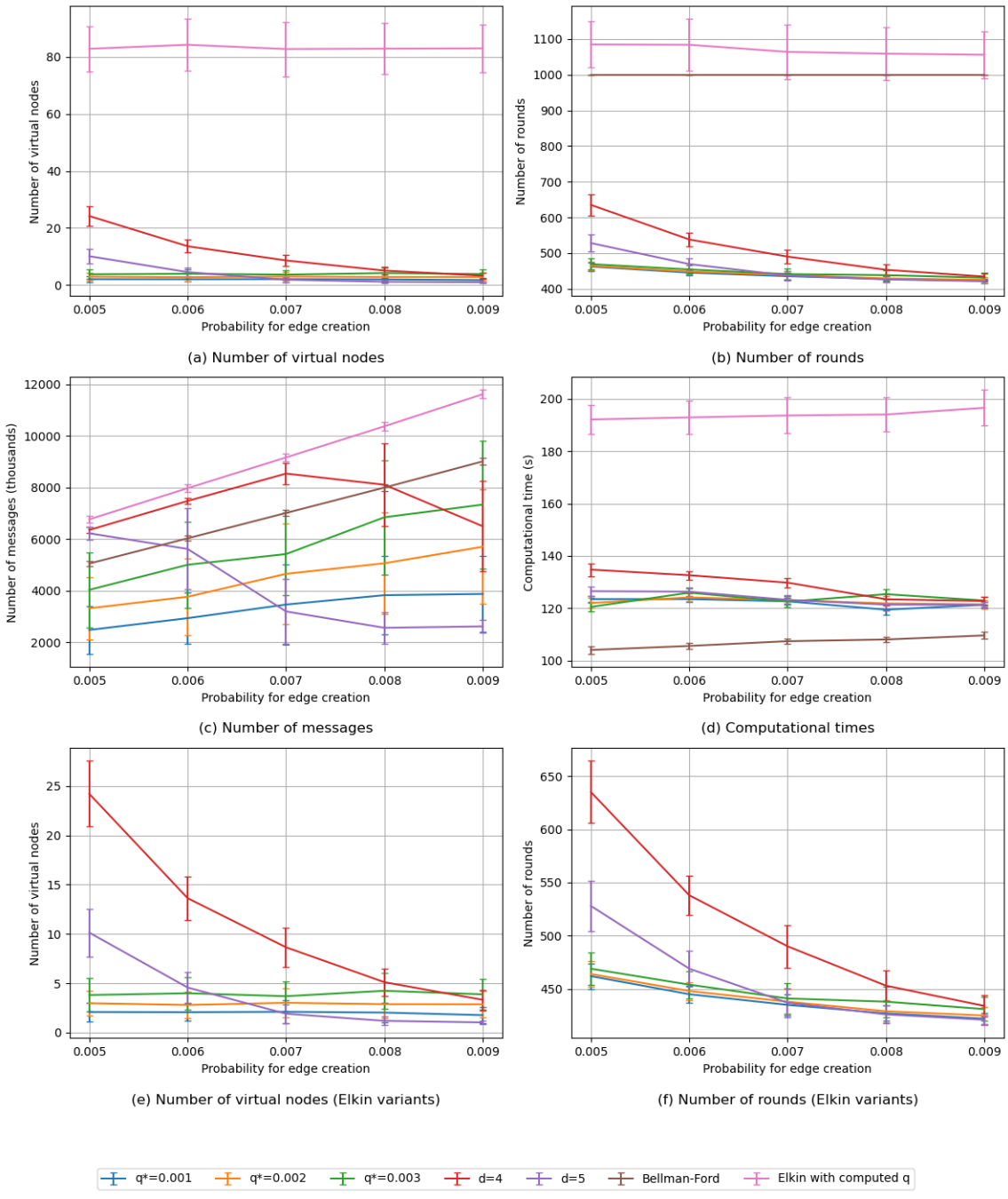


Figure 4.12: Comparison performance for sparse graphs

In this section, we compare the performance of our optimized Elkin, Bellman-Ford with the best versions of Elkin-D and Elkin-R as determined empirically in the previous sections. For Elkin-R, our experimental results indicate that $q^* = 0.001, 0.002, 0.003$ give the best results, while for Elkin-D, $d = 4, 5$ give the best

results. Figure 4.12-(a) shows the number of virtual nodes for all algorithms. Clearly, Bellman-Ford uses no virtual nodes, but all our variants use very few virtual niodes, especially as p increases to 0.009, while Elkin uses over 80 virtual nodes. Notably, Elkin-R with $q^* = 0.001$ shows the most stable and lowest number of virtual nodes across graph densities.

Figure 4.12-(b) illustrates the comparison of the number of rounds across different variations of the Elkin algorithm and the Bellman-Ford algorithm for sparse graphs. The Elkin variant with computed q shows consistently high round counts, maintaining approximately 1100 rounds across all edge creation probabilities. The Bellman-Ford algorithm demonstrates stable performance at 1000 rounds. The remaining variants operate at significantly lower round counts, between 450-630 rounds. Among these, Elkin-D with $d = 4$ shows the most notable trend, starting at around 630 rounds and decreasing to approximately 455 rounds as probability increases. The Elkin variant that demonstrates the most efficient number of rounds across different p values is Elkin-R with $q^* = 0.001$.

Figure 4.12-(c) compares the number of messages exchanged (in million) across different variations of the Elkin algorithm and the Bellman-Ford algorithm for sparse graphs. The Elkin variant with computed q consistently shows the highest number of messages, steadily increasing with edge creation probability, reaching nearly 12 million messages at $p = 0.009$. This high message exchange is directly linked to its larger computed q^*, due to the significantly more virtual nodes it selects (around 82, as shown in Figure 4.12-(a)). A similar pattern can be viewed in the Bellman-Ford algorithm. Its message count, starting at roughly 5 million at $p = 0.005$ and increasing to around 9 million at $p = 0.009$, is directly influenced by the graph's density. As the edge creation probability p rises, the graph becomes more connected, meaning each node has a greater number of adjacent nodes (neighbors). Consequently, in each iteration of the Bellman-Ford algorithm, every node needs to exchange distance information with this larger set of neighbors. This increased interaction per node, multiplied across all nodes and the fixed number of rounds (1000), leads to a substantial growth in the total number of messages passed as the graph density increases. The Elkin variants that we have proposed all have lower message counts, especially Elkin-R with $q^* = 0.001$ with $p = 0.005, 0.006$ and Elkin-D variant with $d = 5$ for higher values of p.

Figure 4.12-(d) compares the computational time of different Elkin variations and Bellman-Ford on sparse graphs. Elkin with computed q has the highest computational time, and Bellman-Ford has the lowest computational time, with the Elkin variants falling somewhere in between. It is interesting to note that Bellman-Ford exhibits lower computational time despite having a higher number of messages and a higher number of rounds. As mentioned earlier, the computational time is not a reliable indicator of performance in the distributed setting. Also, the computation in every round of Bellman-Ford is extremely simple, while in the Elkin algorithm and Elkin variants, the computation is heavier. However, more optimization here could

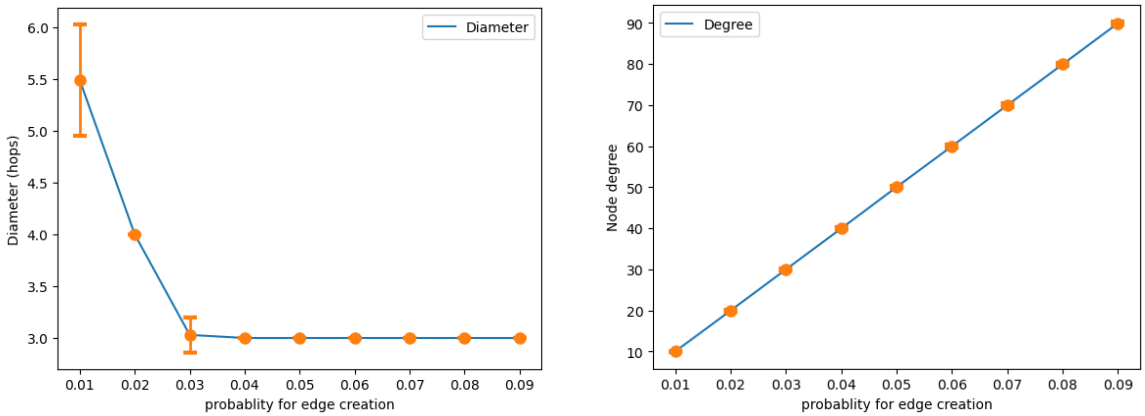
help decrease the computational time taken by Elkin variants.

4.3.2 Medium density graphs

In this section, we report on our experiments with graphs of medium density, that is, graphs where the edge creation probability lies between 0.01 and 0.09.

4.3.2.1 Graph characteristics

First, we describe the characteristics of the random graphs on which we have run our algorithms.



(a) Hop diameter

(b) Average node degree

Figure 4.13: Comparison of hop diameter and average node degree for medium density graphs.

As we can see from Figure 4.13a, for the hop diameter value, the average hop diameter reduces from 5.5 to 3 hops as the probability rises from 0.01 to 0.03. Beyond 0.03, the hop diameter stabilizes around 3 hops, showing little change with further probability increases up to 0.09.

For the node degree, Figure 4.13b shows a clear linear relationship between the probability of edge creation and the average node degree in a network. As the probability for edge creation increases from 0.01 to 0.09, the average node degree rises steadily from about 10 to 90, the value that matches the theoretical average degree predicted by the formula $deg_{avg} = p \cdot n$.

4.3.2.2 Performance of Elkin-R

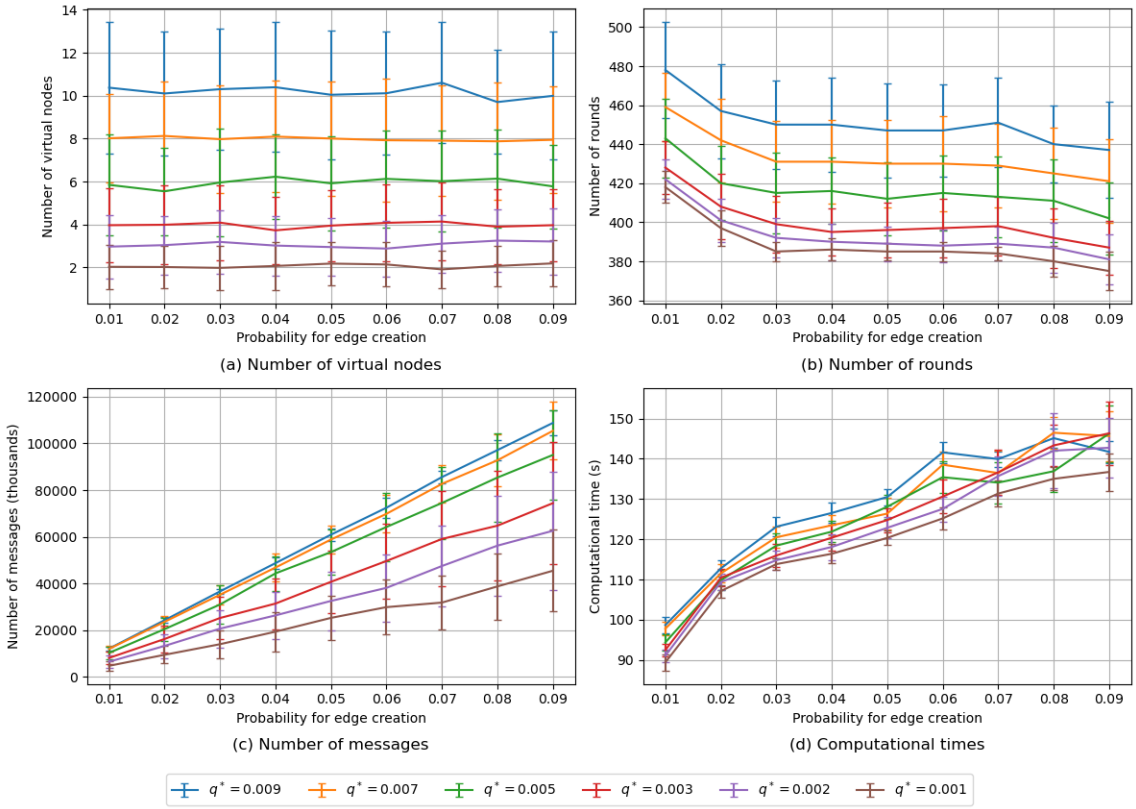


Figure 4.14: Elkin-R performance for medium-density graphs

Number of virtual nodes

As we can see from Figure 4.14-(a), and as explained in Section 4.3.1, a higher probability p of a node being chosen as a virtual node results in a higher number of virtual nodes being selected for any fixed q^*. The expected number of virtual nodes selected is $q^* \cdot (n-1) + 1$.

Number of rounds

Figure 4.14 compares the number of rounds required for different virtual node selection probabilities (q^*) in medium graphs.

To begin, the analysis focuses on the influence of p for a specific, fixed value of q^*. The plot shows a general downward trend in the number of rounds as the edge creation probability increases. As with sparse graphs, this is because increasing p leads to greater edge density, which consequently reduces the graph hop diameter and the depth of the *Breadth-First Search* (BFS) tree (τ) constructed by the algorithm. As indicated by the complexity analysis (Table 3.6), the number of rounds is insensitive to the edge count itself but is significantly influenced by the BFS tree depth ($Depth(\tau)$), primarily in the *Upcast-Broadcast* phase.

Although the initial BFS tree construction is also affected by hop diameter, its contribution to the round count is minimal in comparison. Therefore, the overall number of rounds of the algorithm decreases as p increases, and the reduction is dominated by the *Upcast-Broadcast* process, consistent with our experimental results.

Following this, we analyze the dependence on q^* for a fixed p. The data shows that lower q^* values (0.001 to 0.002) achieve the most efficient performance, requiring approximately 420-430 rounds at $p = 0.01$ and slightly decreasing to approximately 374-380 rounds at $p = 0.09$. In contrast, higher q^* values result in an increased number of rounds, with $q^* = 0.009$ requiring approximately 475 rounds and $q^* = 0.007$ needing around 470 rounds for $p = 0.01$. This is because the number of rounds increases with the number of virtual nodes, which increases with increasing q^*. As with sparse graphs, demonstrated in Section 4.3.1.2, we find that the number of rounds is consistent with the empirically derived Equation (2).

Number of messages

Figure 4.14-(c) compares the number of messages required for different q^* values (ranging from 0.001 to 0.009) across various edge creation probabilities (p) (0.01 to 0.09) in medium graphs.

In general, the results for medium graphs are quite similar in trends to the results for sparse graphs. First, we can see that for every q^*, the number of messages increases with increasing edge creation probability p, as explained in Section 4.3.1. Second, the experiments demonstrate that selecting lower virtual node probabilities (q^*) leads to more efficient performance in terms of message count. As mentioned in Section 4.3.1, this is because the number of messages is mostly affected by the number of messages when constructing k-shortcut hopset, which can be estimated by using Equation (3). Therefore, with a higher q^*, the number of messages is higher compared to a lower q^*.

A noticeable pattern can be observed from the plot: for a selected p, the difference in the number of messages is larger when $q^* = 0.001$, 0.002, and 0.003, while it becomes smaller for $q^* = 0.005$, 0.007, and 0.009. This can be explained by the number of virtual nodes selected. Our analysis shows that, given a specific p and varying q^*, the process of constructing the k-shortcut hopset accounts for the majority of the messages exchanged in the algorithm. Theoretically, the number of messages required to construct the k-shortcut hopset, as described in Algorithm 2, is given by Equation (3). As shown in Table 4.9, the number of messages estimated by Equation (3) closely aligns with the values observed in the experimental results; the difference between the theoretically expected value and the experimentally observed value is a bit larger when $|V^{VIRT}|$ is close to 6, which is expected, because every node sends $min\{|V^{VIRT}|, 6\}$ messages.

Computational time

Figure 4.14-(d) compares computational times for different q^* values (ranging from 0.001 to 0.009) across various edge creation probabilities (p) (0.01 to 0.09) in medium graphs. Firstly, for any fixed q^*, increasing p consistently leads to longer execution times. It can be explained that as p rises, nodes interact with more

Table 4.9: Number of messages when constructing k-shortcut hopset for $p = 0.09$.

q^*	$\lvert V^{\text{VIRT}}\rvert$	**Eq. (3)**	**Actual value**	**% Difference**
0.001	2.08	35,479,269.0	35,147,129	0.9%
0.002	3.13	53,389,476.9	52,560,659	1.5%
0.003	4.11	70,105,671.0	66,597,437	5%
0.005	6.09	102,344,045.3	89,073,706	12.9%
0.007	8.51	102,344,045.3	98,358,233	3.8%
0.009	10.05	102,344,045.3	99,885,025	2.4%

neighbors, which significantly increases the computational workload required for key algorithmic steps, including constructing the k-th shortcut hopset and computing shortest distances. As shown in Figure 4.15, these components become more computationally intensive in denser graphs, therefore driving up the overall algorithm execution time observed.

