UNIVERSITY OF CALIFORNIA,
IRVINE

Efficient Algorithms for Road Networks and Noisy Sorting: an Experimental and Theoretical Perspective

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Evrim Ozel

Dissertation Committee:
Distinguished Professor Michael T. Goodrich, Chair
Professor Michael B. Dillencourt
Distinguished Professor David Eppstein

2024

DEDICATION

To my parents.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

My Ph.D. journey would not have been possible without the support of my colleagues, friends and family. I would first like to thank my advisor, Michael Goodrich, for his generosity in investing his time in my growth as a researcher. His invaluable support and guidance throughout this journey helped me push through many roadblocks in my research. I would also like to thank all the UCI faculty members I've worked alongside for both research and teaching. In particular, I thank Michael Dillencourt and David Eppstein for serving in my defense committee, and for providing feedback and guidance for my research and dissertation. I would like to thank Michael Shindler, who I worked with for my first teaching assistant appointment at UCI, for providing encouragement and feedback for my teaching, as well as for serving in my advancement committee. I also thank Asaf Ferber for serving in my advancement committee.

I'm very lucky to have worked alongside some excellent collaborators throughout this journey; in particular I'd like to thank my co-authors Ramtin Afshar, Gill Barequet, Michael Dillencourt, Shion Fukuzawa, Ofek Gila, Michael Goodrich, David Mount and Martha Osegueda. I would also like to thank David Eppstein, Riko Jacob and Ulrich Meyer for helpful discussions regarding parts of this work. I am thankful to all of my graduate student friends at UCI I've gotten to know over the years; their friendship helped me tremendously throughout my PhD. Last, but not least, I would like to thank my parents, Selda and Melih, for their unconditional love and support throughout this journey.

I am grateful to the Donald Bren School of Information and Computer Science for their funding support. I also thank ACM SIGSPATIAL for awarding our paper "Modeling the Small-World Phenomenon with Road Networks", which is part of this thesis, with the Best Paper Runner-up award in 2022. This work was supported in part by NSF grants 1815073 and 2212129. Chapter 5 of this dissertation is reproduced with permission from Springer Nature as it appears in [65], and the co-author listed in this publication is Michael Goodrich.

VITA

Evrim Ozel

EDUCATION

Doctor of Philosophy in Computer Science — **2024**
University of California, Irvine — *Irvine, CA, USA*

Master of Science in Computer Science — **2021**
University of California, Irvine — *Irvine, CA, USA*

Bachelor of Science in Computer Engineering — **2019**
Middle East Technical University — *Ankara, Turkey*

EXPERIENCE

Teaching Assistant — **2020–2024**
University of California, Irvine — *Irvine, CA*

Applied Research Intern — **2023**
LinkedIn — *Sunnyvale, CA*

Software Engineer Intern — **2022**
Intel — *Hillsboro, OR*

Reader — **2019**
University of California, Irvine — *Irvine, CA*

PUBLICATIONS

Highway Preferential Attachment Models for Geographic Routing **2023**
Ofek Gila, Micheal T. Goodrich, Evrim Ozel.
International Conference on Combinatorial Optimization and Applications (COCOA)

External-Memory Sorting with Comparison Errors **2023**
Michael T. Goodrich, Evrim Ozel.
Algorithms and Data Structures Symposium (WADS)

Noisy Sorting Without Searching: Data Oblivious Sorting with Comparison Errors **2023**
Ramtin Afshar, Michael Dillencourt, Michael T. Goodrich, Evrim Ozel.
International Symposium on Experimental Algorithms (SEA)

Modeling the Small-World Phenomenon with Road Networks **2022**
Michael T. Goodrich, Evrim Ozel.
International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL)

Diamonds are Forever in the Blockchain: Geometric Polyhedral Point-Set Pattern Matching **2022**
Gill Barequet, Shion Fukuzawa, Michael T. Goodrich, David M. Mount, Martha C. Osegueda, Evrim Ozel.
Canadian Conference on Computational Geometry (CCCG)

Efficient Exact Learning Algorithms for Road Networks and Other Graphs with Bounded Clustering Degrees **2022**
Ramtin Afshar, Michael T. Goodrich, Evrim Ozel.
International Symposium on Experimental Algorithms (SEA)

ABSTRACT OF THE DISSERTATION

Efficient Algorithms for Road Networks and Noisy Sorting: an Experimental and Theoretical Perspective

By

Evrim Ozel

Doctor of Philosophy in Computer Science

University of California, Irvine, 2024

Distinguished Professor Michael T. Goodrich, Chair

Experimental algorithmics, also referred to as algorithm engineering, is the principled approach of using empirical methods to complement traditional theoretical methods, both of which provide valuable insights for the analysis of algorithms. In this dissertation, we study various algorithmic problems for road networks and noisy sorting, analyzing them from both an experimental and theoretical perspective.

We first study the problem of exact learning for road networks, and introduce an efficient randomized algorithm using simple distance queries, which can find missing roads and improve the quality of routing services. We provide a partial theoretical analysis based on a *cluster degree* parameter, d_{max}, then empirically show that this parameter is small for road networks by evaluating our algorithm on road network data for the U.S. and 4 European countries of various sizes. This analysis provides experimental evidence that our algorithm issues a quasilinear number of queries in expectation for road networks and similar graphs. We also study the small-world navigability of the U.S. road network, inspired by the famous experiments done by Stanley Milgram which gave rise to the "six degrees of separation" expression. We introduce the Neighborhood Preferential Attachment Model, and perform extensive experiments with this model on U.S. road networks to show that our model out-

performs other small-world models in terms of the average hop length, while having a more realistic scale-free degree distribution.

We then study the problem of sorting n comparable distinct elements, subject to noisy comparison errors, such that the comparison of two elements returns the wrong answer according to a fixed probability $p_e > 1/2$. We provide new methods for sorting with comparison errors that are data oblivious while avoiding the use of noisy binary search methods. We then experimentally compare our algorithms and other sorting algorithms. Lastly, we study the noisy sorting problem in an external-memory setting, providing new efficient methods that are in the external-memory model for sorting with comparison errors. Our algorithms achieve an optimal number of I/Os, in both cache-aware and cache-oblivious settings.

Chapter 1

Introduction

1.1 Road Networks

The development of road networks goes back to antiquity, when road transportation was used for carrying goods over tracks via pack animals. Since then, increased urbanization and the widespread usage of the automobile has led to a rapid increase of the volume of such networks. Today, we rely on road transportation networks more than ever. We use navigation and mapping software (such as Google Maps, Apple Maps, or OpenStreetMap) in our daily lives. This has brought an increased interest in the development of algorithms for road networks.

In Chapters 2 and 3, we present efficient algorithms for computational tasks that are motivated through road networks, and analyze these algorithms from theoretical and experimental perspectives. Formally, we define a road network as an undirected and connected graph, where each edge corresponds to a segment in the road network, and each vertex corresponds to either a junction between two road segments, or a terminus. Depending on the application, we can consider road networks as weighted or unweighted graphs. For weighted

road networks, edge costs are typically either the real-world distance between two vertices, or the travel time (which takes into account the average speed through a road). Another well-known characterization of road networks is having a bounded maximum degree, which is derived by physical and geographic limitations when constructing roads.

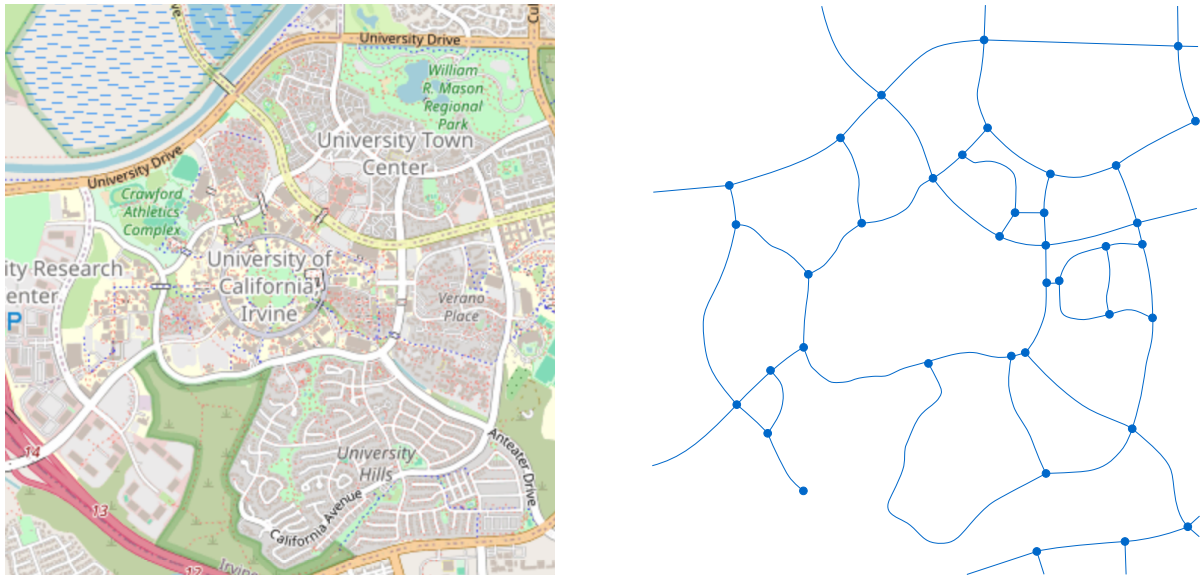


Figure 1.1: Road map of UC Irvine and surrounding areas (left), and a partial unweighted graph representation of the this network (right). Left image is from OpenStreetMap and is licensed under the Open Data Commons Open Database License (ODbL) by the OpenStreetMap Foundation (OSMF).

1.1.1 Exact Learning of Road Networks

The first problem we study is the exact learning of graphs. In this problem, we are given the vertices V of an unknown graph $G = (V, E)$, as well as access to an oracle, which is able to answer queries involving the vertices in the graph. The goal is to reconstruct the graph by learning the set of edges E of the graph. While there are several querying models that can be considered for this problem, we focus on distance queries, where the oracle is given a pair of vertices u, v, and returns the number of edges on the shortest path between them in G. In Chapter 2, we specifically study reconstruction algorithms for *road networks*, which is

motivated by the importance of the completeness of road network datasets used in various routing services [9].

Prior Work and Motivation. The general problem of exact learning falls under the umbrella of *computational learning theory* [74, 109, 60, 103]. The concept of exact learning itself goes back to work done by Angluin [14, 13], who studied the problem of learning regular language sets by introducing a two-party algorithm involving a teacher and learner, called the L* algorithm. This involves the learner issuing a sequence of queries to the teacher, such as membership queries (whether a given string belongs to the unknown regular set), or equivalence queries (whether a predicted description of the regular set is accurate or not). The learner, using these responses, updates its own prediction of the regular set iteratively.

Exact learning was later applied to graphs: Griebinski and Kucherov [67, 68] study the problem of reconstructing an unknown Hamiltonian cycle in a complete graph by considering different querying models: the *multi-vertex model*, which issues queries on whether the complete graph on a set of vertices intersects with the unknown Hamiltonian cycle or not; the *quantitative multi-vertex model*, where the number of edges in the intersection is queried instead; and two additional variants where the set of vertices issued in each query is constrained by a predetermined parameter. Alon, Beigel, Kasif, Rudich, and Sudakov [12] studied the problem of reconstructing matchings by issuing queries on whether a given induced subgraph shares any edges with the unknown matching. Angluin and Chen [16] extend this problem to general graphs using the same type of queries (referred to as *edge detection queries*). The problem of graph reconstruction has since been studied extensively, with applications in a variety of fields such as computer security, molecular biology, genome sequencing, and internet network mapping; and through different query models, such as distance queries (the distance of the shortest path between two vertices), shortest path queries (the vertices that appear in the shortest path between two vertices) and betweenness

queries (whether a given vertex exists on the shortest path of two given vertices), e.g. see [1, 6, 11, 22, 96, 113, 5, 7, 8, 16, 32, 36, 77, 98].

More recently, Kannan, Mathieu and Zhou [72] introduced a reconstruction algorithm for connected, unweighted graphs using $O(\Delta^3 n^{3/2} \operatorname{polylog}(n))$ distance queries in expectation, where Δ denotes the maximum degree of the graph, and also raised the open question of whether we can achieve an algorithm that only uses $O(n \operatorname{polylog}(n))$ distance queries in expectation for bounded degree graphs. Mathieu and Zhou [87] later provided a partial answer for that open question by providing an algorithm that uses $O(n \operatorname{polylog}(n))$ distance queries in expectation for random Δ-regular graphs. For general graphs of bounded degree, their algorithm uses $O(n^{5/3} \operatorname{polylog}(n))$ distance queries in expectation.

Results. In Chapter 2 present a randomized incremental algorithm for reconstructing road networks (and similar graphs). We first theoretically analyze our algorithm and prove an upper bound of $O(d_{max}^2 n \log n)$ expected queries, where d_{max} is an upper bound on the degrees of vertex clusters defined during our algorithm. We then conduct an experimental analysis of our algorithm using real-world road network datasets, and show empirically that d_{max} scales logarithmically with the size of the input graph, which, combined with our theoretical analysis, results in an empirical query complexity of O(n polylog(n)), providing an empirical answer to an open question raised by [72]. We further experimentally compare our algorithm with existing reconstruction algorithms, and show that our algorithm issues significantly fewer queries.

1.1.2 The Small-World Phenomenon in Road Networks

Inspired by the famous small-world experiments conducted by Stanley Milgram in the 1960s [88], we study the problem of modeling the small-world phenomenon, which is the idea that all people are connected through a short chain of acquaintances that can be used to route

messages, through random graph models. This topic has a rich history, with several published papers proposing mathematical models to explain the phenomenon. However, most prior work in this area, such as the well-known *preferential attachment* model by Barabási and Albert [18], focus on the "short chain" of acquaintances rather than their ability to efficiently route messages. A notable exception is the well-known model by Jon Kleinberg [79], which shows that it is possible for participants to route a message in $O(\log^2 n)$ hops using a random graph model based on an $n \times n$ grid and using a simple greedy routing strategy. Although this model is intriguing, it does not take into account the road network of the United States used in the original Milgram experiments and its $O(\log^2 n)$ number of hops for messages is actually quite far from the average of six hops for successful messages observed by Milgram in his experiments, which gave rise to the "six-degrees-of-separation" expression. Motivated by the geographic nature of the small-world phenomenon, in Chapter 3 we present a new small-world model that is based on the U.S. road network.

Prior Work and Motivation.

The small-world phenomenon was popularized through experiments done in the 1960s by the social psychologist Stanley Milgram [88, 107]. The experiments involved randomly selecting individuals in Omaha, Nebraska and Wichita, Kansas, and asking them to send a postcard to a target person who lived in Boston, Massachusetts. The participants were given instructions asking them to either mail the postcard directly if they knew the target person, or to forward the postcard to someone who they think would be more likely to know the target person. The surprising result from these experiments was that among the postcards that made it to the target people, the median number of hops the postcards took was only about 6. This popularized the idea that most people can be connected through a small chain of acquaintances.

The small-world phenomenon has since had a large influence on a wide variety of fields, such as rumor spreading, epidemics, electronic circuits, wireless networks, the World Wide Web,

network neuroscience, and biological networks. For a survey on the small-world phenomenon and its applications, we refer the reader to [114]. This has also led to an increased interest in mathematical models that attempt to explain why the small-world phenomenon occurs, e.g., see [41].

One well-known mathematical model for social networks is the *preferential attachment* model, which is a random graph model for non-geographic social networks, such as the World Wide Web. The history of this model goes back 100 years e.g., see [120, 37, 101], and was popularized and formalized by Barabási and Albert [18], who also coined the term *scale-free*, which describes networks where the fraction of vertices with degree d follows a power law, $d^{-\alpha}$, where $\alpha > 1$. The construction of a preferential attachment network follows what is often called a "rich-get-richer" process, which involves starting constant-sized "seed" graph, and iteratively adding vertices to the graph one by one, where each new vertex is connected to a fixed number of other vertices randomly, where the probability of a vertex receiving a connection is proportional to the number of connections it already has, e.g., see [20]. A rigorous analysis of this model, e.g. to determine the diameter and degree distribution, was later studied by Bollabas and Riordan [26].