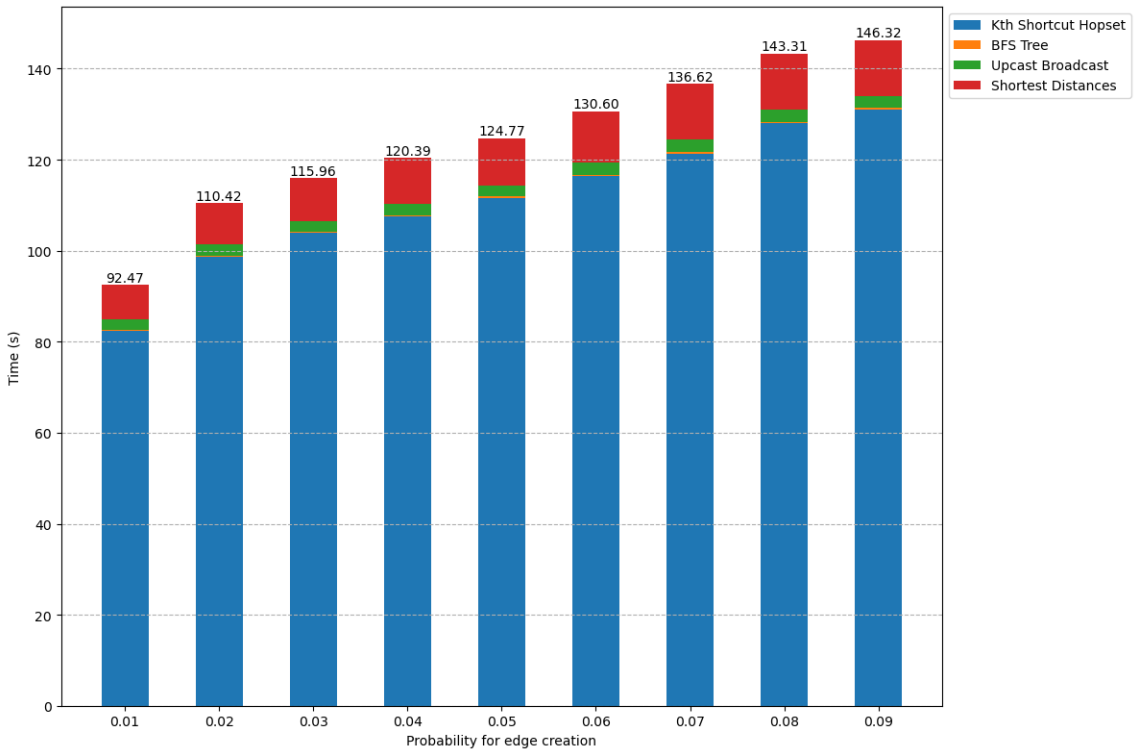


Figure 4.15: Computational times with different p values for $q^* = 0.003$.

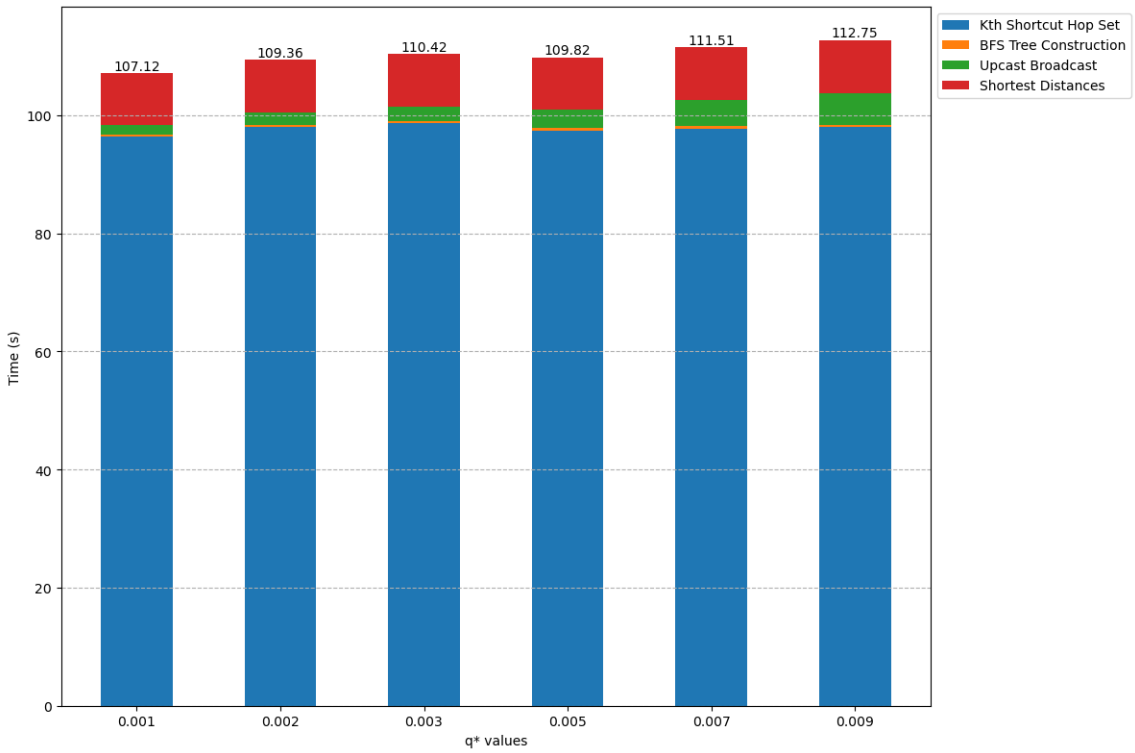


Figure 4.16: Computational times with different q^* values for $p = 0.02$.

Secondly, in medium graphs, for a fixed p, the experiments demonstrate that selecting lower virtual node probabilities (q^*) leads to better computational efficiency, which has the same behaviour as sparse graphs in Section 4.3.1. To be more specific, the results show that lower q^* values (0.001 to 0.002) achieve the most efficient computational times, around 89-92 seconds at $p = 0.01$, and increase to about 130 and 136 seconds at $p = 0.09$, respectively. In contrast, higher q^* values (0.003 to 0.009) result in longer computational times, requiring approximately 93-98 seconds at $p = 0.01$, then increasing to 141-146 seconds at $p = 0.9$. This can be explained by looking at the connection between q^* and how well the algorithm performs, which can be seen in Figure 4.14-(a) and Figure 4.16. As q^* goes up, the algorithm chooses more virtual nodes. This larger number of nodes directly causes the *Upcast-Broadcast* part (Algorithm 12) to take longer.

4.3.2.3 Performance of Elkin-D

Here, we analyze the performance of the Elkin-D algorithm, which operates by choosing virtual nodes that are at least d hops away from each other. Our study focuses on the impact of the parameter d, specifically exploring $d = 2, 3$ on the algorithm's behavior. Recall that for sparse graphs, we considered $d = 4, 5$ as well. However, for medium-density graphs, the results for $d = 4, 5$ are almost the same as for $d = 3$, so we only

report results for $d = 2, 3$.

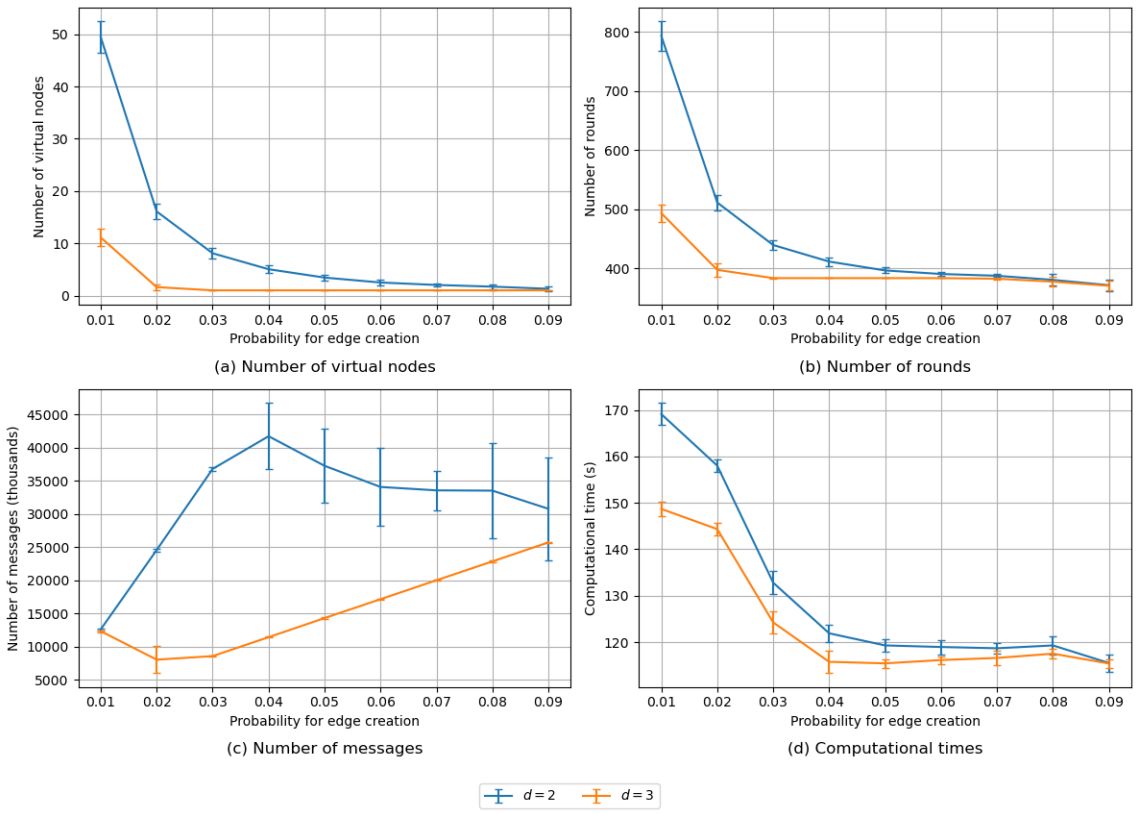


Figure 4.17: Elkin-D performance for medium-density graphs

Number of virtual nodes

As we can see from Figure 4.17-(a) and as explained in Section 4.3.1.3, for any fixed d, increasing the graph density parameter p leads to fewer virtual nodes.

Number of rounds

Figure 4.17-(b) illustrates the relationship between edge creation probability and the average number of rounds required for medium graphs, comparing the results for $d = 2$ and $d = 3$. First, we note that with increasing p and a fixed d, the number of rounds decreases, which is explained in Section 4.3.1.3. To be more specific, the 2-hop configuration shows significantly higher round counts at low edge creation probabilities, peaking at approximately 800 rounds when $p = 0.01$. However, its performance improves dramatically as the edge creation probability increases, with a steep decline in round count up to $p = 0.04$. For the 3-hop configuration, starting at around 500 rounds when $p = 0.01$, the curve shows a gentler decline compared to the 2-hop configuration across varying graph densities. The two settings converge at approximately $p = 0.06$, after which they perform similarly, both requiring around 390 rounds. Beyond this point, both configurations

show minimal decrease with increasing edge creation probability (p), reaching 370 rounds at $p = 0.09$

Examining the effect of different d on a specific value of p, higher d leads to a lower value of communication rounds as explained in Section 4.3.1.3, where the performance of $d = 3$ is better compared to $d = 2$ for $p < 0.06$.

Number of messages

Here we see that the shapes of the curves are different for $d = 2$ and $d = 3$. To investigate this, we drill down on the number of messages sent in different components of the algorithm, as shown in Figures 4.18 and 4.19. We see that the number of messages sent by the component of the algorithm that computes shortest distances increases with increasing p for both values of d. However, the number of messages sent in the k-shortcut hopset construction first increases and then decreases with increasing p for $d = 2$, while it first decreases and then increases for $d = 3$. Tables 4.10 and 4.11 show the number of messages given by our theoretical analysis as well as those experimentally measured, and find them to be extremely close.

The explanation for the shape of the curves is similar to that given for $d = 4$ and $d = 5$ for the case of sparse graphs (see Section 4.3.1.3). Essentially, the number of messages sent is proportional to the number of edges in the graph and inversely proportional to the number of virtual nodes, when the latter is less than 6. As p increases, the number of edges increases, and the number of virtual nodes decreases. As long as the number of virtual nodes is greater than 6, the number of messages keeps increasing with p. Once the number of virtual nodes reaches 6 and starts decreasing beyond this, we expect the number of messages to decrease. For even higher values of p, if the number of virtual nodes stabilises, once again, the number of messages should increase with p. This is what we see happening. For $d = 2$, the number of virtual nodes is greater than 6 for $p < 0.04$ and it is around 6 at $p \approx 0.04$, so we see the number of messages increases up to $p = 0.05$ and subsequently decreases. For $d = 3$, the number of virtual nodes drops from 11.11 to around 1 from $p = 0.01$ to $p = 0.02$, and then stays at 1. So we see a drop in the number of messages from $p = 0.01$ to $p = 0.02$, followed by an increase in the number of messages, as the number of virtual nodes stays stable at 1 while the number of edges keeps increasing.

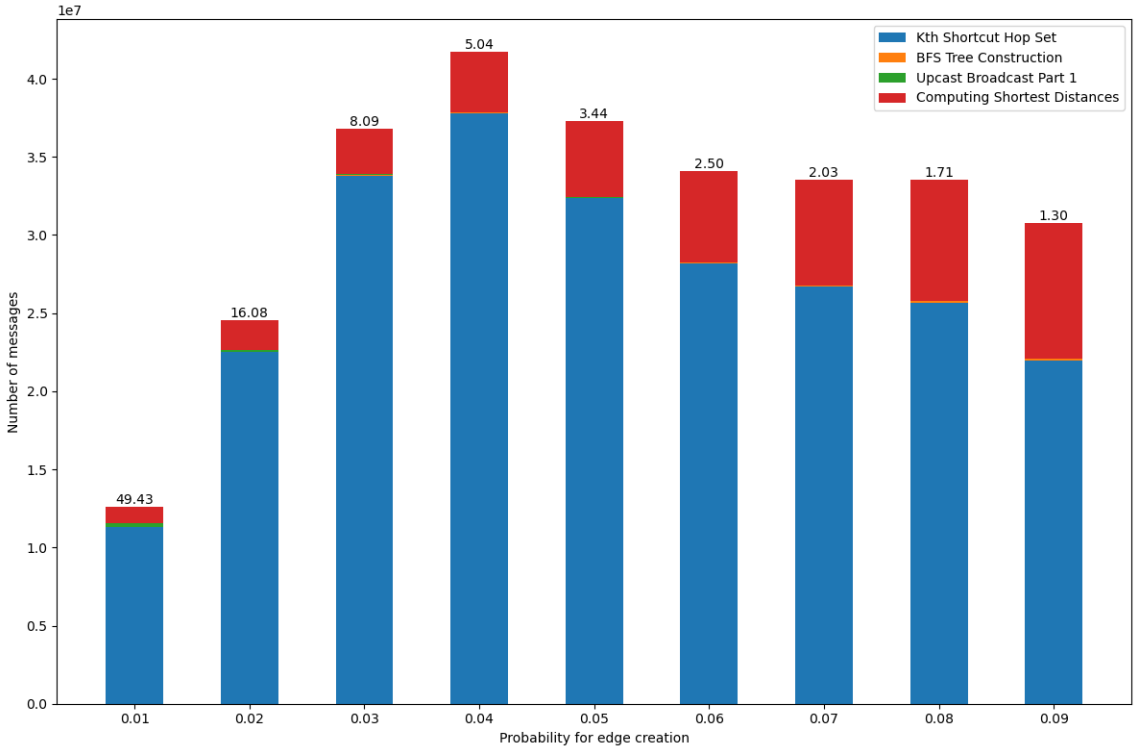


Figure 4.18: Number of messages in k-shortcut hopset for Elkin-D with $d = 2$

Table 4.10: Comparison of Equation (3) vs. experimental message counts in k-shortcut hopset for different p values for Elkin-D with $d = 2$

Probability (p)	$\lvert V^{\text{VIRT}} \rvert$	**Eq.** (3)	**Real number of messages**	**% Difference**
0.01	49.43	11,387,455	11,283,608	0.91%
0.02	16.08	22,717,011	22,509,357	0.91%
0.03	8.09	34,118,287	33,806,218	0.91%
0.04	5.04	38,202,695	37,779,937	1.11%
0.05	3.44	32,648,164	32,347,264	0.92%
0.06	2.50	28,446,601	28,188,764	0.91%
0.07	2.03	26,943,839	26,699,166	0.91%
0.08	1.71	25,923,199	25,683,604	0.92%
0.09	1.30	22,174,543	21,968,110	0.93%

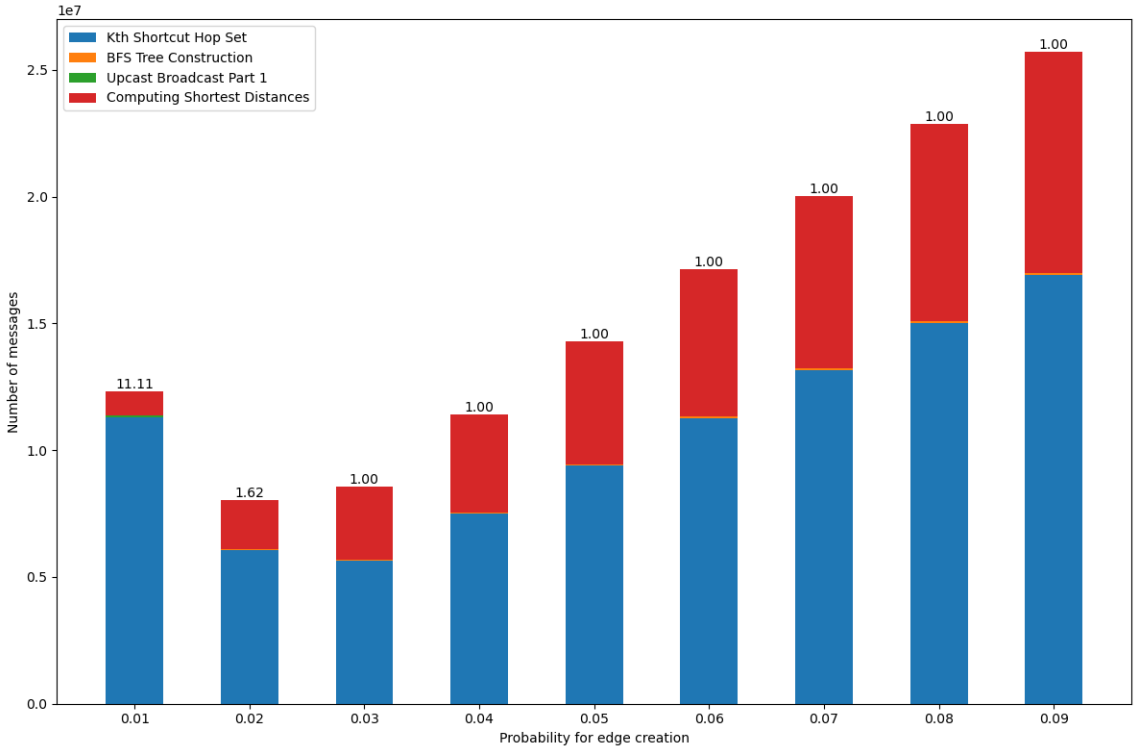


Figure 4.19: Number of messages in k-shortcut hopset for Elkin-D with $d = 3$

Table 4.11: Comparison of Equation (3) vs. experimental message counts in k-shortcut hopset for different p values for Elkin-D with $d = 3$

Probability (p)	$\lvert V^{\text{VIRT}} \rvert$	**Eq.** (3)	**Actual value**	**% Difference**
0.01	11.11	11,387,455	11,283,298	0.91%
0.02	1.62	6,133,593	6,078,684	0.90%
0.03	1.00	5,686,381	5,634,363	0.91%
0.04	1.00	7,579,900	7,510,560	0.91%
0.05	1.00	9,490,745	9,403,924	0.91%
0.06	1.00	11,378,640	11,274,550	0.91%
0.07	1.00	13,272,827	13,151,410	0.91%
0.08	1.00	15,159,766	15,021,087	0.91%
0.09	1.00	17,057,341	16,901,305	0.91%

Computational time

Figure 4.17-(d) illustrates the comparison of computational times between 2-hop and 3-hop implementations for medium-density graphs.