Watts and Strogatz [115] introduced a network model for generating random graphs with small-world properties, such as a low average path length and high levels of clustering. However, unlike the Barabási-Albert model, this model does not result in a scale-free degree distribution. Roughly, the model involves constructing a "re-wired" ring lattice, by starting with a ring lattice where each vertex is connected to a fixed number of its closest neighbors, called the *local connections*, then reassigning some of edges to other vertices chosen uniformly at random, called *long-range connections*.

The paradigm introduced by Watts and Strogatz was then generalized by Kleinberg [79] to address some of its drawbacks. A notable difference of this model from previous ones was that Kleinberg's model focused not only on the existence of short paths within the social

network, but also on how people were able to *find* these paths using only knowledge of their own local connections. Moreover, unlike previous models, this model incorporated distances between people in the network as the deciding factor when adding random edges to the network, illustrating how a simple decentralized routing strategy can work in a geographic setting to efficiently route messages between pairs of points. Kleinberg's model is built on a two dimensional grid, where each grid point corresponds to a single person, and just like the Watts-Strogatz model, it utilizes two types of connections—local connections and long-range connections. The local connections of the network are made by connecting each grid point to every other grid point within a fixed lattice distance. The long-range connections are made by connecting each grid point to a fixed number of other grid points chosen randomly, such that the probability that two grid points are connected is inversely proportional to their lattice distance. Kleinberg showed that through a simple decentralized greedy algorithm, where each message holder forwards its message to an acquaintance that is closest to the target grid point, messages can be routed in $O(\log^2 n)$ hops in expectation.

Results. In Chapter 3, we introduce a new small-world model, the Neighborhood Preferential Attachment model [64], which combines elements from both Kleinberg's model and the Barabási-Albert model, such that long-range links are chosen according to both the degrees and (road-network) distances of vertices in the network. We empirically evaluate all three models by running a decentralized routing algorithm, where each vertex only has knowledge of its own neighbors, and find that our model outperforms both of these models in terms of the average hop length. Moreover, our experiments indicate that similar to the Barabási-Albert model, networks generated by our model are scale-free, which could be a more realistic representation of acquaintanceship links in the original small-world experiment.

1.2 Sorting with Noisy Comparison Errors

Sorting is a fundamental problem in computing. An important category of sorting algorithms is comparison-based sorting, where we have access to a primitive operation, COMPARE(a, b), that takes as input two comparable elements a, b and determines which of the two elements should appear first in a sorted ordering. In the classical setting, the task is to sort n distinct comparable elements using a series of COMPARE operations. In Chapters 4 and 5, we study the Noisy Sorting problem, where with some fixed probability $p_e < 1/2$, the comparison of two elements returns the wrong answer, and otherwise returns the correct answer. Noisy sorting algorithms with persistent errors are typically evaluated by the amount of *dislocation* that occurs for each element, which is the distance between its position in a given (current or output) array and its position in a sorted array. For sorting n elements, we can then evaluate the *maximum* and *total* dislocation of the output array.

Comparison errors can be categorized into *persistent errors*, where the result of a comparison of two given elements, x and y, always returns the same result, and *non-persistent errors*, where the probabilistic determination of correctness is determined independently for each comparison, even if it is for a pair of elements, (x, y), that were previously compared. Note that sorting with persistent comparison errors is a much more difficult problem than sorting with non-persistent comparison errors, as there is a trivial $O(n \log^2 n)$ time solution that sorts perfectly with high probability for the latter case, whereas for the former, there exists known lower bounds of $O(\log n)$ and $O(n)$ in expectation for the maximum and total dislocation respectively [55, 54, 56].

Prior Work and Motivation.

Problems involving probabilistic comparison errors goes back to a classic two-player game by Rényi [95] where player A is trying to guess something player B is thinking of by asking yes/no questions, and player B lies with some probability. A similar two-player game is proposed by

Ulam [108], where player A is trying to guess a number that player B is thinking of through yes/no questions, but with player B being allowed to lie up to a fixed number of times; for a survey, see [33, 92]. These ideas were later applied to the digital realm, with work being done on noisy boolean gates [93], and sorting networks with noisy comparators [118].

Feige, Raghavan, Peleg and Upfal [46] consider a non-persistent noisy comparison framework, where each comparison fails independently with a fixed probability $p < 1/2$, and provide tight bounds for a variety of algorithms, including sorting and searching. Karp and Kleinberg [73] study binary searching for a value $x \in [0, 1]$ in an array of biased coins ordered by their biases. There is also work done on a variant of this problem when the total number of faulty comparisons is bounded rather than having probabilistic noisy comparisons (similar to the constraint in Ulam's game [108]), see e.g. [75, 97].

Finocchi and Italiano [47] consider a different error framework, the *faulty-memory model*, where the contents of a memory location may get altered unexpectedly before or during an algorithm's runtime, such that unlike the noisy comparison framework, strategies based on comparison replications would not work. They achieve a time and space-optimal comparison-based sorting algorithm that is resilient to $O((n \log n)^{1/3})$ memory faults.

Braverman and Mossel [27] introduce the persistent-error model, where "resampling" comparisons made previously is not possible. In this framework, where comparisons are persistently wrong with a fixed probability, $p < 1/2 - \varepsilon$, they achieve a sorting algorithm that achieves optimal maximum and total dislocations, and runs in $O(n^{3+f(p)})$ time, where $f(p)$ is some function of p. Klein, Penninger, Sohler, and Woodruff [78] improve the running time to $O(n^2)$, but have $O(n \log n)$ total dislocation w.h.p. The running time for sorting optimally in the persistent-error model with respect to maximum and total dislocation was subsequently improved to $O(n^2)$, $O(n^{3/2})$, and ultimately to $O(n \log n)$, in a sequence of papers by Geissmann, Leucci, Liu, and Penna [55, 56, 54].

Motivation for sorting with comparison errors comes from real-world applications of sorting algorithms on large volumes of data, e.g. as part of A/B testing [116], where there is a risk of faults occurring while sorting. Another use case for noisy sorting is in applied cryptography, where cryptographic comparison protocols have some known probability of failure, see e.g. [84, 48, 117].

1.2.1 Data Oblivious Noisy Sorting

A sorting algorithm is data oblivious if its memory access pattern does not reveal any information about the data values being sorted. This property plays an important role in applications such as secure cloud computing and cryptographic protocols, where privacy-preserving algorithms can be beneficial. However, existing noisy sorting algorithms are unfortunately not data oblivious, as they all make use of a data-sensitive procedure called noisy binary search. In Chapter 4, we design efficient algorithms that are tolerant to noisy comparison errors while not relying on noisy binary search.

Results. In Chapter 4, we provide several algorithms for sorting an array of n comparable distinct elements subject to probabilistic comparison errors, with one of these algorithms being data oblivious, while avoiding the use of noisy binary search methods. We provide theoretical analyses for the time complexity and maximum dislocation of our algorithms, and prove that they run in quasilinear time and achieve an optimal maximum dislocation of $O(\log n)$. We also analyze our algorithms experimentally, and show that all of the data-oblivious algorithms we provide have maximum and total dislocations that are comparable to the optimal bounds of $O(\log n)$ and $O(n)$ respectively for the best algorithms that are not data oblivious.

1.2.2 External-Memory Noisy Sorting

In Chapter 5, we study noisy sorting algorithms for the external memory model [10], which is a model of computation consisting of a two-layer memory hierarchy: a fast cache of small size, and a disk that can store larger files but is slower. The running time of an external memory algorithm is defined as the number of memory transfers or input/output operations (I/Os) between the two memory layers, where each I/O retrieves a block of size B from external memory to internal memory. Efficient external memory algorithms then make use of *locality* in their memory access pattern, by processing data elements that are stored close to each other and thus minimizing the number of block transfers. External-memory noisy sorting algorithms are motivated by applications in real-world systems where there is a large volume of data to be sorted for which external-memory sorting can be beneficial. There are various existing algorithms that can be utilized for sorting or near-sorting elements subject to probabilistic comparison errors, but these algorithms do not translate into efficient external-memory algorithms, because they all make heavy use of noisy binary searching. The main disadvantage of relying on noisy binary search is that it is cache inefficient, in that it requires performing memory accesses for widely-distributed storage locations.

Results. In Chapter 5, we provide several algorithms for sorting an array of n comparable distinct elements subject to probabilistic comparison errors in cache-aware and cache-oblivious settings. We provide a theoretical analysis for both algorithms and show that they run in time $O(n \log^2 n)$ in internal memory, or in external memory with an optimal $O((n/B) \log_{M/B}(n/B))$ I/O's, and have a maximum dislocation of $O(\log n)$ w.h.p., where M denotes the size of internal memory and B denotes the size of a block of memory.

Chapter 2

Exact Learning of Road Networks

2.1 Introduction

We study the problem of reconstructing an undirected, unweighted and connected graph $G = (V, E)$, by taking as input its set of vertices V and issuing queries to a *distance oracle*, which takes as input a pair of vertices $u, v \in V$ and returns the number of edges on the shortest path between them. The goal is to learn the edges in E by using the results that are returned from these queries. In particular, we are concerned with reconstructing *road networks*, which have been characterized in numerous ways, e.g., see [34, 43, 44, 53]. As a starting point, we can view road networks as undirected, unweighted, and connected graphs with a constant maximum degree, where each vertex corresponds to a road junction or terminus, and each edge corresponds to road segments that connect two vertices. In this chapter, we present a randomized incremental algorithm for *exact learning* of road networks, where we assume the existence of a distance oracle that responds to distance queries.

Even though our algorithm only works with unweighted graphs, it is possible to use weighted graphs as input by subdividing each edge, replacing each edge e with $\lceil w(e) \rceil$ edges, where

$w(e)$ is the weight of e. Since the average edge weight in road networks is typically small (e.g., as observed in [53]), this will only increase the number of vertices and edges in the graph by a constant factor that is independent of the size of the graph. This preprocessing step is important for applications of road network reconstruction in routing services, where the completeness of road network data has great importance. For example, machine learning techniques have been utilized in the past to find the missing roads in incomplete road network data [52]. Though our experiments focus on unweighted road networks, we include some experimental results for weighted road networks with subdivided edges as well.

Another application relevant to this work is the use of structured encryption [31] in the context of cloud computing, where a data owner encrypts structured data, such as a graph, stores it in a database managed by a third-party cloud provider, and wishes to query it privately (e.g., using single-pair shortest path queries [57]). In the scenario where an adversary server is able to generate valid queries of its own, it would be able to use a graph reconstruction algorithm to learn the edges in the graph, resulting in a breach of privacy.

A graph reconstruction algorithm A is evaluated based on the number of queries it issues, which we call the *query complexity* of A, following nomenclature from learning theory (e.g., see [2, 3, 32, 39, 104]) and complexity theory (where this is also known as "decision-tree complexity," e.g., see [119, 25]). For instance, Kannan, Mathieu and Zhou [72] present exact learning algorithms for connected, undirected graphs that have bounded degree, including a randomized algorithm that has expected query complexity $O(\Delta^3 n^{3/2} \operatorname{polylog}(n))$, where Δ is the maximum degree of the graph, using distance queries. This bound simplifies to $O(n^{3/2} \operatorname{polylog}(n))$ for graphs with maximum degree $O(\operatorname{polylog}(n))$.

We note that a bound on the maximum degree is necessary for subquadratic exact learning algorithms, as there is a simple $\Omega(n^2)$ lower bound for the query complexity of graphs with unbounded degrees, e.g., see [72]. Likewise, a trivial upper bound for the task of reconstructing a general graph G is $O(n^2)$ distance queries, as one can issue a distance query

for every pair of vertices in the graph and return all pairs of vertices that have distance 1 between them as edges. We refer to this as an *exhaustive search* on G.

2.1.1 Related Prior Results

The problem of reconstructing graphs by issuing queries has been studied extensively, e.g., see [1, 6, 11, 22, 96, 113, 5, 7, 8, 12, 15, 16, 32, 36, 68, 69, 77, 98]. These works differ in terms of their assumptions about the hidden graph (e.g., whether the hidden graph is a tree, a general graph, or something else) or the types of queries that they issue.

In terms of the most relevant prior work, Kannan, Mathieu and Zhou [72] showed how to reconstruct a connected, unweighted graph G using $O(\Delta^3 n^{3/2} \operatorname{polylog}(n))$ distance queries in expectation, where they performed an exhaustive search on the Voronoi cells created by a call to a graph clustering algorithm inspired by Thorup and Zwick [106]. They also raised the open question of whether we can achieve an algorithm that uses $O(n \operatorname{polylog}(n))$ distance queries in expectation for bounded degree graphs. In a recent work [87], Mathieu and Zhou provided a partial answer for that open question by providing an algorithm that uses $O(n \operatorname{polylog}(n))$ distance queries in expectation for random Δ-regular graphs. However, this does not imply an algorithm with an expected query complexity of $O(n \operatorname{polylog}(n))$ distance queries for road networks as they are not necessarily regular. For general graphs of bounded degree, their algorithm uses $O(n^{5/3} \operatorname{polylog}(n))$ distance queries in expectation.

In another work, Afshar, Goodrich, Matias and Osegueda [7] introduced a parallel implementation of the graph clustering technique of Thorup and Zwick [106] and presented a parallel algorithm for reconstructing connected, unweighted graphs using $O(\Delta^2 n^{3/2+\varepsilon})$ distance queries in $O(1)$ parallel rounds for constant $0 < \varepsilon < 1/2$, with high probability.

2.1.2 Our Contributions

In this chapter, we introduce a randomized incremental algorithm for exact reconstruction of bounded degree graphs and demonstrate through experiments that it has expected empirical query complexity $O(n \operatorname{polylog}(n))$, providing an empirical answer to the open question raised by [72] mentioned above.

The main idea of our algorithm is to cluster the graph into cells by incrementally selecting random vertices as centers. We then issue distance queries between that center and the rest of the graph to decide which vertices should be added to the new cell. We continue this process until the size of each cell is below some threshold value. The final step is to then perform exhaustive searches in each cell.

Our algorithm uses the same overall strategy used in [72], which is based on finding a Voronoi cell decomposition of the graph. However, our algorithm differs in a number of important ways. In [72], the goal of the algorithm is to produce cells such that the size of each cell is $O(\sqrt{n}/\Delta)$. Our algorithm, however, produces cells that have size at most a chosen constant. Since performing exhaustive searches on all of these cells requires only $O(n)$ queries, the query complexity of our algorithm depends mainly on the initial step of constructing the cells and not the exhaustive querying step. Moreover, our algorithm incrementally constructs the cell decomposition by updating it with each newly added center, whereas [72] updates the cell decomposition only after adding multiple centers.

We perform experiments on several real-world road networks and show, by considering the number of queries performed at each step, that our algorithm has expected empirical query complexity $O(n \operatorname{polylog}(n))$. Moreover, we theoretically analyze our algorithm and prove an upper bound of $O(d_{\max}^2 n \log n)$ expected queries, where $d_{\max}$ is the maximum degree in the dual graph of cells during our algorithm. To characterize $d_{\max}$, we collect data on the maximum cell degrees during our experiments, and find that the value of $d_{\max}$ scales logarith-

mically with respect to n for road networks. When combined with our theoretical analysis, this results in an alternative way to obtain an empirical upper bound of $O(n \operatorname{polylog}(n))$ expected queries for our algorithm. In addition, we perform experiments to directly compare the number of queries our algorithm issues to the number of queries issued by existing algorithms, and observe empirically that our algorithm issues significantly fewer queries.

This chapter is organized as follows. We provide some preliminaries in Section 2.2, our algorithm is in Section 2.3, the results from our analysis are in Section 2.4, experimental results are in Section 2.5, comparisons between theoretical/experimental results are in Section 2.6, and the conclusions are in Section 2.7.

2.2 Preliminaries

We reconstruct graphs $G = (V, E)$ that consist of $n = |V|$ vertices and $m = |E|$ edges, and are undirected, unweighted and connected. For a graph $G = (V, E)$, a *cell* is defined as any subset of V. A *cover* of G is a set of cells $\mathcal{C}$ such that $\bigcup_{C \in \mathcal{C}} C = V$ and for each edge $(u, v) \in E$ there exists at least one cell $C \in \mathcal{C}$ such that $u, v \in C$.

For two vertices $u, v \in V$, $\delta(u, v)$ denotes the number of edges on the shortest path between u and v in G. For a subset of vertices $A \subseteq V$, $\delta(v, A) = \min_{a \in A} \delta(v, a)$. For a vertex v and a cell C, the subroutine Distances(v,C) determines $\delta(v, u)$ $\forall u \in C$ by issuing distance queries between v and every vertex in C.