Generally, increasing p leads to faster execution times for all d, which is already explained in Section 4.3.1.3. To be more specific, 2-hop execution takes about 169 seconds at $p = 0.01$. The 3-hop time, starting at roughly 148 seconds at $p = 0.01$, demonstrates this downward trend by falling to about 115 seconds at $p = 0.09$.

Next, we consider the effect of d for a fixed p. Our experiments demonstrate that selecting a higher d leads to better computational efficiency, which has the same pattern as sparse graphs, and it is explained in 4.3.1.3. As can be seen from the graph, at $p = 0.01$, the time taken for $d = 2$ is approximately 169 seconds, while that for $d = 3$ is lower at about 150 seconds.

4.3.2.4 Comparison between Elkin, Elkin-R, Elkin-D, and Bellman-Ford

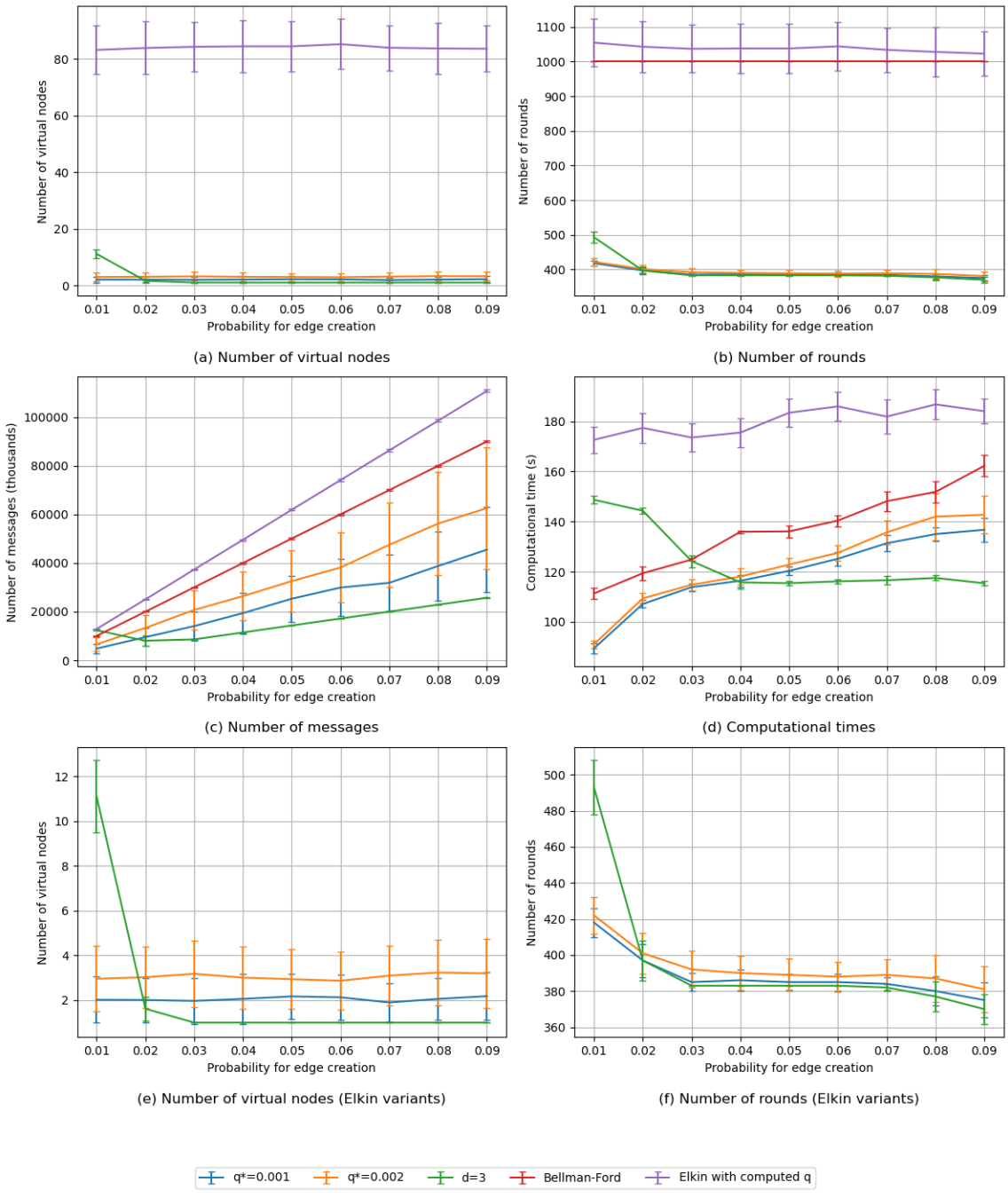


Figure 4.20: Medium density graphs performance for all algorithms

In this section, we compare the performance of our optimized Elkin, Bellman-Ford with the best versions of Elkin-D and Elkin-R as determined empirically in the previous sections. For Elkin-R, our experimental results indicate that $q^* = 0.001$ and $q^* = 0.002$ give the best results, while for Elkin-D, $d = 3$ gives the

best results. Figure 4.20-(a) shows the number of virtual nodes for all algorithms. Similar to sparse graphs in 4.3.1.4, the number of virtual nodes for Elkin is much greater than other algorithm variants. Among the tested q^* values, the average number of virtual nodes is lowest at ≈2 for $q^* = 0.001$, rising slightly to ≈3 for $q^* = 0.002$. In comparison, the $d = 3$ setting shows a different pattern dependent on p: 11 virtual nodes are selected at $p = 0.01$, but this drops to only ≈1 virtual node for $p > 0.01$. In general, Figure 4.20-(e) shows that Elkin-D with $d = 3$ has the lowest number of virtual nodes for $p > 0.01$, while for $p = 0.01$, the number of virtual nodes is lowest at $q^* = 0.001$.

Figure 4.20-(b) illustrates the comparison of the number of rounds across different variations of the Elkin algorithm and the Bellman-Ford algorithm. Similar to experiments with sparse graphs in 4.3.1.4, the Elkin variant with computed q and the Bellman-Ford algorithm demonstrates stable performance at about 1000 rounds. The remaining variants, however, require significantly lower round counts and decrease as the graph becomes denser. Among all these algorithms, Elkin-R with $q^* = 0.001$ stands out as the most stable and efficient number of rounds across all the graph densities.

Figure 4.20-(c) presents a comparison of the number of messages required across the aforementioned algorithms. The data shows distinct patterns in message complexity as edge creation probability increases from 0.01 to 0.09. Overall, this comparison demonstrates that the choice of q^* significantly impacts message complexity, with smaller fixed values of q^* generally resulting in lower message overhead across all probabilities. Following the same pattern with sparse graphs in Section 4.3.1.4, the Elkin variant with computed q exhibits the highest message overhead, and the Bellman-Ford algorithm demonstrates the second-highest message count compared to other Elkin-R and Elkin-D variants. Notably, the Elkin-D variant with $d = 3$ proves to be the most message-efficient, requiring the smallest number of messages across nearly all graph densities.

Figure 4.20-(d) presents a detailed computational time comparison for medium graphs, with results averaged across 100 graphs. The data reveal several distinct patterns in algorithm performance compared to sparse graphs: This detailed comparison reveals that Elkin-R with different q^* values performs relatively well; the choice of q^* value notably impacts computational efficiency, with $q^* = 0.001$ providing the most consistent and efficient performance across all edge creation probabilities from 89 seconds at $p = 0.01$ to 136 seconds at $p = 0.09$. Elkin with $q^* = 0.002$ follows a similar pattern to $q^* = 0.001$, with slightly higher computational times around 91-142 seconds, showing stable performance across different edge creation probabilities (p). On the other hand, similar to sparse graphs in Section 4.3.1.4, the version of the Elkin algorithm that calculates q consistently takes the longest time to compute.

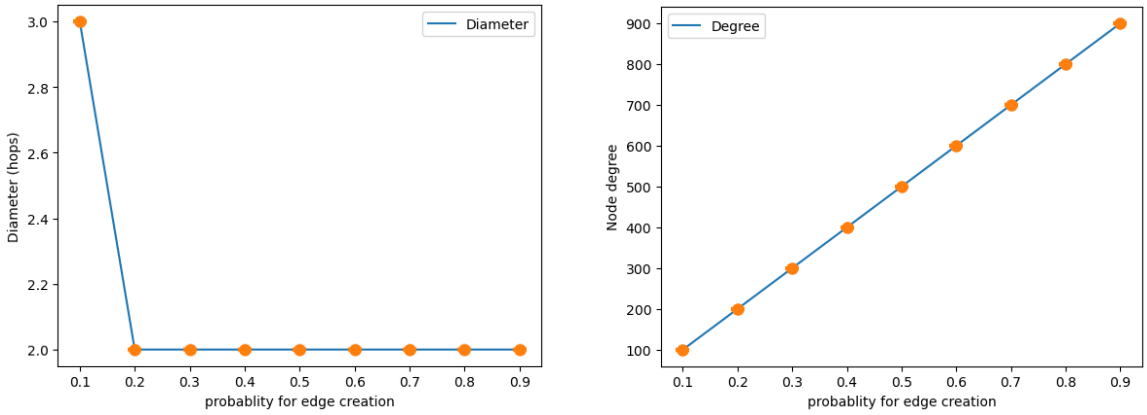
The Bellman-Ford algorithm displays a clear upward trend as edge probability increases, starting at around 111 seconds and rising steadily to approximately 162 seconds at $p = 0.09$. As the graph becomes

more connected (at higher edge probabilities, p), Elkin-R shows a significant speed advantage over Bellman-Ford. This is because the increased number of connections means each point in the graph has more neighbors to communicate with and update information from in each step, which slows down Bellman-Ford considerably. Furthermore, Elkin-R (for all p) and Elkin-D (with $p > 0.03$) demonstrates a significant speed advantage under these denser graph conditions and, as evidenced in Figure 4.20-(b), achieves fewer rounds compared to Bellman-Ford (specifically with parameters $q^* = 0.001$ and $q^* = 0.002$). Among all the algorithms, the variant Elkin-R with $q^* = 0.001$ consistently achieves the best computational efficiency across all graph densities.

4.3.3 Dense graphs

In this section, we report on our experiments with dense graphs, that is, graphs in which the edge creation probability ranges from 0.1 to 0.9.

4.3.3.1 Graph characteristics



(a) Hop diameter

(b) Average node degree

Figure 4.21: Comparison of hop diameter and average node degree for dense graphs.

As we can see from Figure 4.21a, the average hop diameter drops from about 3 to 2 hops as the probability rises from 0.1 to 0.2. From 0.2, the hop diameter stabilizes around 2 hops, showing no change with further p increases up to 0.9. It is logical that a higher edge creation probability (p) results in a denser graph, which in turn reduces the graph's hop diameter. However, the hop diameter can be reduced to the value of 2 at most, since once the graph is dense enough (from $p = 0.2$ in this case), most non-adjacent node pairs can reach each other via a neighbor (a 2-hop path). Further increases in p (up to 0.9) introduce more edges but are

unlikely to reduce the hop diameter below 2 for a graph that is not yet a clique. Only when $p = 1$ does the graph become complete, achieving the minimum possible hop diameter of 1.0.

For the node degree, Figure 4.21b shows a clear linear relationship between the probability of edge creation and the average node degree in a network. As the probability for edge creation increases from 0.1 to 0.9, the average node degree rises steadily from about 100 to 900, confirming the value obtained using the formula $deg_{avg} = p \cdot n$.

4.3.3.2 Performance of Elkin-R

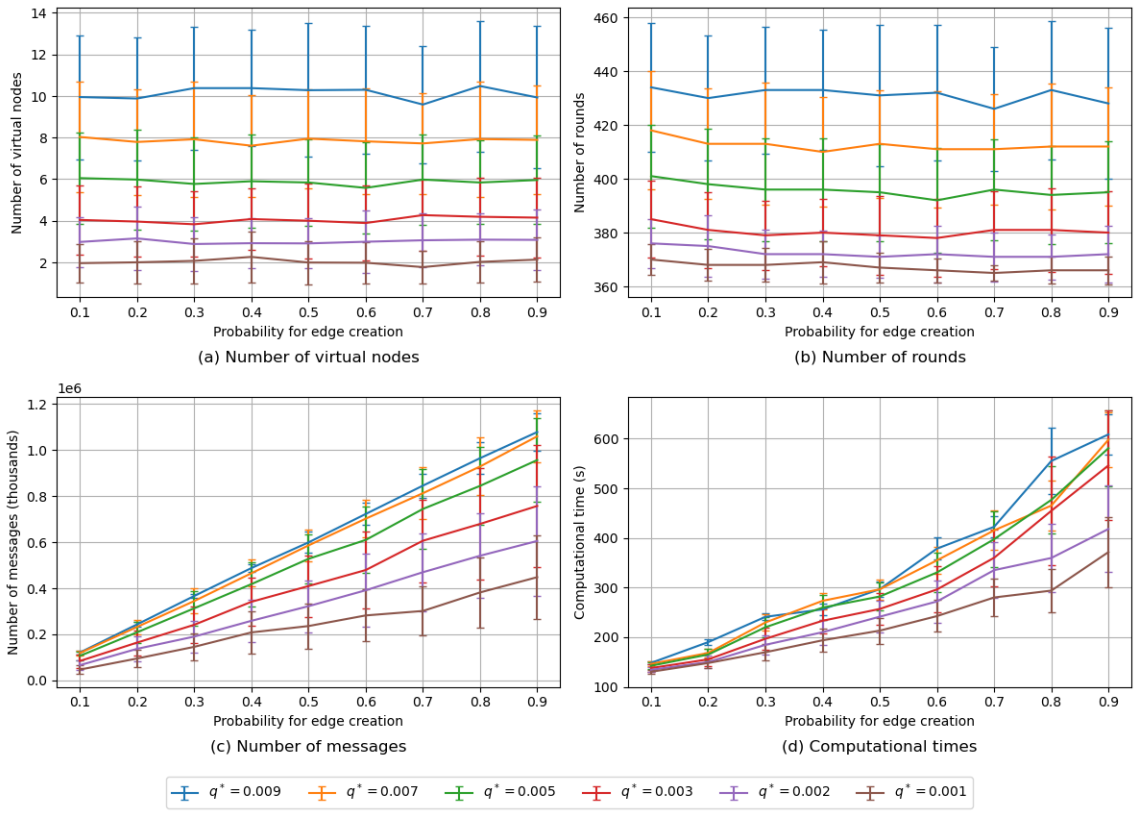


Figure 4.22: Elkin-R performance for dense graphs

Number of virtual nodes

Figure 4.22-(a) illustrates that increasing the probability p of a node being chosen as a virtual node results in a higher total number of virtual nodes being selected, since the expected number of virtual nodes is $q^* \cdot (n - 1) + 1$.

Number of rounds

Figure 4.22-(b) compares the number of rounds required for different virtual node selection probabilities (q^*)

in dense graphs.

When analyzing dense graphs with a constant q^*, the impact of varying the edge creation probability p (from 0.1 to 0.9) on performance is shown in Figure 4.22-(b). We observe that the number of rounds remains remarkably consistent across this range of p values. The reason for this stability appears to be that the depth of the BFS tree, constructed by the algorithm, does not change significantly within the same q^*, as supported by data in Table 4.12. Since the theoretical number of rounds estimated from Equation (2) is strongly tied to this tree depth (as indicated in Table 3.6, particularly for phases like *Upcast-Broadcast*), a stable depth naturally translates into a stable round count. This contrasts sharply with sparse and medium-density graph scenarios, where increasing p typically has a more pronounced effect on reducing graph hop diameter and tree depth. Therefore, for any given q^* in these dense graphs, the algorithm's round performance is largely insensitive to adjustments in p.