2.3 Algorithm

The first component of our algorithm is RECONSTRUCT(V), which takes as input the set of vertices V of the input graph and returns the reconstructed graph with the correct edge

Algorithm 1: RECONSTRUCT(V)

```
1  E ← ∅, C ← ∅, W ← V
2  Let M denote the maximum size of cells, initially +∞
3  while M > g do // g is a chosen constant
4      a ← a random vertex from W
5      W ← W \ {a}
6      C ← cover(V, C, a)
7      M ← max_{C∈C} {|C|}
8  for C ∈ C do
9      E ← E ∪ EXHAUSTIVE-QUERY(C)
10 return E
```

assignments. We start by choosing a constant, g, which is the threshold value for the maximum sized cell in our cover. In a loop, we randomly select an unselected vertex to be the center for the new cell, and call COVER$(V, \mathcal{C}, a)$ to get a new cover which includes the new cell with center a. We describe how COVER$(V, \mathcal{C}, a)$ works later in this section.

We keep performing this loop until the maximum sized cell in the cover becomes less than g, in which case we terminate the loop and perform an exhaustive search on each cell of the cover. The function EXHAUSTIVE-QUERY(C) takes as input a cell C and returns all edges between vertices in C by issuing distance queries for each pair of vertices in C. We provide details in Algorithm 1.

The second and main component of our algorithm is COVER$(V, \mathcal{C}, a)$ (see Algorithm 2), which takes as input the set of vertices V, a set of cells $\mathcal{C}$ and a vertex a, and returns a new cover where a is the center of a new cell N. We define S, which we call the *frontier*, to be the set of cells that we should search in expanding N, and we initialize it with the cells that a belongs to. The only exception is when we first call COVER$(V, \mathcal{C}, a)$, in which case we initialize S to be $\{V\}$ (see lines 3-6). Then, an arbitrary cell, C, from S is chosen, and we issue distance queries between a and all of the vertices of C.

Using the results from these queries, we determine which vertices in C are close to a, compared to their distances to all the other centers. We define A to be a global variable that stores the set of all centers that were added before the new center a. For a vertex $v \in C$, if $\delta(a, v) \leq \delta(A, v) - 1$, we remove v from all of the cells that contains it (see line 18). If, however, $\delta(a, v) = \delta(A, v)$ or $\delta(a, v) = \delta(A, v) + 1$, we consider v to be on the *boundary* between C and N and so we do not remove v from any cells. In both cases, we add v to the new cell N, and we add any unvisited cells that contain v to S since they might have vertices that are close to a as well.

We say that a cell $C_2 \in \mathcal{C}$ is a *neighbor* of $C_1 \in \mathcal{C}$ if $C_1 \cap C_2 \neq \emptyset$. In other words, two cells are neighbors if there exists a boundary vertex that belongs to both of them. So, each iteration of COVER$(V, \mathcal{C}, a)$ ends up adding to the frontier all unvisited neighbors of the current cell $C \in S$ that share at least one boundary vertex with C such that this boundary vertex can be added to N according to the closeness definition in line 15. Note that we do not necessarily add all the neighbors of C to the frontier: if none of the boundary vertices v between C and a neighboring cell N' have distance at most $\delta(A, v) + 1$ to N's center, then it is clear that none of the vertices in N' can be added to N.

2.4 Correctness and Analysis

Theorem 2.4.1. *For any undirected, unweighted and connected graph $G = (V, E)$,* RECONSTRUCT(V) *correctly reconstructs E.*

Proof. We use an inductive argument to prove that the union of exhaustive searches performed on the cells created by the algorithm discovers all $(u, v) \in E$.

Initially, there is a single cell containing all of the vertices V, which trivially covers all the edges of E. Now, let A_i represent the first i centers that we add in the algorithm and assume,

Algorithm 2: COVER$(V, \mathcal{C}, a)$ algorithm for constructing a new cover after adding a cell centered at vertex a

$N \leftarrow \{a\}$ // N is the new cell centered at a
$L \leftarrow \emptyset$ // L is the set of cells that have been visited
if $\mathcal{C} = \emptyset$ **then**
    $S \leftarrow \{V\}$
else
    $S \leftarrow \{C \in \mathcal{C} \mid a \in C\}$ // S is the set of cells that we should search in expanding N
while $S \neq \emptyset$ **do**
    $C \leftarrow$ an arbitrary cell from S
    $S \leftarrow S \setminus \{C\}$
    $L \leftarrow L \cup C$
    Distances(a, C)
    // A is a global variable denoting the set of all cell centers
    If $A = \emptyset$, $\forall v \in V$: set $\delta(A, v) = +\infty$.
    for $v \in C$ **do**
        if $\delta(a, v) \leq \delta(A, v) + 1$ **then**
            $S \leftarrow S \cup \{C' \in \mathcal{C} \mid v \in C'$ **and** $C' \notin L\}$ // add all of the unvisited cells that contain v to the frontier S
            if $\delta(a, v) \leq \delta(A, v) - 1$ **then**
                Remove v from all the cells that contain it
            $N \leftarrow N \cup \{v\}$
$A \leftarrow A \cup \{a\}$
return $\mathcal{C} \cup \{N\}$

at every step $2 \leq s \leq i$, that for each edge $(u, v) \in E$ there is a cell with its center in A_s that contains both u and v. We then prove that if we create a new cell N, centered at the $(i+1)$-th center $a \in A_{i+1}$, the union of the new cells still covers all the edges in E.

Consider an edge $(e_1, e_2) \in E$. Let x be the last center among the first $i+1$ centers such that $x = \text{argmin}_{a \in A_{i+1}} \{\min(\delta(a, e_1), \delta(a, e_2))\}$. In other words, x is the last center that is closest to either endpoint of the edge (e_1, e_2). If $\delta(x, e_1) \neq \delta(x, e_2)$, we denote the endpoint that is closer to x as v, and denote the other endpoint as u. Otherwise, we denote the endpoints arbitrarily as u and v. So, we have $\min(\delta(x, e_1), \delta(x, e_2)) = \delta(x, v) = \delta(A_{i+1}, v) \leq \delta(A_{i+1}, u)$. We prove that both u and v belong to the cell centered at x, after the $(i+1)$-th iteration.

First, we prove that we add both v and u to the cell at x. Let $(s_1, s_2, \ldots, s_m)$ denote the ordered vertices on a shortest path from x to v, where $s_1 = x$ and $s_m = v$. Using the inductive hypothesis for each $2 \leq j \leq m$, there exists a cell that contains both s_{j-1} and s_j right before adding center x. Now, consider the smallest j such that s_j is not added to the cell at x during the loop at line 14. Since s_{j-1} is added to the cell at x, and since s_{j-1} and s_j are connected, then by the inductive hypothesis there is a cell C that contains both s_{j-1} and s_j. Therefore, when we add s_{j-1} to the cell at x, we add C to the set of cells that we should explore in expanding the cell centered at x (see line 16). On the other hand, since s_j is on the shortest path from x to v and $\delta(x, v) = \delta(A_{i+1}, v)$, then $\delta(x, s_j) = \delta(A_{i+1}, s_j)$. Therefore, s_j will be added to the cell at x when exploring cell C. Using this inductive approach, all vertices on the shortest path from x to v will be added to the cell at x. Finally, since $\delta(x, u) \leq \delta(x, v) + 1 = \delta(A_{i+1}, v) + 1 \leq \delta(A_{i+1}, u) + 1$, and since u and v also have a common cell, we add u to the cell at x.

Next, we prove that if we add v and u to the cell centered at x, no other cells that we create later on in the first $(i+1)$-th steps removes v or u from the cell at x. Note that for removing a node from cell x, the condition at line 17 must hold. Since $\delta(x, v) = \delta(A_{i+1}, v)$, there will be no center b among the first $(i + 1)$ centers such that $\delta(b, v) \leq \delta(A_{i+1}, v) - 1$, meaning that v will stay in cell at x. On the other hand, we remove u from x only if for a center b: $\delta(b, u) \leq \delta(x, u) - 1$. If $\delta(b, u) \leq \delta(x, u) - 1$, and given the fact that $\delta(x, u) \leq \delta(x, v) + 1$ and $\delta(x, v) \leq \delta(x, u)$, then $\delta(b, u) \leq \delta(x, v) \leq \delta(x, u)$. However, we assumed that x is the last center, among the first $i + 1$ centers, that is closest to either of the endpoints u and v. Therefore, u will also stay in the cell at x. □

Theorem 2.4.2. *The expected query complexity of* RECONSTRUCT(V) *is* $O(d_{\max}^2 n \log n)$, *where* $d_{\max}$ *is the maximum cell degree over all steps.*

Proof. We use a backwards analysis [99] to derive an expression for the expected query complexity of the algorithm. We assume i centers have already been added, and analyze the expected number of queries we issue at step i.

We observe that our algorithm only issues distance queries for cells in the set S. Moreover, the only cells we add to S are the ones that contain vertices that get added to the ith cell. This means that all cells in S will become neighbors of the ith cell at the end of step i. So the number of distance queries issued at step i is the sum of the sizes of each cell that gets added to S, which is at most the sum of the sizes of the ith cell and its neighbors at the end of step i. Denoting the set of cells at the end of step i as $\mathcal{C}_i$, and the set of cells neighboring any cell C as $N(C)$, we have that the expected number of queries issued at the ith step is

$$\begin{aligned}
&\leq \sum_{C \in \mathcal{C}_i} \frac{1}{i}(|C| + \sum_{C' \in N(C)} |C'|) \\
&= \sum_{C \in \mathcal{C}_i} \frac{(d(C)+1)|C|}{i},
\end{aligned}$$

by observing that each cell size $|C|$ is summed $d(C)+1$ times, where $d(C)$ denotes the *degree* of cell C, i.e. the number of neighboring cells it has. To bound this summation, we express each cell size as the sum of boundary and non-boundary vertices. We have

$$\begin{aligned}
&\sum_{C \in \mathcal{C}_i} \frac{(d(C)+1)|C|}{i} = \sum_{C \in \mathcal{C}_i} \frac{(d(C)+1)(|C|_{NB} + |C|_B)}{i} \\
&\leq \frac{(d_{\max}+1)}{i} \left((\sum_{C \in \mathcal{C}_i} |C|_{NB}) + (\sum_{C \in \mathcal{C}_i} |C|_B) \right)
\end{aligned}$$

where $d_{\max}$ is the maximum degree of any cell during any step, $|\cdot|_B$ denotes the number of boundary vertices, and $|\cdot|_{NB}$ denotes the number of non-boundary vertices. We use the fact that $\sum_{C \in \mathcal{C}_i} |C|_{NB} \leq n$, and observe that $\sum_{C \in \mathcal{C}_i} |C|_B \leq (d_{\max}+1) \cdot n$ as each boundary vertex can belong to at most $d_{\max}+1$ cells, and thus can only be counted that many times

at most in the summation. So, we have

$$\frac{(d_{\max}+1)}{i}\left((\sum_{C\in\mathcal{C}_i}|C|_{NB})+(\sum_{C\in\mathcal{C}_i}|C|_B)\right)<\frac{n(d_{\max}+2)^2}{i}.$$

The expected number of queries when all steps are considered is

$$<\sum_{i=1}^{\#\text{steps}}\frac{n(d_{\max}+2)^2}{i}=n(d_{\max}+2)^2\sum_{i=1}^{\#\text{steps}}\frac{1}{i},$$

which is $O(d_{\max}^2 n\log n)$. To finish our analysis, we also need to consider the number of queries issued during the exhaustive searches in each cell. Since the total number of cells is $O(n)$, and each cell is of size at most a constant g, the exhaustive querying part has query complexity $O(n)$. □

2.5 Experimental results

2.5.1 Implementation and Datasets

We implemented our algorithm in C++, and simulated the distance query oracle by performing BFS in each iteration to compute distances between nodes while keeping track of how many distance queries would be necessary to find these distances. We selected the value of g to be 50. We include experimental results for road networks from 50 U.S. states and Washington, D.C. obtained from the formatted TIGER/Line dataset available from the 9th DIMACS Implementation Challenge website[1] and road networks from Belgium, the U.K. (limited to the road network of Great Britain), Italy, Luxembourg, and the Netherlands obtained from formatted OpenStreetMaps data available from the 10th DIMACS Implementation Challenge [17]. For all of the datasets, only the largest connected component is

[1] `http://www.diag.uniroma1.it/~challenge9/data/tiger/`

considered. In Section 2.6, we discuss how the upper bounds derived from these experiments compare to our theoretical upper bound.

2.5.2 Batch Length

We define the *batch length* of a step to be the number of distance queries issued at that step. To find the relation between batch length and the step number, based on the theoretical upper bound we derived in Section 2.4, we fit the function $\textsc{Batch-Length}(step) = a + b\frac{n}{\text{step}}$ to the data points in our results, where a and b are the fitting parameters. Data points for the batch lengths of some of our datasets are provided in Figure 2.1. We list our results for all of the datasets in Table 2.1, which includes the best-fit parameters in columns a and b, and the maximum number of cells visited at any step in column M.

We can see that parameter b does not exceed 2, and that parameter a is close to 0 for all of the datasets. This suggests that a constant or logarithmic factor of $\frac{n}{\text{step}}$ could be an upper bound for the batch size at any step, which leads us to predict an upper bound of $\log n \cdot \frac{n}{\text{step}}$ which we show in Figure 2.1. We report the percentage of steps that fall below this upper bound for each dataset in Table 2.1, column U.

2.5.3 Maximum Cell Degree

To combine our experimental results with our theoretical upper bound, we collected data on the maximum cell degree at each step. We combine the step-wise data in each dataset using different measures: mean, max, and the 1st, 2nd and 3rd quartiles to see how the data is spread. Then for each dataset, we represent the value corresponding to each measure as a point. Based on our intuition, we fit the function $a \log n + b$ for each measure. We list our results in Figure 2.2, which includes the best-fit parameters for each measure. We

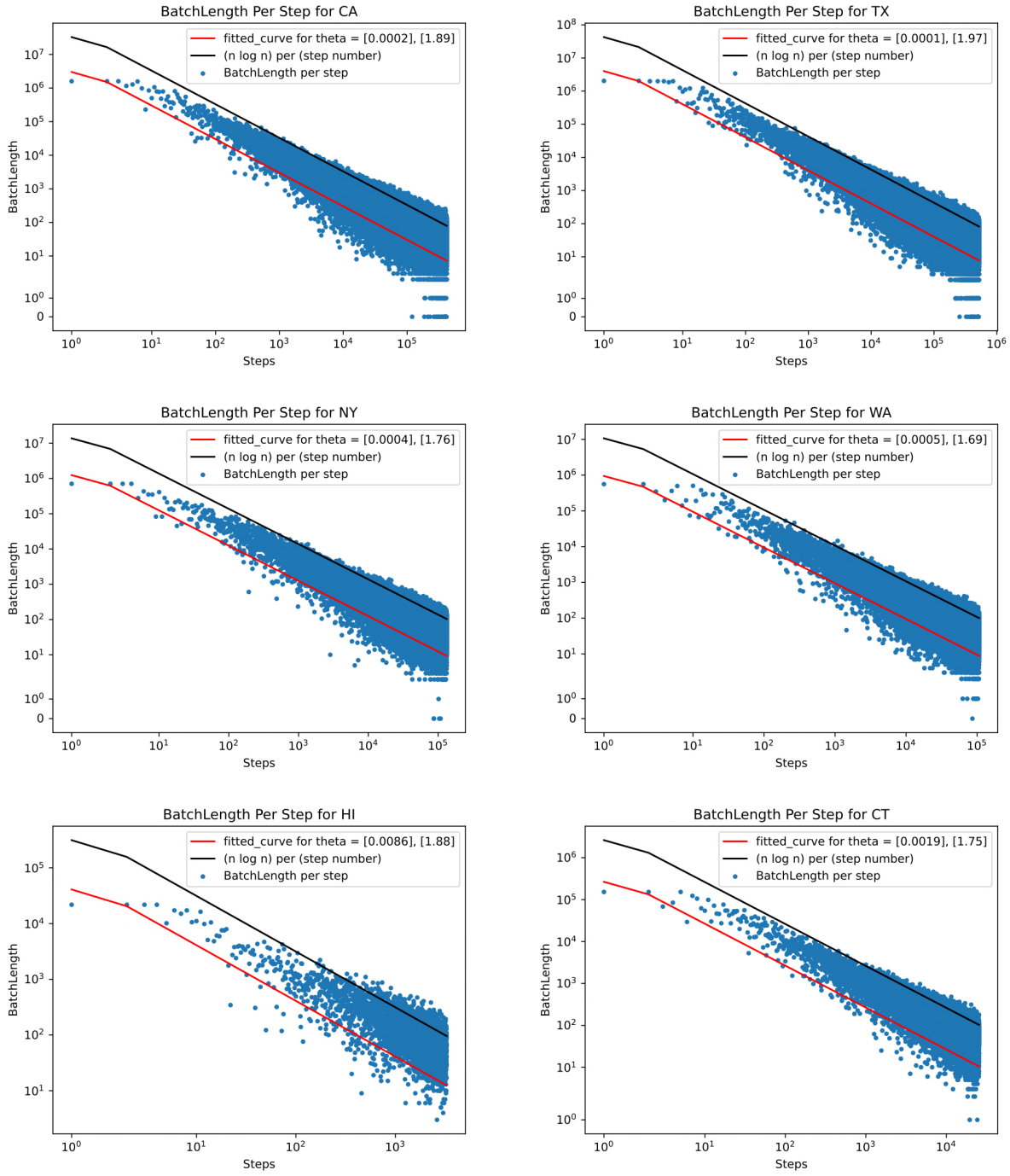
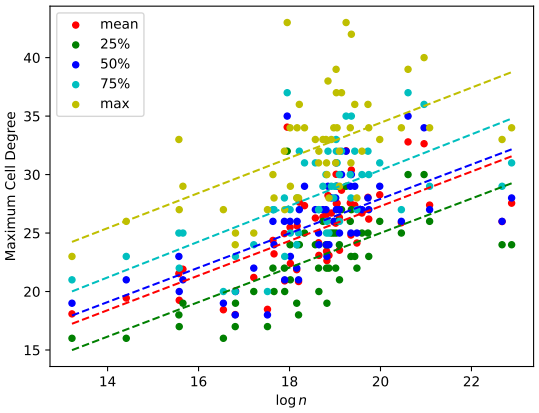