Table 4.12: Average $Depth(\tau)$, number of virtual nodes $|V^{\text{VIRT}}|$, and number of rounds with $q^* = 0.001$

p	$Depth(\tau)$	$\lvert V^{\mathbf{VIRT}}\rvert$	**Eq.(2)**	**Experiment**
0.1	2.06	1.96	17.1	15
0.2	2	2	17	15
0.3	2	2.07	17.2	15
0.4	2	2.26	17.8	16
0.5	2	1.99	16.9	15
0.6	2	1.98	16.9	15
0.7	2	1.77	16.3	14
0.8	2	2.02	17	14
0.9	2	2.13	17.4	15

Regarding the effect of q^* for a fixed value of p, similar to sparse and medium-density graphs in Section 4.3.1.2 and Section 4.3.2.2, lower q^* values (0.001 to 0.002) achieve the most efficient performance.

Number of messages

Figure 4.22-(c) compares the number of messages required for different q^* values (ranging from 0.001 to 0.009) across various edge creation probabilities (p) (0.1 to 0.9) in dense graphs. First, we can see that selecting lower virtual node probabilities (q^*) leads to more efficient performance regarding message count. This observation aligns with the analysis in Section 4.3.1.2 and Section 4.3.2.2; the number of messages is primarily driven by the k-shortcut hopset construction, estimated by Equation (3). Consequently, a higher q^* increases messaging during this critical phase, resulting in higher overall counts compared to using a lower q^*. Additionally, consistent with Section 4.3.2.2, for any fixed q^*, message totals also rise as the edge creation probability p increases.

We observe a similar noticeable pattern in the plotted data compared with sparse and medium density: with a fixed p, the difference in the number of messages is greater when q^* is 0.001, 0.002, or 0.003, but less pronounced for q^* equal to 0.005, 0.007, and 0.009. This can be attributed to the number of virtual nodes involved, which is lower than 6 for the first three values, and 6 or more for the higher 3 values of q^*. The reasons why this makes a difference have been described already for sparse graphs in Section 4.3.1.2.

Computational time

Figure 4.22-(d) compares computational times for different q^* values (ranging from 0.001 to 0.009) across various edge creation probabilities (p) (0.1 to 0.9) in dense graphs. Two key trends are consistent with findings for medium-density graphs presented in Section 4.3.2.2. Firstly, for any fixed q^*, increasing p consistently leads to longer execution times. Secondly, for a fixed p, selecting lower virtual node probabilities ($q^* = 0.001$ or $q^* = 0.002$) leads to better computational efficiency.

4.3.3.3 Performance of Elkin-D

Here, we evaluate the Elkin-D algorithm, which selects virtual nodes separated by at least d hops. For dense graphs, we restrict this analysis to $d = 2$. This is because preliminary results showed that higher values ($d = 3, 4, 5$), previously relevant for sparse and medium graphs, yielded nearly identical performance to $d = 2$ in the dense setting.

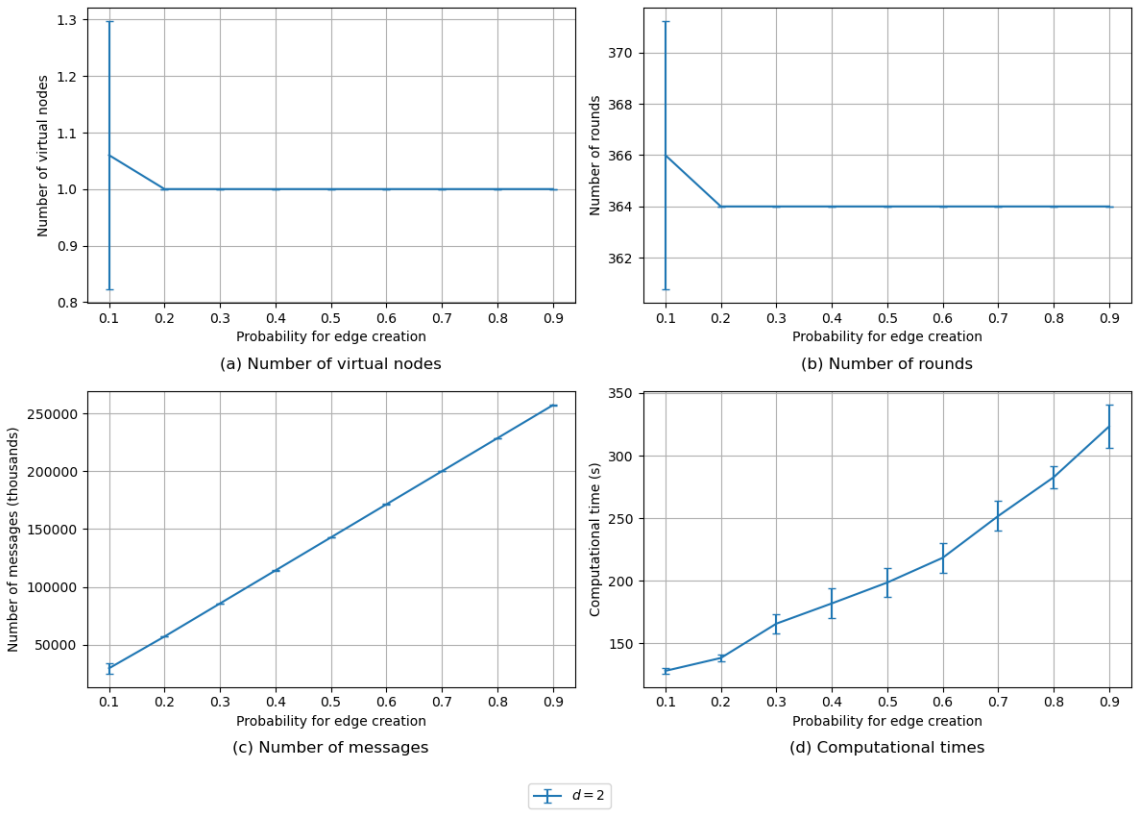


Figure 4.23: Elkin-D performance

Number of virtual nodes

Figure 4.23-(a) displays the average virtual node count for Elkin-D with $d = 2$ against varying p. Consistent with Section 4.3.1, increasing density (p) generally reduces the number of selected virtual nodes.

At $p = 0.1$, the average value is $\approx$1.06 with high variance. This occurs because the graph hop diameter is 3 (Figure 4.21a), exceeding $d = 2$. As a result, depending on the specific graph, either only the source node s (1) or the source s plus a second node ≥ 2 hops away (2) are selected, averaging slightly above 1.

For $p \geq 0.2$, the number of virtual nodes at exactly 1.0. This is because the hop diameter drops to 2 (Figure 4.21a), equaling d. In this case, where the hop diameter equals d, the algorithm consistently selects only the source node s.

Number of rounds

Figure 4.23-(b) illustrates the relationship between edge creation probability and the average number of rounds required for dense graphs, comparing the results for $d = 2$. We note that with increasing p and a fixed d, the number of rounds decreases, which is explained in Section 4.3.1 and 4.3.2. However, this trend

is affected by virtual node selection. The significant variance in rounds observed at $p = 0.1$ corresponds directly to the variable number of virtual nodes selected at that specific density (Figure 4.23-(a)). The number of rounds is stable at 394 from $p = 0.2$ to $p = 0.9$ because the number of virtual nodes stabilizes at exactly 1 (Figure 4.23-(a)).

Number of messages

Figure 4.23-(c) shows that the number of messages exchanged by Elkin-D ($d = 2$) on dense graphs increases with the density parameter p. This occurs because the number of virtual nodes becomes fixed at approximately one for $p \geq 0.2$. With a constant virtual node structure, increasing graph density leads to more message exchanges. This trend parallels that of $d = 3$ on medium-density graphs (Section 4.3.2.3).

Computational time

Figure 4.23-(d) illustrates the comparison of computational times for 2-hop implementations for dense graphs. This occurs because the $d = 2$ condition consistently selects approximately one virtual node for these dense graphs, largely independent of p. Therefore, as the graph becomes denser, the computational time required to process the paths related to this fixed virtual node structure increases. Furthermore, with effectively only one virtual node, the performance trend of Elkin-D ($d = 2$) follows the same pattern as the Elkin-R algorithm using any fixed q^* as p increases, as discussed in Section 4.3.2.2.

4.3.3.4 Comparison between Elkin, Elkin-R, Elkin-D, and Bellman-Ford

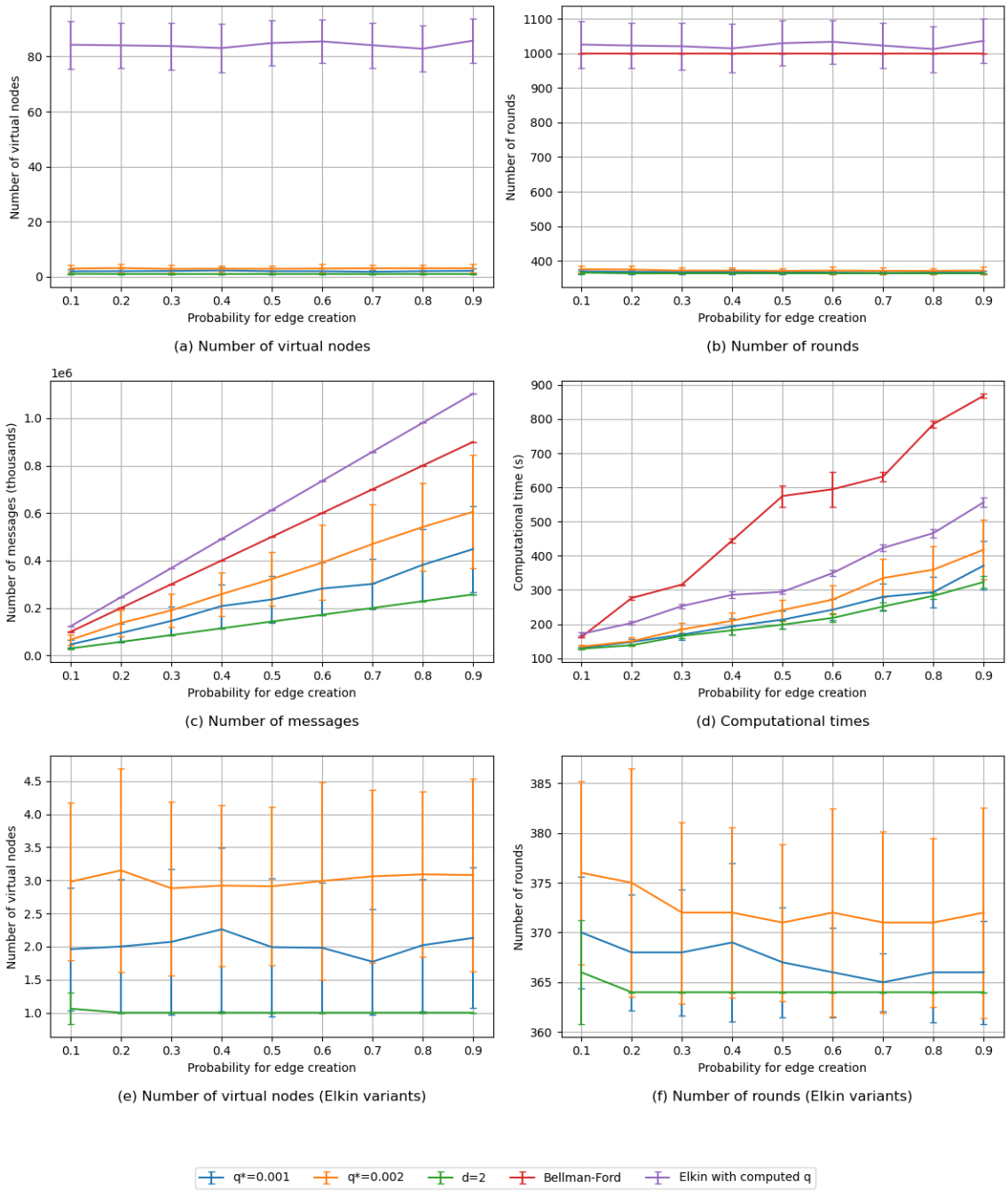


Figure 4.24: Dense graphs performance for all algorithms

This section compares our optimized Elkin's algorithm against Bellman-Ford and the empirically best-performing configurations of Elkin-D (using $d = 2$) and Elkin-R (using $q^* \in \{0.001, 0.002\}$), as determined previously.

Figure 4.24-(a) displays the number of virtual nodes for these algorithms. Consistent with observations for sparse and medium-density graphs (Sections 4.3.1.4 and 4.3.2.4), the optimized-Elkin algorithm selects significantly more virtual nodes than the other variants. Figure 4.24-(e) shows that the Elkin-D with $d = 2$ achieves the lowest number of virtual nodes, which is about 1.0 across several graph densities.

Figure 4.24-(b) compares the number of rounds for the various algorithms on dense graphs, following the same patterns observed for sparse and medium-density graphs (Sections 4.3.1.4 and 4.3.2.4). Similarly, Figure 4.24-(c) shows the number of messages required by each algorithm on dense graphs. The observed trends are consistent with patterns from medium-density graphs (Section 4.3.2.4). Notably, the Elkin-D variant with $d = 2$ demonstrates the most efficient algorithm in both number of messages and rounds, consistently requiring the fewest of each across almost all graph densities evaluated.

Figure 4.24-(d) compares computational times for different algorithm variants in dense graphs. Similar to medium-density in Section 4.3.2.4, Elkin-R with different q^* values and Elkin-D with $d = 2$ perform better than Bellman-Ford. A noticeable feature is that, optimized Elkin significantly outperforms Bellman-Ford in these denser graphs (where $p > 0.1$) because the increased connectivity forces Bellman-Ford to do much more work per step due to the larger number of neighbors each node must process even the number of rounds for optimized Elkin is slightly higher than that of Bellman-Ford which is described in Figure 4.24-(b). It is important to note that, across several algorithms, the variant of Elkin with $d = 2$ achieves the most efficient computational time across different p.

4.3.4 Discussion

Table 4.13: Summary of best parameters and graph characteristics for optimal number of messages and number of rounds.

Graph types	**Best q^***	**Number of virtual nodes (for q^*)**	**Best d**	**Number of virtual nodes (for d)**	**Average hop diameter**
Sparse	0.001	2	5	3.76	7.17
Medium	0.001	2	3	2.19	3.39
Dense	0.001	2	2	1.01	2.10

Table 4.13 summarizes the best parameters for each type of random graph. To be more specific, our analysis reveals different optimal algorithm configurations depending on the graph density:

For sparse graphs, Elkin-R configured with $q^* = 0.001$ generally provides the best performance regarding computational time and the number of communication rounds. However, concerning message complexity, the optimal choice shifts with density in this way: while Elkin-R may be competitive at the lowest densities,

Elkin-D with $d = 5$ becomes more message-efficient as the graph density increases slightly (specifically noted for $p = 0.007$ to $p = 0.009$).

In medium-density graphs, Elkin-R with $q^* = 0.001$ again demonstrates the lowest computational time. Regarding optimizing both message count and the number of rounds, Elkin-D with $d = 3$ proves to be the optimal solution.

For dense graphs, Elkin-D with $d = 2$ consistently outperforms all other tested variants across computational time, message complexity, and number of rounds.

After considering the performance across different graph types, we can conclude that to achieve optimal performance, balancing computational time, message complexity, and rounds is typically achieved when the chosen parameters result in a small expected number of virtual nodes, specifically around 1 or 2. For Elkin-R, this means selecting a small q^* value where $1 + q^* \cdot (n - 1)$ is close to 1 or 2, n is the number of nodes. For Elkin-D, this involves choosing a d (relative to the graph's characteristics) closer to the hop diameter of G, which limits the virtual nodes selected.

4.4 Experiments on communication network topologies

4.4.1 CONUS network

The CONUS network graph is based on a real-life telecommunication network, and has been used in numerous studies (see for example [37] and [41]). Its hop diameter is 14 and average node degree is 2.64.

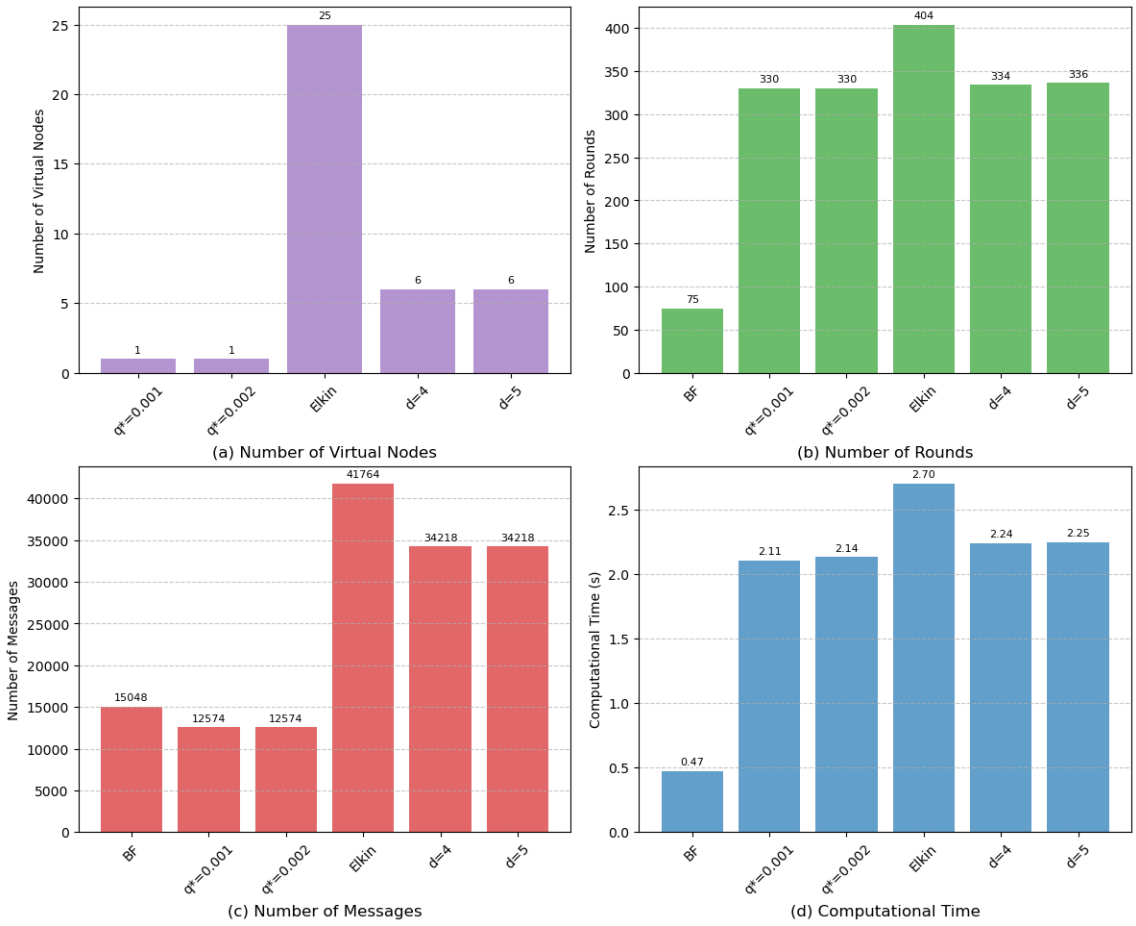


Figure 4.25: CONUS graph performance for all algorithms.

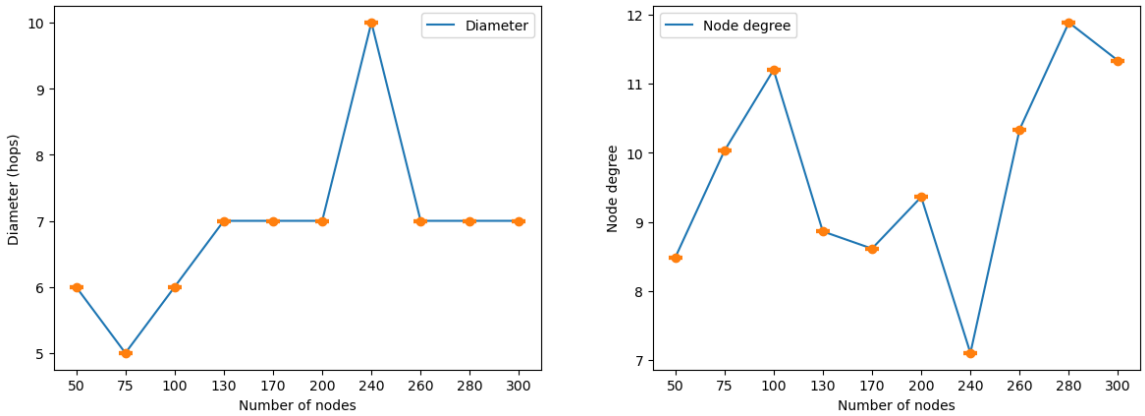
Since the CONUS graph is sparse, similar to the sparse graphs analyzed in Section 4.3.1, we applied the configuration previously identified as best for sparse networks. The results presented in Figure 4.25 show that Bellman-Ford significantly outperforms the other tested algorithms in terms of computational time and number of rounds. This efficiency is likely attributable to the CONUS graph's small size (75 nodes) and sparsity. However, regarding message complexity, the configurations using $q^* = 0.001, 0.002$ achieved better results than Bellman-Ford.

4.4.2 GNN competition graphs

We evaluated the performance of several algorithms using the Test and Validation graph datasets from the Graph Neural Networking Challenge [39].

4.4.2.1 Test dataset

Graph characteristics



(a) Hop diameter

(b) Node degree

Figure 4.26: Comparison of hop diameter and node degree for graphs from GNN competition.

Performance comparison

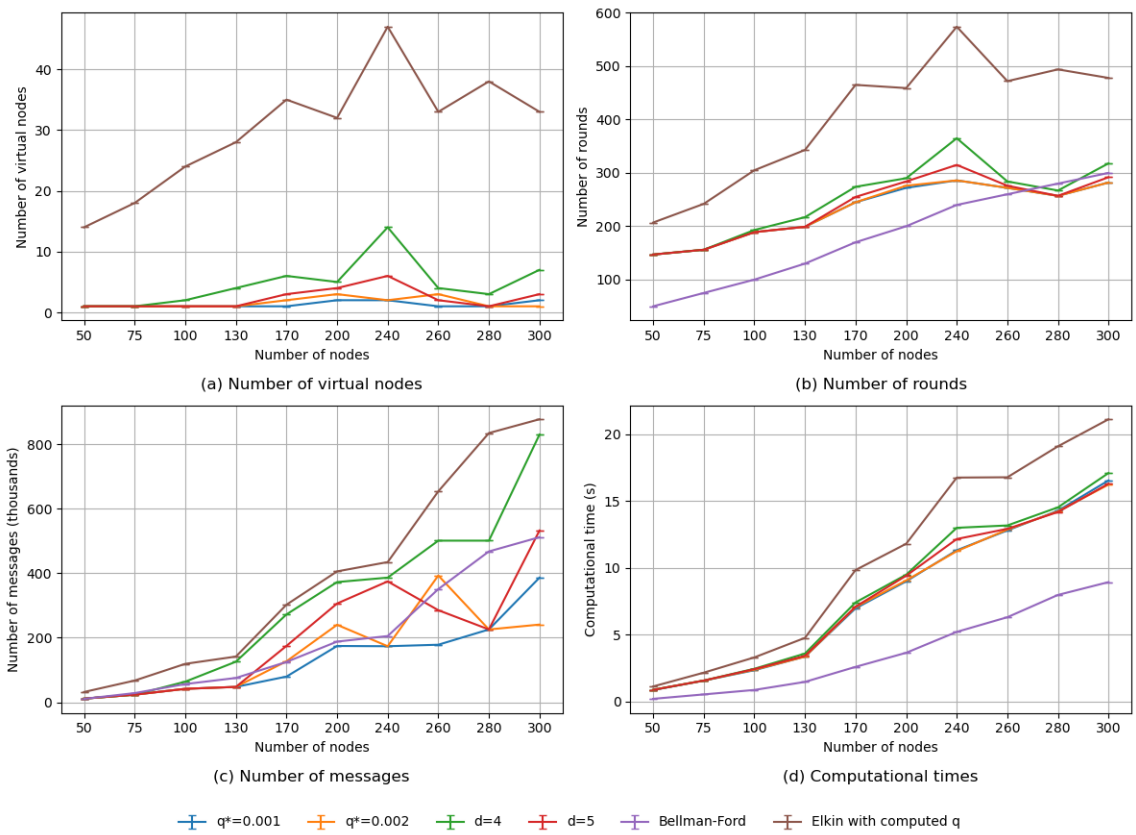


Figure 4.27: Test graphs performance

The average node degree range for the test dataset graphs aligns with the range observed for the sparse graphs discussed in Section 4.3.1. Consequently, we utilized the optimal configuration derived from the sparse graph experiments when evaluating this dataset.

Figure 4.27-(a) shows the number of virtual nodes for all algorithms. Similar to sparse graphs in 4.3.1.4, the number of virtual nodes for Elkin is much greater than other algorithm variants. Generally, we can see Elkin-R with $q^* = 0.001, 0.002$ and Elkin-D with $d = 5$ select the fewest number of virtual nodes.

Figure 4.27-(b) compares the number of rounds required by different algorithm implementations across various competition graph sizes. Bellman-Ford consistently demonstrates superior performance in terms of round efficiency for graph sizes from 50 to 260 nodes. However, for a greater size of the graph, 280 and 300 nodes, $q^* = 0.002$ and $d = 5$ outperform Bellman-Ford. The Elkin implementation with computed q shows the poorest performance, requiring nearly twice as many rounds as Bellman-Ford for most graph sizes.

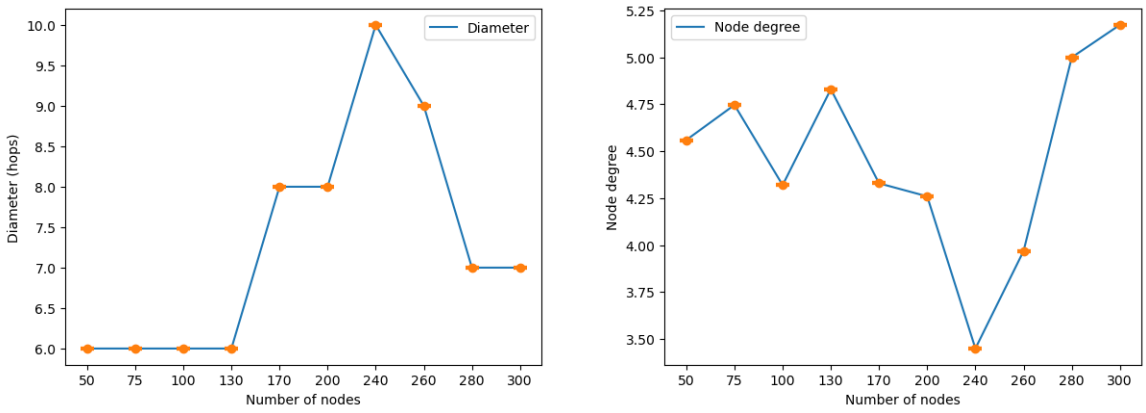
Figure 4.27-(c) illustrates the comparison of message counts across different algorithm implementations for competition graphs of varying sizes in the test dataset. The results reveal notable differences in communication efficiency among the algorithms. Among the Elkin implementations, we observe that Elkin with

$q^* = 0.001$ and $q^* = 0.002$ demonstrate the most communication-efficient performance for most graph sizes. The Bellman-Ford algorithm is communication-efficient for most graph sizes, but for graphs larger than 200 nodes, some of the Elkin variants use fewer messages.

Figure 4.27-(d) presents a comparison of computational time for various algorithm implementations across different competition graph sizes in the test dataset. Overall, in this dataset, due to the sparsity, it is clear that the Bellman-Ford algorithm consistently outperforms all Elkin variants across all graph sizes, requiring approximately 9 seconds for the largest graphs with 300 nodes. Among the Elkin implementations, those with fixed parameters ($d = 4, 5$, and fixed q^* values of 0.001, and 0.002) show similar performance patterns, with computational times ranging between 16 and 18 seconds for the largest graphs. The Elkin implementation with computed q demonstrates noticeably worse performance than other variants, requiring approximately 21 seconds for graphs with 300 nodes.

4.4.2.2 Validation dataset

Graph characteristics



(a) Hop diameter

(b) Node degree

Figure 4.28: Comparison of hop diameter and node degree for validation graphs from GNN competition

Figure 4.28a illustrates the hop diameter values for various competition graphs across different graph sizes. The results reveal distinct patterns in graph connectivity that merit attention. For smaller graphs (50 to 130 nodes), the hop diameter remains constant at 6 hops, indicating consistent shortest-path distances across these graph sizes. At 170 nodes, we observe a significant increase to 8 hops, which remains stable through the 200-node graph. The most striking feature is the sharp spike at the 240-node graph, where the hop diameter reaches 10 hops—the maximum observed across all graph sizes. This peak is followed by a rapid decline to

9 hops at 260 nodes, and then a further drop to 7 hops for the largest graphs (280 and 300 nodes).

For the node degree, Figure 4.28b illustrates the variation in average node degree across different graph sizes from the competition. Starting with an average degree of approximately 4.5 at graph size 50, we observe an increase to about 4.76 at size 75, dropping slightly to approximately 4.32 at size 100. This is followed by an increase to about 4.8 at size 130, with further sharp drops to around 3.2 at size 240, followed by a dramatic recovery that reaches a peak of nearly 5.25 at size 300.

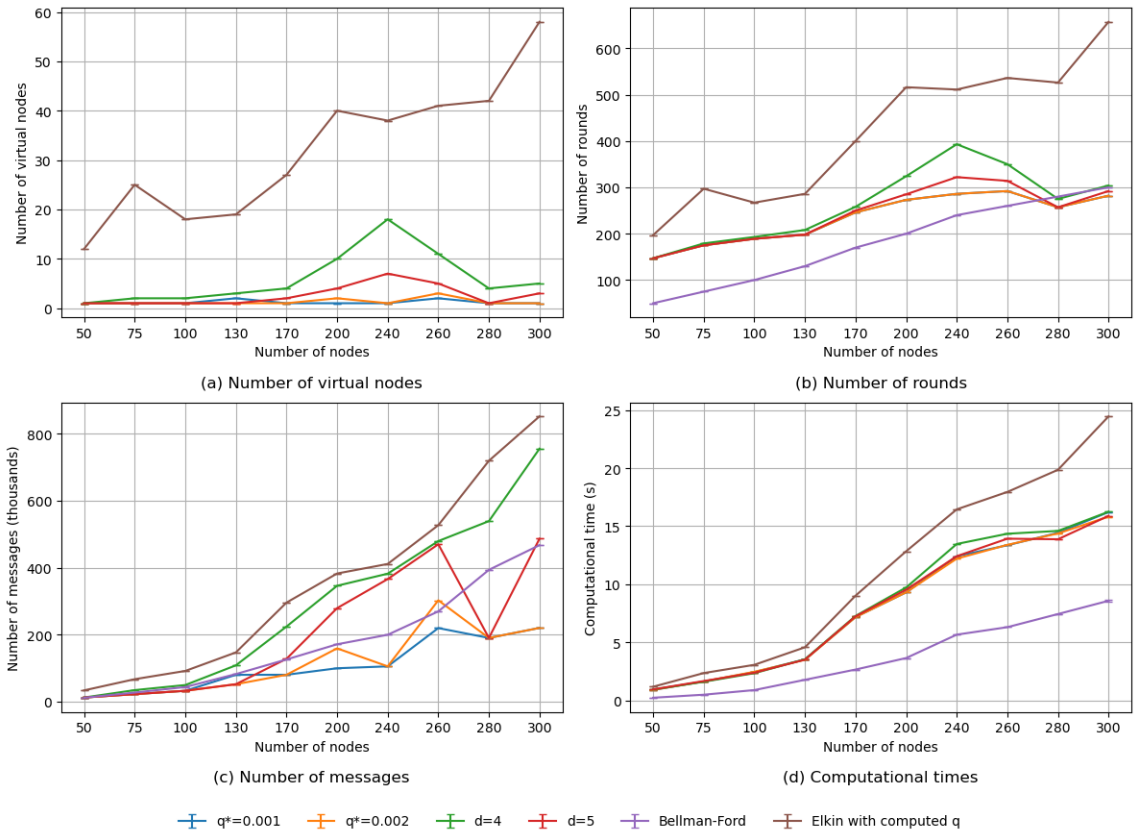


Figure 4.29: Validation graphs performance

Performance comparison

Figure 4.29 shows the results for the so-called validation graphs from [39]. The results are quite similar to those for the test graphs. Essentially, Bellman-Ford outperforms all other algorithms in terms of number of rounds and computational time, and the Elkin variants outperform the original Elkin's algorithm. For the number of messages, some of the Elkin variants perform better than Bellman-Ford.

4.4.3 Discussion

Our experimental results on graphs based on real-life communication networks appear to show that Bellman-Ford remains the best algorithm in terms of number of rounds and computational time. We believe this is because the graphs studied here are sparse as well as small. Our results on random graphs showed that Elkin's algorithm and its variants perform better in dense graphs and large graphs. Since communication network topologies tend to be sparse in general, in the next section, we attempt to see the effect of size on the relative performance of the algorithms.

4.5 Grid graphs

In this section, we study a specific set of sparse graphs: square grid graphs of degree 4. We compare all our algorithms for square grid graphs with size 34×34, 50×50, 60×60, 70×70, and 100×100. An example of a square grid graph (5×5) is demonstrated in Figure 4.30. Note that the hop diameter of an $n \times n$ grid is $2n - 2$.

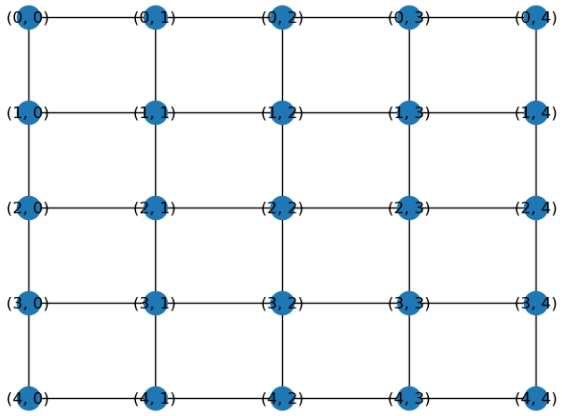


Figure 4.30: Square grid graph 5×5.