Figure 2.1: Batch lengths for select datasets of varying sizes.

Table 2.1: Batch length results for all datasets. Columns a, b and U were rounded to 4, 2 and 2 decimal places respectively.
a, b: best-fit parameters for BATCH-LENGTH$(step\#) = a + b\frac{n}{step\#}$
U: percentage of batch lengths that are below the upper bound of $\frac{n}{step\#} \log n$

Dataset	n	a	b	U	Dataset	n	a	b	U
AK	48 560	0.0036	1.69	96%	ND	203 583	0.0015	1.76	92%
AL	561 459	0.0005	1.91	98%	NE	304 335	0.0011	1.99	93%
AR	478 024	0.0005	1.85	98%	NH	115 055	0.0023	1.91	97%
AZ	533 008	0.0005	1.95	95%	NJ	329 404	0.0010	1.90	94%
CA	1 595 577	0.0002	1.89	96%	NM	456 896	0.0006	1.88	97%
CO	436 084	0.0006	1.91	96%	NV	253 012	0.0009	1.85	94%
CT	152 036	0.0019	1.75	94%	NY	708 520	0.0004	1.76	97%
DC	9522	0.0321	1.89	70%	OH	672 527	0.0005	1.93	95%
DE	48 812	0.0053	1.77	91%	OK	535 032	0.0006	1.87	96%
FL	1 036 647	0.0003	1.86	97%	OR	529 702	0.0005	1.90	98%
GA	731 954	0.0004	1.97	97%	PA	866 352	0.0004	1.83	97%
HI	21 774	0.0086	1.88	90%	RI	51 642	0.0047	1.88	92%
IA	388 487	0.0008	1.92	93%	SC	460 763	0.0005	1.81	97%
ID	265 552	0.0010	1.81	97%	SD	206 998	0.0014	1.85	94%
IL	790 439	0.0004	1.89	96%	TN	578 981	0.0004	1.70	98%
IN	495 581	0.0007	1.88	94%	TX	2 037 156	0.0001	1.97	97%
KS	471 066	0.0007	1.87	93%	UT	242 432	0.0010	1.95	96%
KY	463 542	0.0006	1.90	98%	VA	620 680	0.0004	1.92	98%
LA	408 161	0.0006	1.84	97%	VT	95 672	0.0022	1.95	98%
MA	294 345	0.0009	1.98	95%	WA	560 336	0.0005	1.69	97%
MD	264 378	0.0010	1.86	95%	WI	514 687	0.0006	1.89	96%
ME	187 315	0.0013	1.81	99%	WV	292 557	0.0006	1.89	99%
MI	661 718	0.0005	1.74	95%	WY	243 545	0.0010	1.92	97%
MN	541 166	0.0006	1.76	95%	BE	1 441 295	0.0002	2.00	99%
MO	668 322	0.0004	1.89	97%	GB	7 733 822	0	1.81	100%
MS	409 994	0.0007	1.93	98%	IT	6 686 493	0	1.81	100%
MT	300 809	0.0007	1.80	98%	LU	114 599	0.0019	1.96	99%
NC	876 954	0.0003	1.72	99%	NL	2 216 688	0.0001	1.86	99%

can see that $a < 2$, and b is a small constant for each measure. The datasets with the largest maximum cell degrees turned out to be VA, NV and OH, with values of 43, 43 and 42 respectively. It is clear from the figure that a small constant multiple of $\log n$ would be enough to produce an upper bound that covers all of the data points, suggesting that the maximum cell degree might have an upper bound of $O(\log n)$.



(a) Best-fit lines for the function $a + b \log n$.

Measure	a	b
mean	-2.34	1.48
1st quartile	-4.52	1.48
2nd quartile	-1.49	1.47
3rd quartile	-0.2	1.53
max	4.4	1.50

(b) Parameters for best-fit lines. Columns a and b were rounded to 2 decimal places.

Figure 2.2: Results from combining step-wise maximum cell degrees.

Road Networks with Subdivided Edges.

We provide experimental results in Table 2.2 for the maximum cell degrees of the weighted road networks of the District of Columbia and the state of Hawaii, which we transform into unweighted graphs by replacing each edge e with $\lceil w(e) \rceil$ edges, where $w(e)$ is the weight of e. The size of each road network increased by factors of approximately 192 and 167 respectively, while the maximum cell degree values ended up decreasing for both road networks. This indicates that our algorithm can also perform efficiently on weighted road networks.

Table 2.2: Maximum cell degrees for weighted road networks compared to their unweighted versions.

	Unweighted		Weighted	
	$\lvert V \rvert$	$d_{\max}$	$\lvert V \rvert$	$d_{\max}$
DC	9522	23	1 826 049	12
HI	21 774	26	3 643 818	11

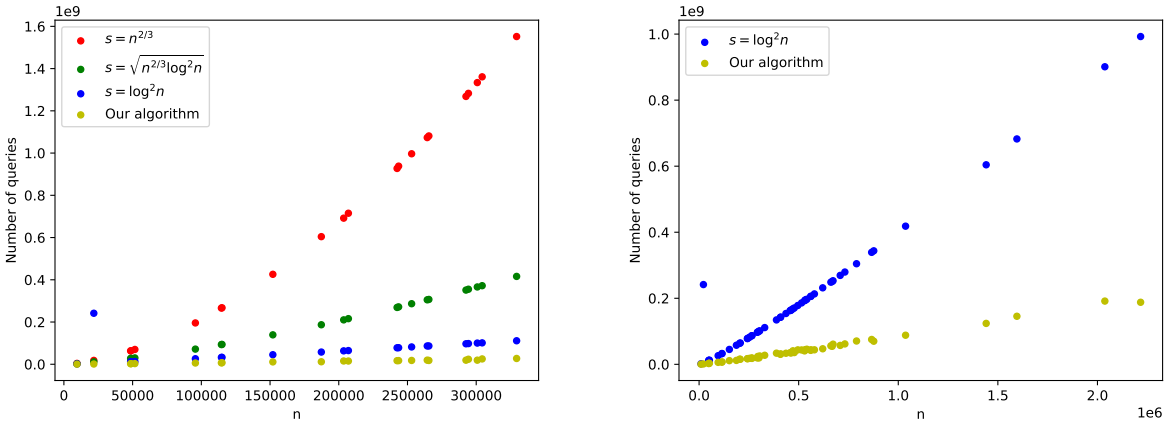


Figure 2.3: Number of queries issued by our algorithm compared to [87].