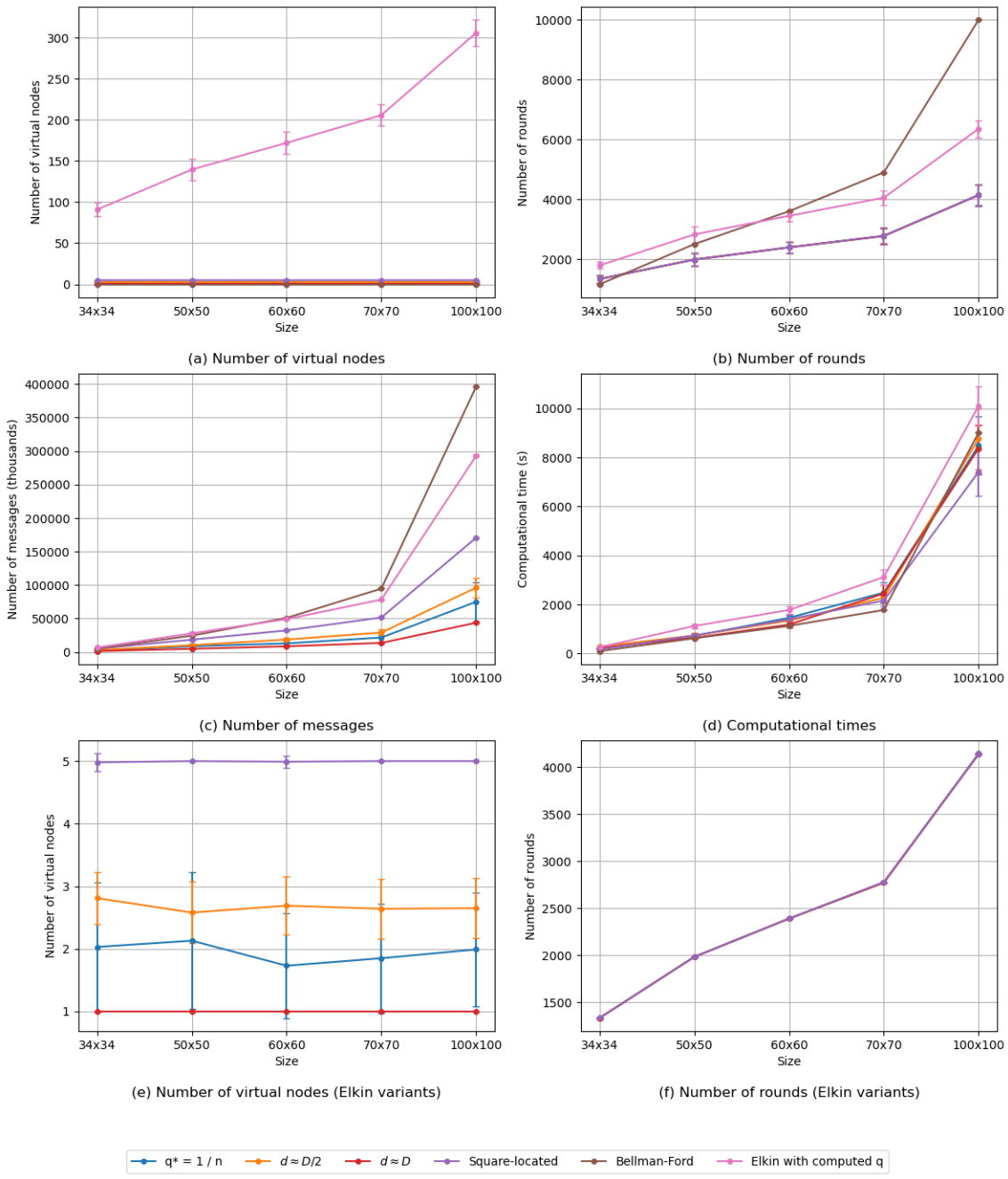


Figure 4.31: Grid graphs performance for all algorithms

In this section, we compare the performance of our optimized Elkin, Bellman-Ford, Elkin-D with $d \approx D/2$, $d \approx D$ and Elkin-R with $q^* = 1/n$, standard Bellman-Ford, and the basic Elkin's algorithm. Additionally, to explore the impact of strategic virtual node placement, we evaluate a scenario with a random source and four fixed virtual nodes, in addition to the source node. These four nodes are positioned to form a central square within the grid graph, based on the hypothesis that such a structured, central placement might enhance

overall algorithm performance by providing well-distributed virtual nodes; we call this the *square-located* version of Elkin's algorithm. Figure 4.31-(a) shows the number of virtual nodes for all algorithms. Similar to sparse and medium-density graphs in Section 4.3.1.4 and 4.3.2.4, the number of virtual nodes for Elkin is much greater than that of other algorithm variants. Elkin with $d \approx D$ and $q^* = 1/n$ select the fewest number of virtual nodes.

Figure 4.31-(b) illustrates the comparison of the number of rounds across different variations of Elkin's algorithm and the Bellman-Ford algorithm. Similar to experiments with sparse and medium-density graphs in 4.3.1.4 and 4.3.2.4, the Elkin variant with computed q and the Bellman-Ford algorithm witnessed the highest number of rounds as p increases. The remaining variants, $q^* = 1/n$, $d \approx D/2$, $d \approx D$, and square-located virtual nodes are at quite similar values and rise as the graph size increases. Notably, these variants achieve the optimal number of rounds across different graph sizes.

Figure 4.31-(c) presents a comparison of the number of messages required across the algorithms mentioned above. Overall, this comparison demonstrates that the choice of q^* significantly impacts message complexity, with smaller fixed values of q^* generally resulting in lower message overhead across all sizes. Following the same pattern with sparse, medium-density, and dense graphs in Section 4.3.1.4, 4.3.2.4, and 4.3.3.4, the Elkin variant with computed q and the Bellman-Ford algorithm demonstrates the highest message count compared to other Elkin-R and Elkin-D variants. However, for larger square grid graphs at 60×60 to 100×100, Optimized-Elkin outperforms Bellman-Ford due to the greater size of the graphs. Notably, $d \approx D$ and $q^* = 1/n$ achieve the most efficient number of messages compared to other algorithms since they select the fewest number of virtual nodes.

Figure 4.31-(d) shows the computational time taken by different algorithm variants as the size of square grid graphs increases. The graph clearly shows that for all algorithms, computational time grows as the graph size becomes larger. Looking closer at the performance between sizes 34×34 and 70×70, Bellman-Ford maintains a relatively stable performance increase, going from 129 seconds to 2320 seconds. At the 70×70 grid size, Elkin-D ($d \approx D$) and Elkin-R ($q^* = 1/n$) perform slightly better, with computational times between 1821 and 2023 seconds, respectively. The Optimized-Elkin algorithm and the Elkin-D variant with $d \approx D/2$ are less efficient at this size, taking between 2635 and 3006 seconds.

However, when the graph size expands to 100×100, Elkin-R ($q^* = 1/n$), Elkin-D ($d \approx D/2$ and $d \approx D$), and square-located virtual nodes become more efficient than Bellman-Ford, requiring approximately 6000 to 7753 seconds, respectively, compared to Bellman-Ford's 9821 seconds. Once again, the basic Elkin algorithm is the least efficient, taking around 10,000 seconds. Notably, Elkin-D with $d \approx D$ and Elkin-R with $q^* = 1/n$ are the most efficient algorithms because they select the fewest number of virtual nodes (approximately 1), which reduces computational time across different processes, making the overall computation more efficient,

particularly when the graph size increases to 100×100.

From this point, we can conclude that, square grid graph also follows the same pattern with Erdős-Rényi graphs as suggested in Section 4.3.4, the closer d is to the hop diameter or q^* is selected where $|V^{\text{VIRT}}| \approx 1$ or $|V^{\text{VIRT}}| \approx 2$, the more efficient the performance of the algorithm is. Additionally, in our experiments, the strategic placement of virtual nodes in a centrally-located square did not make a difference to the performance of the algorithm.

4.6 High-diameter graphs

Our experiments indicate that our Elkin variants perform best with very few virtual nodes. This raises the question of the performance of Elkin using a single virtual node, that is, the source node s. In that case, the k-shortcut hopset for every virtual node is empty, and Elkin's algorithm performs a lot of computation to no effect. So, does it make sense to simply run Bellman-Ford for $\sqrt{n}$ rounds? Clearly, if the graph has a diameter that is $\omega(\sqrt{n})$, then running Bellman-Ford for $\sqrt{n}$ rounds is not sufficient to find all shortest paths. Similarly, even if the hop diameter of the graph is at most $\sqrt{n}$, but there are shortest paths in terms of total weight that have $\omega(\sqrt{n})$ hops then running Bellman-Ford for $\sqrt{n}$ rounds will not compute all shortest paths.

To verify the correctness of Elkin's algorithm as well as our variants, which use far fewer virtual nodes than the prescribed number, as well as to determine the dependence of the performance on the diameter, we performed an experiment where the number of nodes stays constant but the diameter of the graph is varied. We fixed $n = 2500$, and ran experiments on the following grid graphs: 50×50, 25×100, 10×250, 5×500, and 1×2500. They have diameters $98, 123, 258, 503, 2499$ respectively. We compared the performance of Bellman-Ford, Elkin with computed q, Elkin-R with $q^* \in \{0.002, 0.0006\}$, and Elkin-D with $d \in \{D/2, D\}$, and a version of Elkin's algorithm where the (randomly chosen) source node is the only virtual node.

On graphs with high diameter, we do not expect the Elkin variants with few virtual nodes to succeed in computing the shortest paths correctly. Table 4.14 summarizes the correctness of each algorithm for each graph type. The results are averaged over 10 graphs, with source node and weights chosen randomly. We see that as the diameter increases, the Elkin variants, and even Elkin's original algorithm, do not succeed in computing all shortest paths correctly.

Table 4.14: Success Rate of Elkin's Algorithm and other variants

Graph type	**Diameter**	Correctness of shortest path computations **Optimized Elkin**	q^***=0.002**	q^***=0.0006**	$d \approx D$	$d \approx D/2$	**Single Virtual Node**
50×50	98	100%	100%	100%	100%	100%	100%
25×100	123	100%	100%	100%	100%	100%	100%
10×250	258	70%	10%	0%	0%	0%	0%
5×500	503	10%	10%	0%	0%	0%	0%
1×2500	2499	10%	0%	10%	30%	40%	10%

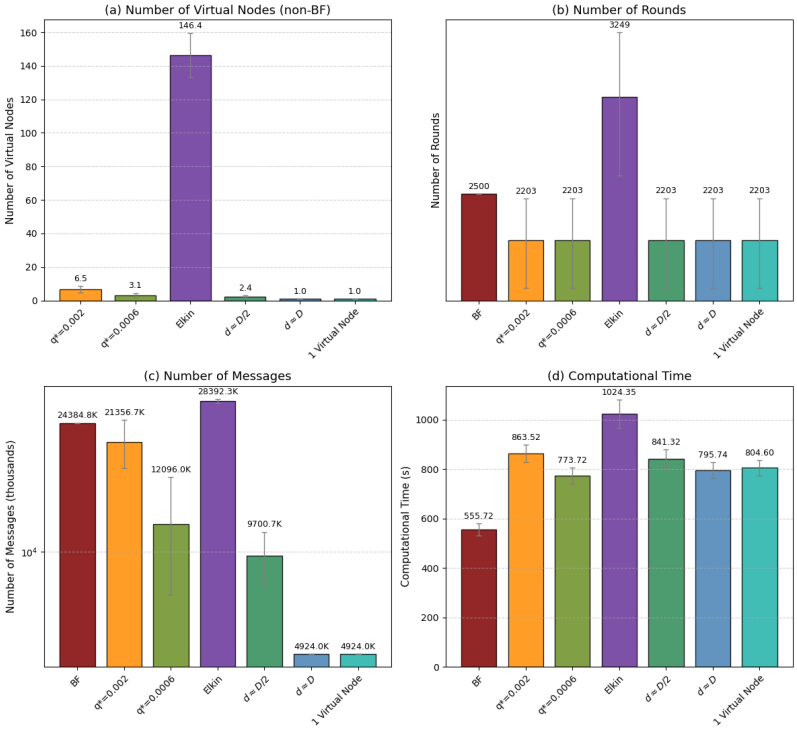


Figure 4.32: Performance for 25×100 grid graphs

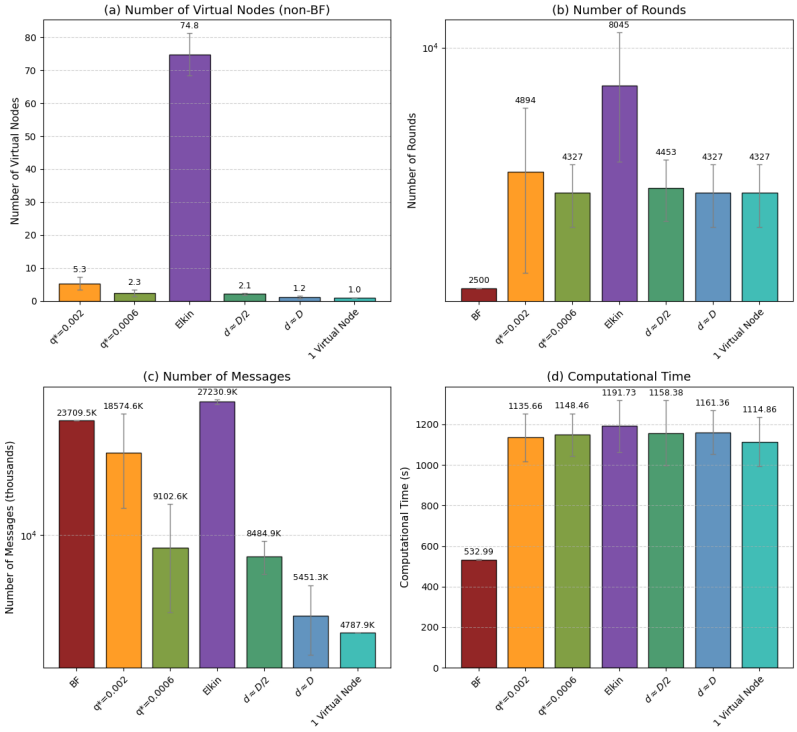


Figure 4.33: Performance for 10×250 grid graphs

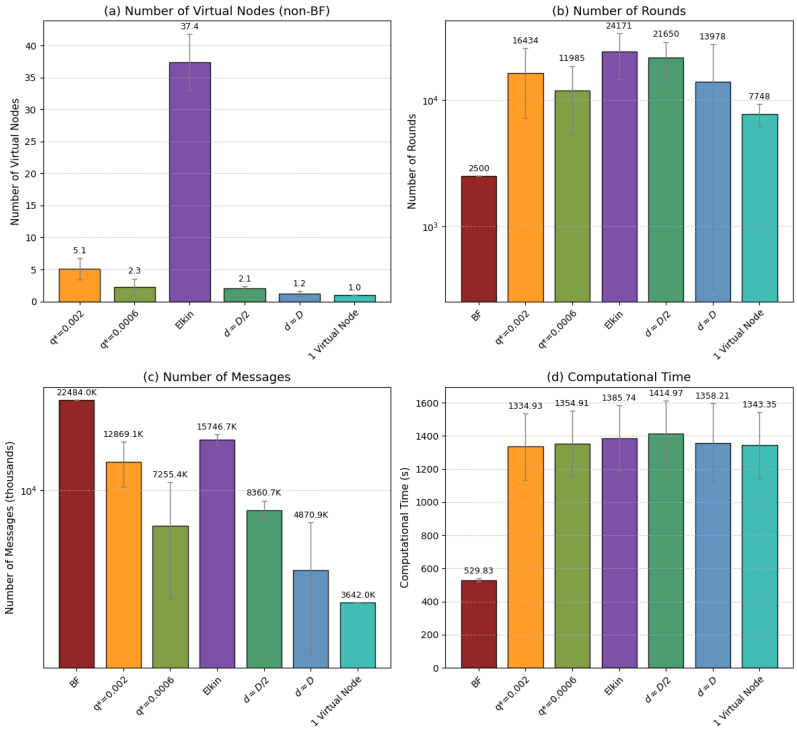


Figure 4.34: Performance for 5×500 grid graphs

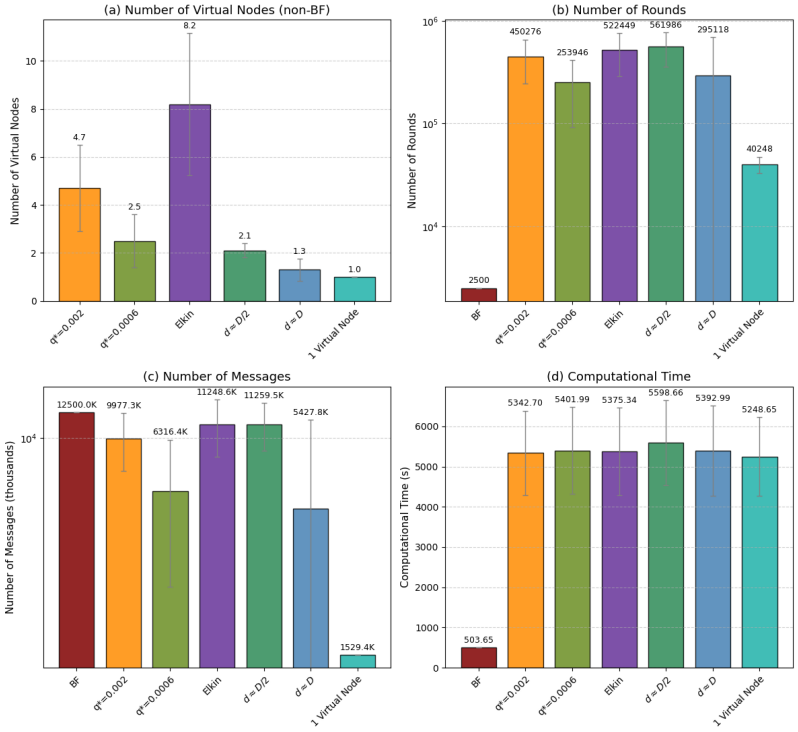


Figure 4.35: Performance for 1×2500 grid graphs

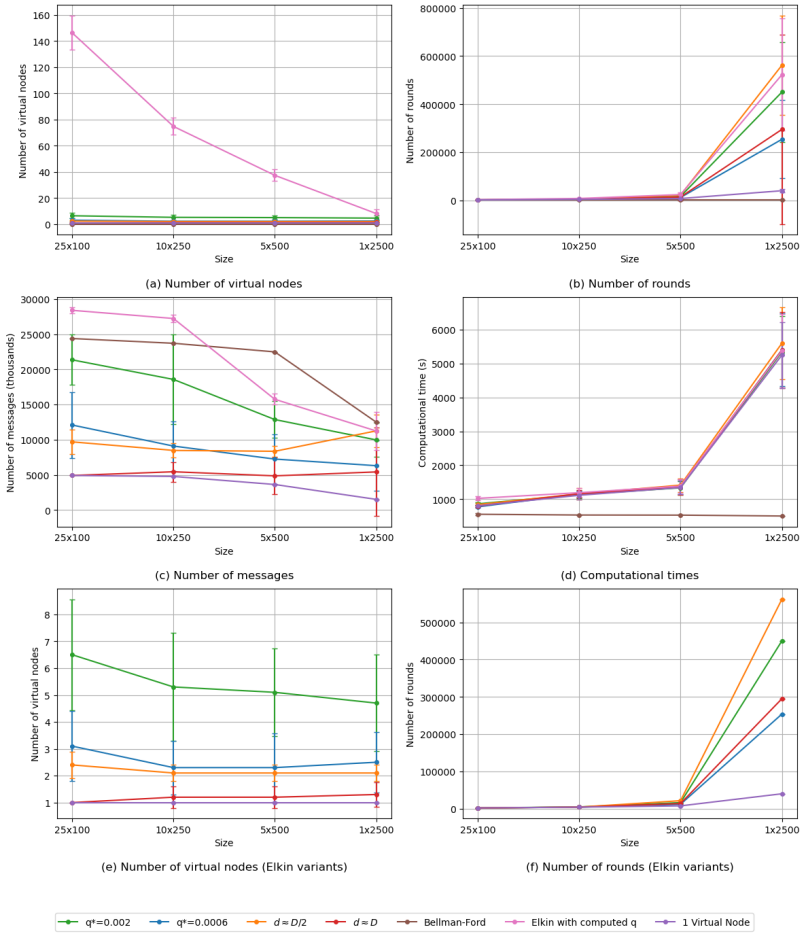


Figure 4.36: Performance for all algorithms in grid graphs

Figures 4.32 to 4.35 show the performance of all algorithms on the 25×100, 10×250, 5×500 and 1×2500 grid graphs. Figure 4.31 from Section 4.5 shows the performance on 50×50 square grids. We see that on the 50×50 and 25×100 grid graphs, Elkin's algorithm as well as all variants, succeed in computing the shortest paths, and all considered variants require fewer rounds than Bellman-Ford. However, on the remaining grid graphs ($10 \times 250, 5 \times 500, 1 \times 2500$) Elkin's algorithm and the considered variants not only do not obtain the correct results, they require many more rounds than Bellman-Ford, with the difference increasing with the diameter of the graph. For the 1×2500 line graph, Elkin's variants require about 200 times the number of rounds of Bellman-Ford. Indeed, the theoretical analysis of Elkin's algorithm gives a super-linear number of rounds when the diameter is $\Theta(n)$. The performance of all algorithms on all grid graphs is summarized in Figure 4.36.

4.7 Discussion

In summary, our experiments demonstrate that specific Elkin's algorithm configurations outperform Bellman-Ford in terms of the number of messages across all tested graph types. In terms of the number of rounds and computation time, the Elkin variants perform better for very large graphs or very dense graphs. However, for graphs with fewer than 1000 nodes that are sparse, Bellman-Ford works with fewer rounds and takes less time. For very dense graphs ($p > 0.2$), our optimised version of the Elkin algorithm outperforms Bellman-Ford.

Regarding the parameter selection, optimizing the balance between time, messages, and rounds generally requires selecting parameters that result in a small number of virtual nodes. This translates to using a small q^* in Elkin-R and choosing d close to the graph's hop diameter in Elkin-D, as this limits the number of virtual nodes selected.

Chapter 5

Conclusion & Discussion

This thesis addressed the distributed single-source shortest path problem by focusing on the theoretically efficient Elkin's algorithm [14], which offers a sub-linear time complexity advantage over the standard $O(n)$ Distributed Bellman-Ford. Our contributions centered on Elkin's approach, beginning with the *correction* of a small error in the original algorithm. We then presented an *improved implementation* utilizing techniques like *pipelined upcast-broadcast* tailored for the synchronous model. Finally, we expanded upon the original algorithm by introducing two novel variants, *Elkin-R* (using *probabilistic* virtual node selection) and *Elkin-D* (using *distance-based (hop count)* selection), to explore alternative strategies for potentially enhancing performance and accuracy in distributed environments.

Our simulation implemented the distributed algorithm using a direct thread-per-node mapping in *Java*. This approach offered the most practical means of evaluating performance for large networks; however, it has the disadvantage that the clock time is not a true reflection of computational time for a true distributed implementation. A comprehensive *performance evaluation* rigorously compared the improved Elkin algorithm and our newly proposed variants against Bellman-Ford across key metrics like rounds, messages, and clock time. Our study demonstrates that, depending on parameter settings and graph characteristics, certain Elkin variants outperform Bellman-Ford. Notably, message complexity improvements were observed across all tested graphs, while improvements in the number of rounds and computational time are achievable for larger or denser graphs.

Regarding Elkin's original algorithm, our conclusion is that it would outperform Bellman-Ford's algorithm only on very large graphs or very dense graphs. Indeed, the only sparse graphs that we saw Elkin's algorithm improving over Bellman-Ford's algorithm were in square grid graphs with 10000 nodes. However, real-life telecommunication networks are typically sparse and are not yet that large. One possible application

area might be social network graphs, which are indeed quite large and also have higher node degrees.

An interesting result from our experiments is that the best-performing variants have very few virtual nodes, sometimes even one virtual node, the source itself. Our validation showed that exact shortest paths were always computed when the diameter of the graph was small, that is, $o(n)$, though this was not the case for high-diameter graphs. Our results suggest that in practical situations where the maximum length of shortest paths is likely sub-linear in the number of nodes, using very few virtual nodes would give big performance improvements, and still succeed in finding exact shortest paths.

Future research directions include several key areas. First, our results on computational time, though not an accurate indicator of the time required in a true distributed implementation, indicate that the time needed for the k-shortcut hopset is the true bottleneck of the Elkin algorithm as well as the variants. More investigation is required to achieve efficiency in the implementation of this part of the algorithm. Secondly, in the spirit of Elkin-D, it would be interesting to verify if the exact placement of virtual nodes makes a difference to the efficiency of the algorithm.

Adapting and evaluating the improved Elkin implementation and its variants (Elkin-R, Elkin-D) in a fully asynchronous distributed model is an important future direction. This is more realistic for many real-world networks and introduces challenges like variable message delays and potentially different termination detection mechanisms. Secondly, we can extend the algorithms to handle dynamic graph changes (edge/node additions/deletions) efficiently. Furthermore, we can implement and test the algorithms on large-scale distributed simulators or actual hardware testbeds to validate performance beyond single-machine simulations and explore real-world communication bottlenecks.

Appendix A

Appendix

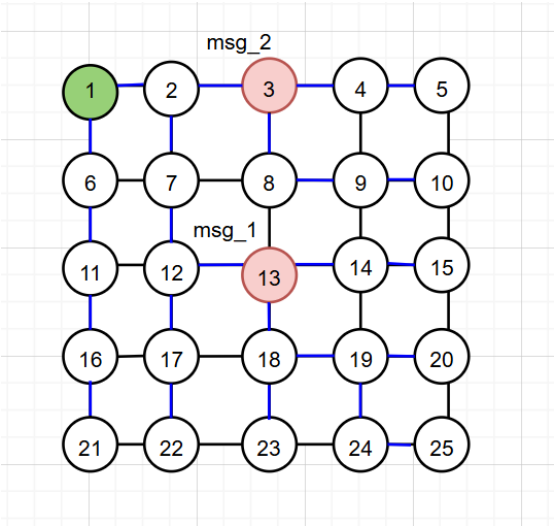
A.1 Pipelined Upcast-Broadcast example

Algorithm initial configuration:

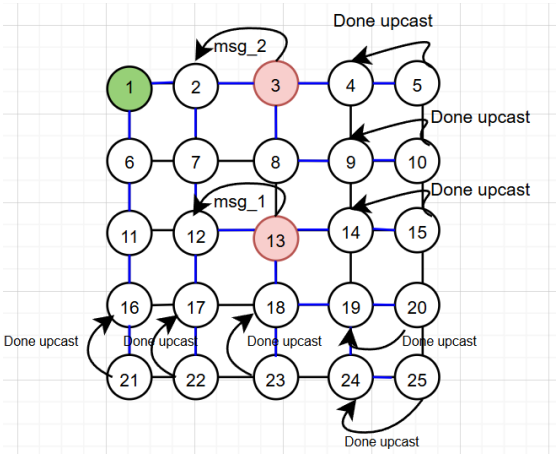
The following describes the starting state for the algorithm steps shown in the accompanying diagrams. *(Notation: msg_1 represents Message 1, msg_2 represents Message 2.)*

- **Source node:** The process begins with Node 1 designated as the source s.
- **Message placement:**
  - Message 2 (msg_2) originates at Node 3.
  - Message 1 (msg_1) originates at Node 13.

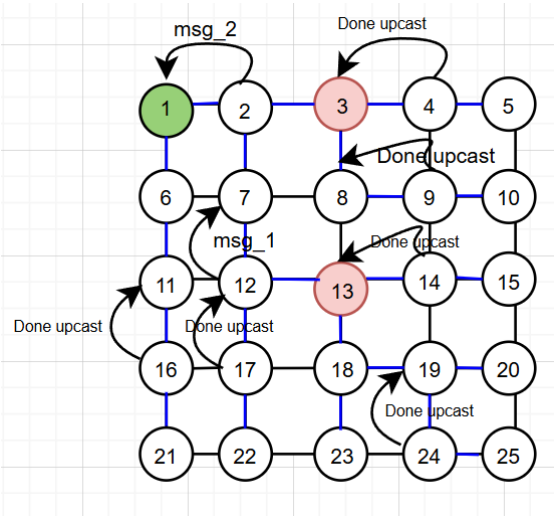
  (Nodes 3 and 13 are virtual nodes.)
- **Underlying structure:** The blue lines highlight the edges forming a Breadth-First Search (BFS) tree computed from the root (Node 1).



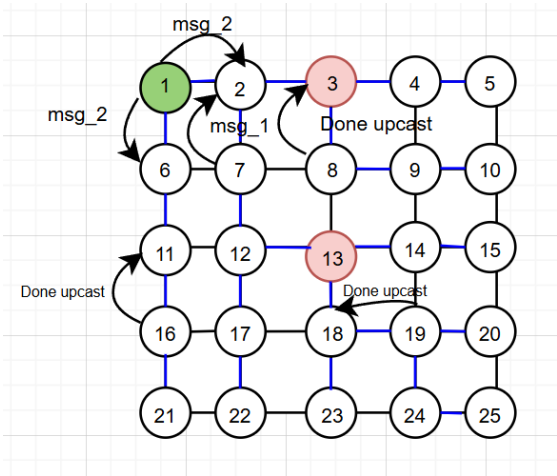
(a) Step 1



(b) Step 2

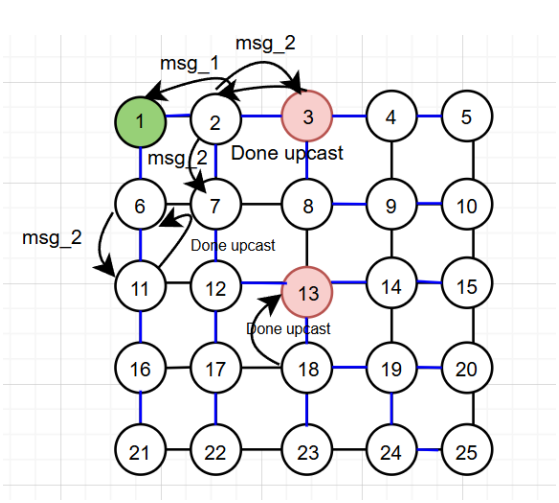


(c) Step 3

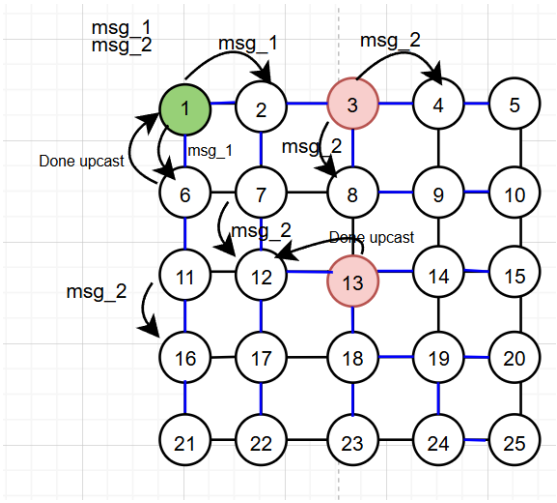


(d) Step 4

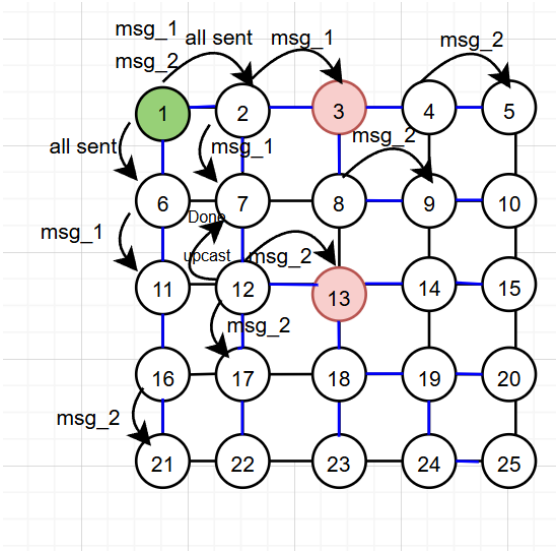
Figure A.1: *Upcast-Broadcast* process steps 1 through 4



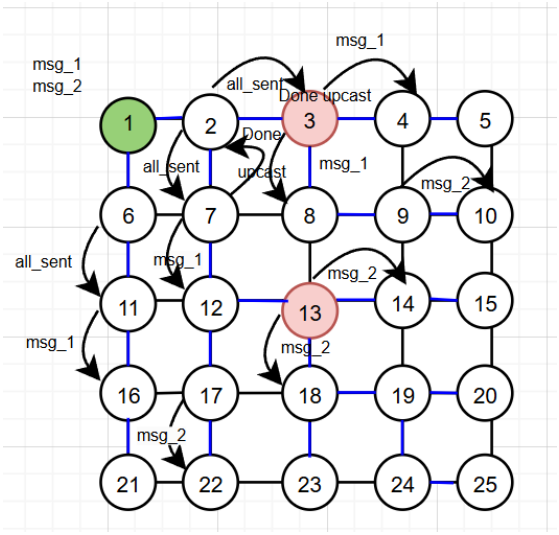
(a) Step 5



(b) Step 6

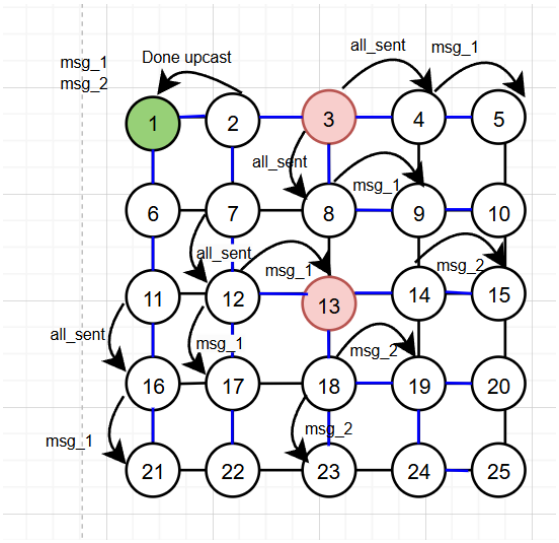


(c) Step 7

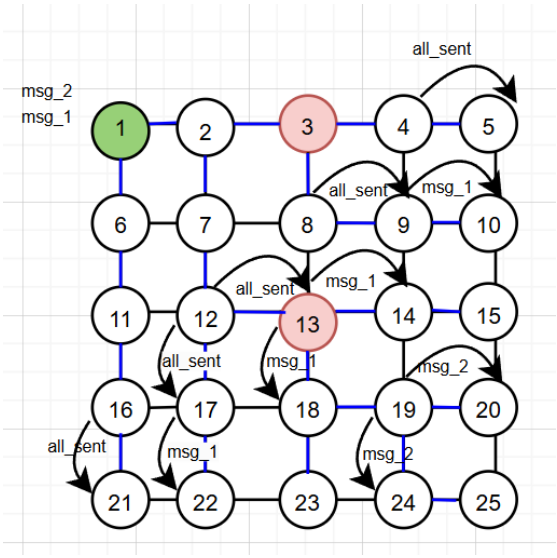


(d) Step 8

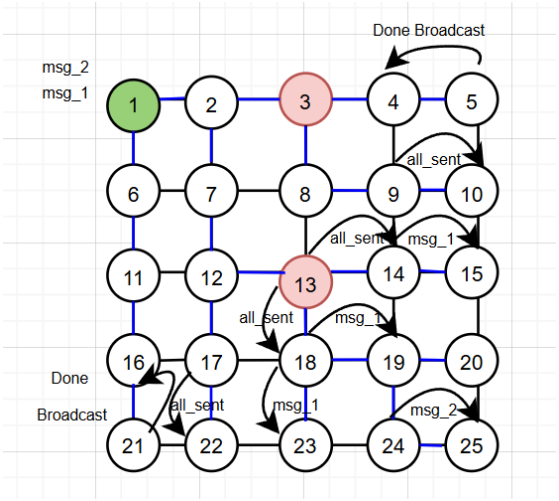
Figure A.2: *Upcast-Broadcast* process steps 5 through 8



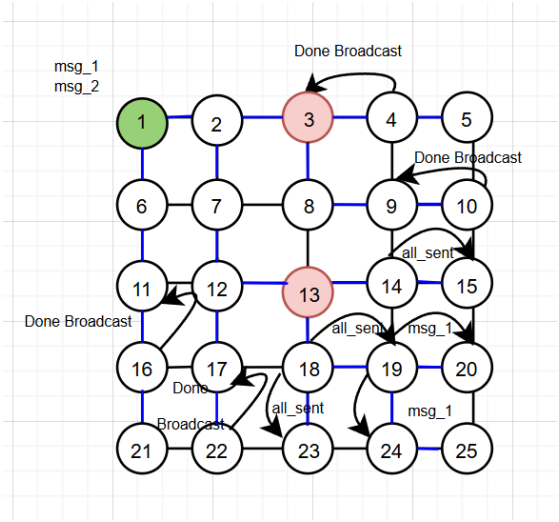
(a) Step 9



(b) Step 10

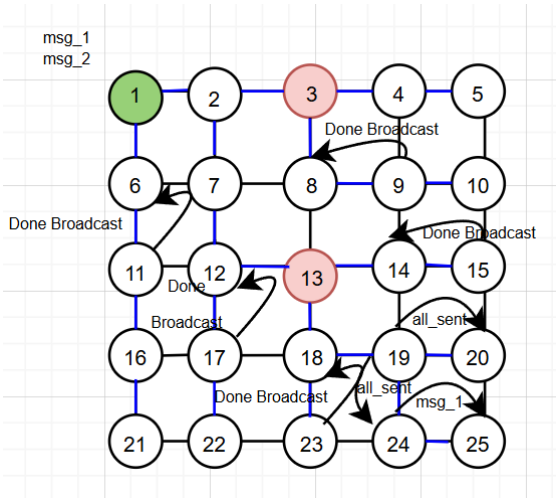


(c) Step 11

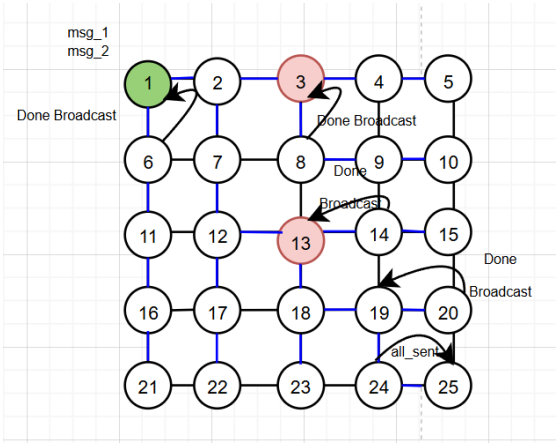


(d) Step 12

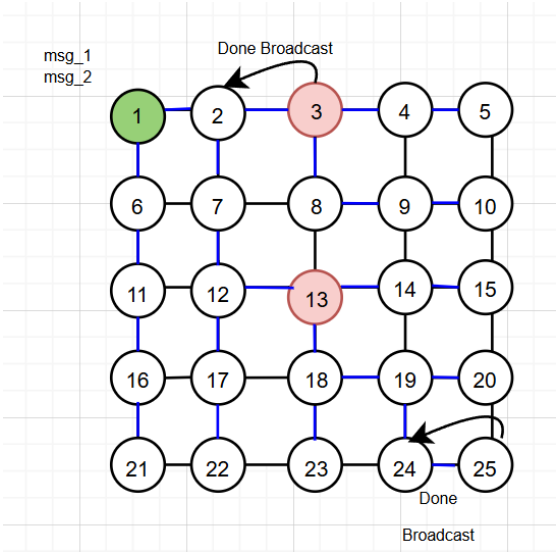
Figure A.3: *Upcast-Broadcast* process steps 9 through 12



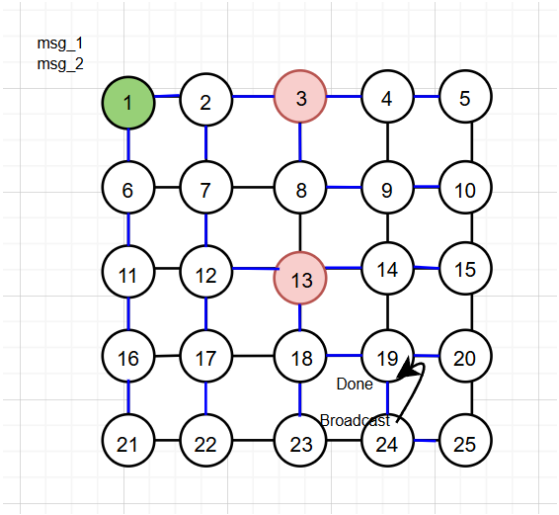
(a) Step 13



(b) Step 14

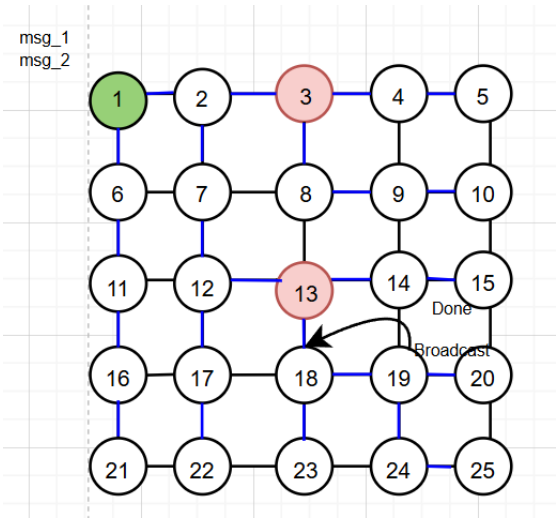


(c) Step 15

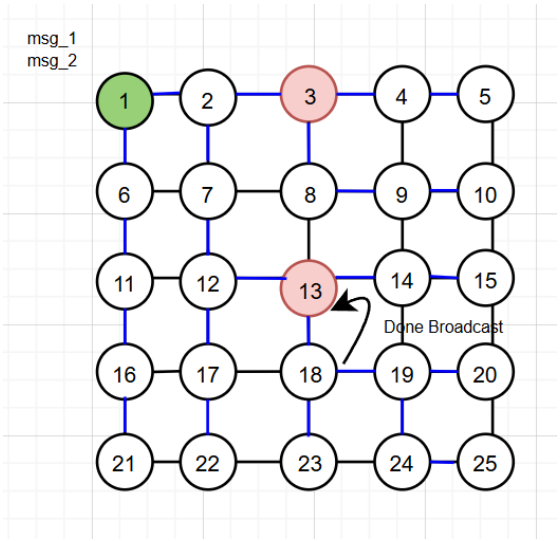


(d) Step 16

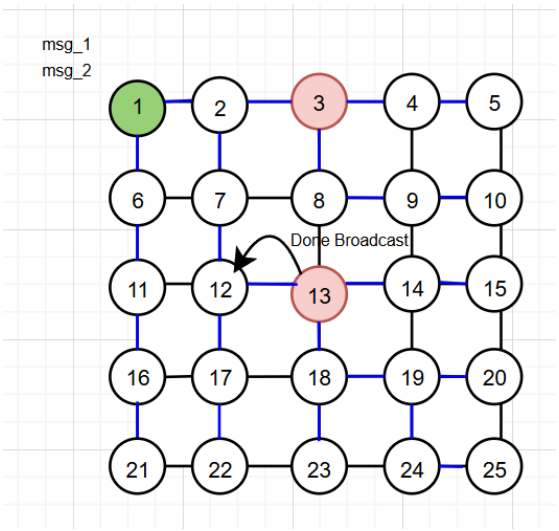
Figure A.4: *Upcast-Broadcast* process steps 13 through 16



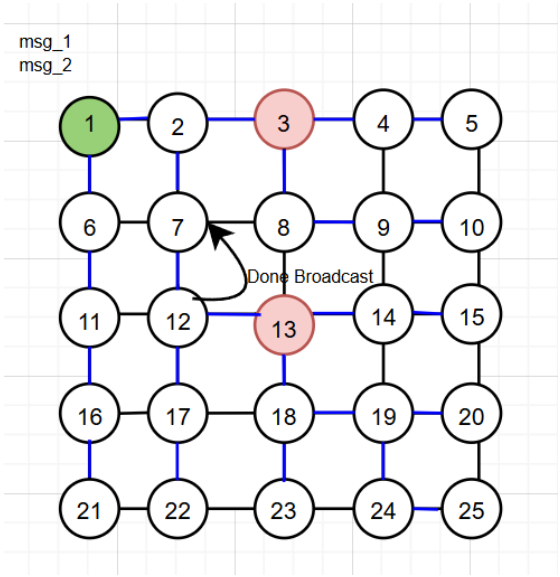
(a) Step 17



(b) Step 18

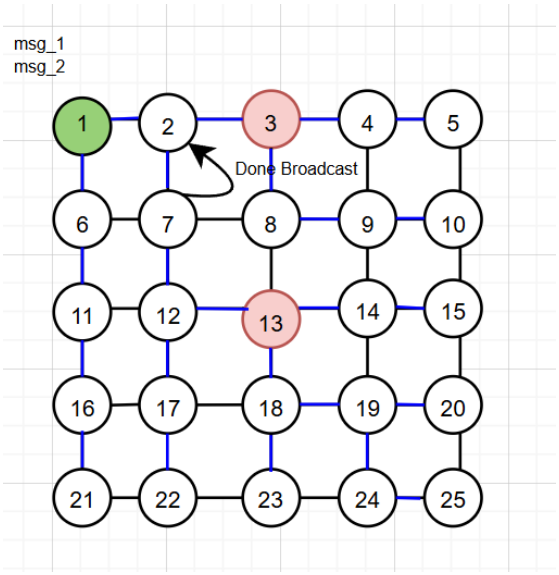


(c) Step 19

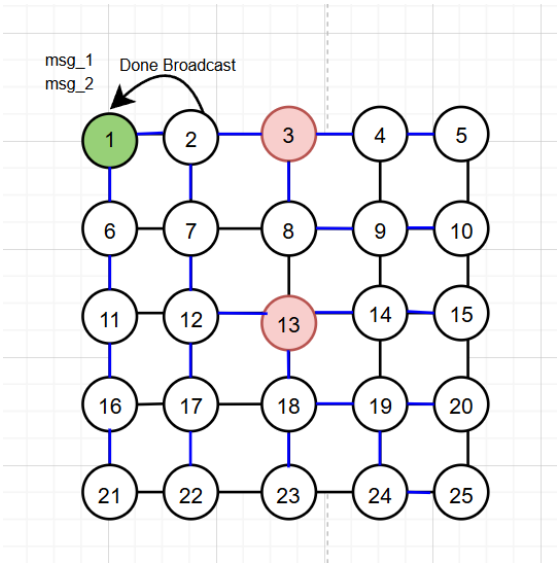


(d) Step 20

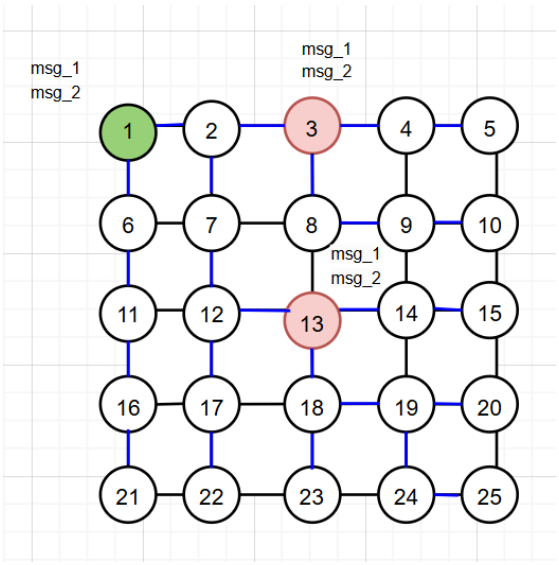
Figure A.5: *Upcast-Broadcast* process steps 17 through 20



(a) Step 21



(b) Step 22



(c) Step 23

Figure A.6: *Upcast-Broadcast* process steps 21 through 23

Bibliography

[1] U. Agarwal and V. Ramachandran. Distributed weighted all pairs shortest paths through pipelining. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 23–32, 2019.

[2] U. Agarwal and V. Ramachandran. Faster deterministic all pairs shortest paths in congest model. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '20, page 11–21, New York, NY, USA, 2020. Association for Computing Machinery.

[3] U. Agarwal, V. Ramachandran, V. King, and M. Pontecorvi. A deterministic distributed algorithm for exact weighted all-pairs shortest paths in õ (n 3/2) rounds. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 199–205, 2018.

[4] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, 1993.

[5] M. Bastian, S. Heymann, and M. Jacomy. Gephi: An open source software for exploring and manipulating networks. In *International AAAI Conference on Weblogs and Social Media*, 2009.

[6] R. Becker, S. Forster, A. Karrenbauer, and C. Lenzen. Near-optimal approximate shortest paths and transshipment in distributed and streaming models. *arXiv preprint arXiv:1607.05127*, 2016.

[7] R. Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[8] A. Bernstein and D. Nanongkai. Distributed exact weighted all-pairs shortest paths in near-linear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, page 334–342, New York, NY, USA, 2019. Association for Computing Machinery.

[9] R. Botez, A.-G. Pasca, A.-T. Sferle, I.-A. Ivanciu, and V. Dobrota. Efficient network slicing with sdn and heuristic algorithm for low latency services in 5g/b5g networks. *Sensors*, 23(13), 2023.

[10] A. Buzachis, A. Celesti, A. Galletta, J. Wan, and M. Fazio. Evaluating an application aware distributed dijkstra shortest path algorithm in hybrid cloud/edge environments. *IEEE Transactions on Sustainable Computing*, 7(2):289–298, 2022.

[11] S. Chechik and D. Mukhtar. Single-source shortest paths in the CONGEST model with improved bounds. *Distributed Computing*, 35(4):357–374, 8 2022.

[12] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[13] M. Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, page 331–340, New York, NY, USA, 2004. Association for Computing Machinery.

[14] M. Elkin. Distributed exact shortest paths in sublinear time. *Journal of the ACM (JACM)*, 67(3):1 – 36, May 2020.

[15] M. Elkin and O. Neiman. Hopsets with constant hopbound, and applications to approximate shortest paths. *SIAM Journal on Computing*, 48(4):1436–1480, 2019.

[16] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345 – 345, 1962.

[17] S. Forster and D. Nanongkai. A faster distributed single-source shortest paths algorithm. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 686–697, 2018.

[18] M. Ghaffari and J. Li. Improved distributed algorithms for exact shortest paths. In *Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, page 431–444, 2018.

[19] M. Ghaffari and A. Trygub. A near-optimal low-energy deterministic distributed sssp with ramifications on congestion and apsp. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, PODC '24, page 401–411, New York, NY, USA, 2024. Association for Computing Machinery.

[20] S. Gupta, A. Kosowski, and L. Viennot. Exploiting hopsets: Improved distance oracles for graphs of constant highway dimension and beyond. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 1 — 19, 2019.

[21] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In G. Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, 8 2008.

[22] M. Henzinger, S. Krinninger, and D. Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 489–498, 2016.

[23] C.-C. Huang, D. Nanongkai, and T. Saranurak. Distributed exact weighted all-pairs shortest paths in o(n5/4) rounds. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 168–179, 2017.

[24] K. R. Hutson, T. L. Schlosser, and D. R. Shier. On the distributed bellman-ford algorithm and the looping problem. *Informs journal on computing*, 19(4):542–551, 2007.

[25] K. Karur, N. Sharma, C. Dharmatti, and J. E. Siegel. A survey of path planning algorithms for mobile robots. *Vehicles*, 3(3):448–468, 2021.

[26] S. S. Khopkar, R. Nagi, and A. G. Nikolaev. An efficient map-reduce algorithm for the incremental computation of all-pairs shortest paths in social networks. In *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 1144–1148. IEEE, 2012.

[27] C. Lenzen and B. Patt-Shamir. Fast routing table construction using small messages. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 381–390, 2013.

[28] G. Malkin. Rfc2453: Rip version 2, 1998.

[29] L. Meng, Y. Shao, L. Yuan, L. Lai, P. Cheng, X. Li, W. Yu, W. Zhang, X. Lin, and J. Zhou. A survey of distributed graph algorithms on massive graphs. *ACM Comput. Surv.*, 57(2):1 – 39, Oct. 2024.

[30] J. T. Moy. *OSPF: anatomy of an Internet routing protocol*. Addison-Wesley Professional, 1998.

[31] D. Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 565–573, 2014.

[32] Oracle Corporation. Java Platform, Standard Edition 8 API Specification - Class Thread. https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html, 2014. Accessed: 2023-10-27.

[33] D. Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.

[34] D. Peleg and V. Rubinovich. A near-tight lower bound on the time complexity of distributed mst construction. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 253–261, 1999.

[35] V. Rozhoň, C. Grunau, B. Haeupler, G. Zuzic, and J. Li. Undirected (1+ε)-shortest paths via minor-aggregates: near-optimal deterministic parallel and distributed algorithms. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2022, page 478–487, New York, NY, USA, 2022. Association for Computing Machinery.

[36] SchedMD LLC. Slurm Workload Manager. https://slurm.schedmd.com/, 2024.

[37] J. Simons. CORONET networks. http://www.monarchna.com/topology.html. Monarch Network Architects.

[38] P. Singal and R. Chhillar. Dijkstra shortest path algorithm using global position system. *International Journal of Computer Applications*, 101(6), 2014.

[39] J. Suárez-Varela et al. The graph neural networking challenge: a worldwide competition for education in ai/ml for networks. *ACM SIGCOMM Computer Communication Review*, 51(3):9–16, 2021.

[40] The Linux Kernel Organization. The Linux Kernel Archives. https://www.kernel.org/, 2024.

[41] J. Velinska, I. Mishkovski, and M. Mirchev. Routing, modulation and spectrum allocation in elastic optical networks. *IEEE Transactions on Telecommunications Forum*, 26:1–4, 2018.

[42] Y. Zeng, C. Ma, and Y. Fang. Distributed shortest distance labeling on large-scale graphs. *Proceedings of the VLDB Endowment*, 17(10):2641–2653, Aug. 2024.