2.5.4 Comparisons with Existing Algorithms

We directly compare the number of queries issued by our algorithm to the algorithm introduced in [87], which takes as input a parameter s that affects the query complexity. The authors prove their query complexity bounds for Δ-regular graphs and bounded graphs with the value of s being set to $\log^2 n$ and $n^{2/3}$ respectively. We use these values for s in our experiments, and we also try setting s to be the geometric mean of these values. We summarize the results of our experiments on road networks in Figure 2.3.

We then compare our results to the number of queries issued by the algorithm in [72]. Without performing any experiments, it can be observed that this algorithm will issue significantly more queries than our algorithm: the first iteration of Estimated-Centers (Algorithm 2 in [72]) will issue at least $\Omega(n \cdot \Delta\sqrt{n} \cdot \log n \cdot \log\log n)$ distance queries.

2.6 Comparison of Theoretical/Experimental Results and Future Work

The theoretical upper bound we derived in Section 2.4 contains a $d_{\max}^2$ term, which can be $O(n^2)$ in the worst case. However, our experiments show that the maximum cell degree is actually low for road networks throughout the algorithm. From our results in Section 2.5.3, we can see that $O(\log n)$ would be a suitable upper bound for $d_{\max}$. In this case, the expected query complexity of our algorithm would be $O(n \log^3 n)$. The experimental results for batch length support this observation, as the upper bound for the batch length at any step number amounted to be $\frac{n \log n}{step\#}$, from which we get an expected query complexity of $O(n \,\mathrm{polylog}(n))$ as well. Future work might involve trying to find if there exists a better theoretical upper bound on the query complexity of our algorithm. This might require making some additional simplifying assumptions about the graphs being used as input.

We would like to point out a connection between our results and the graph-theoretical Delaunay triangulation of road networks.

2.6.1 Delaunay Triangulations and $d_{\max}$

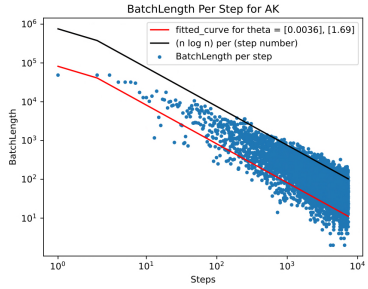
We can consider the cells (resp. covers) that are constructed during our algorithm as a redefinition of graph-theoretical Voronoi cells (resp. diagrams) (e.g., see [45]). Similarly, we can consider the dual graph connecting neighboring cells in the cover as being a form of a graph-theoretical Delaunay triangulation of G. There exists prior work on bounding the expected maximum degree of the Delaunay triangulation of a set of points selected randomly from the Euclidean plane. In [24], the authors consider the Delaunay triangulation of a Poisson point process limited to the portion of the triangulation within a cube of d dimensions. They show that the expected maximum degree of this triangulation is $\Theta(\log n / \log \log n)$. Having

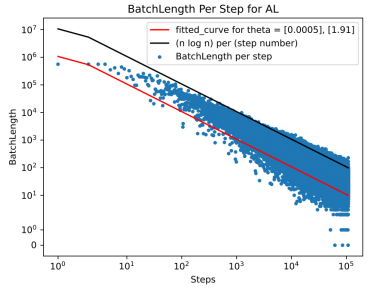
such a bound for the expected maximum degree for our redefinition of the graph-theoretical Delaunay triangulation might allow us to prove a theoretical quasilinear bound for the query complexity of our algorithm, so another interesting direction for future work can be to adapt this result for random point sets in Euclidean d-space to our setting, where the point set is selected randomly from the vertex set of a road network.

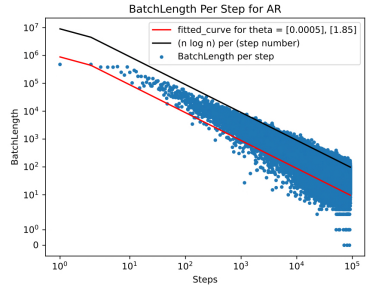
2.7 Conclusions

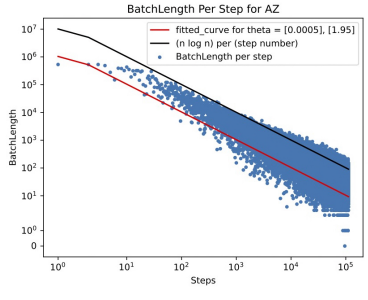
We introduced an efficient exact reconstruction algorithm for road networks and showed through experiments on several real-world road networks that our algorithm has an expected empirical query complexity that is quasilinear. As mentioned in Section 2.6, an important direction for future work can be to derive a theoretical upper bound for our algorithm that matches our experimental results.

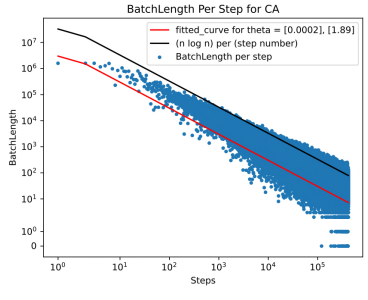
2.8 Batch Length Results for All Datasets

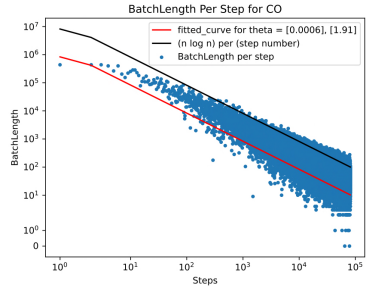


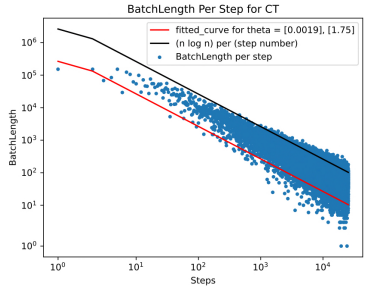


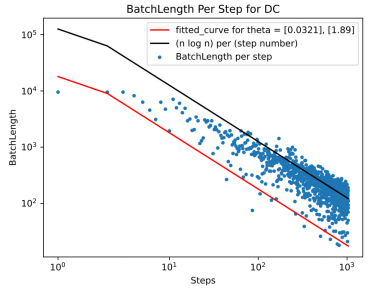


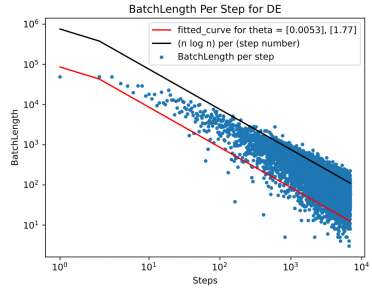


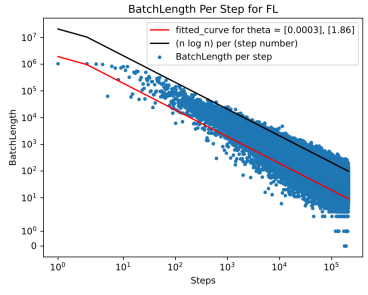


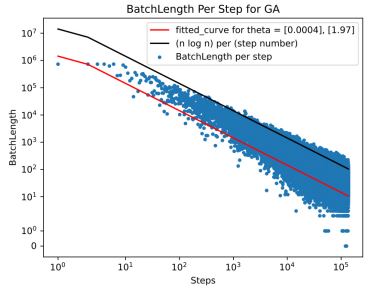


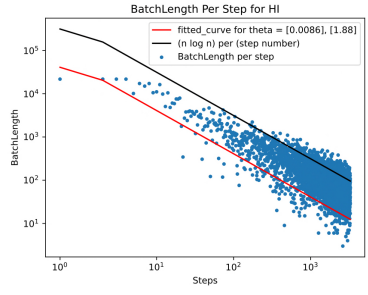


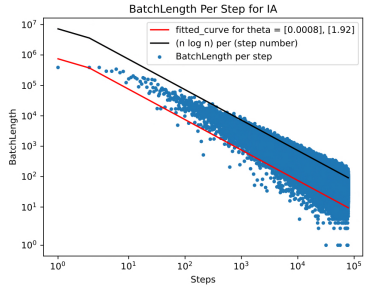


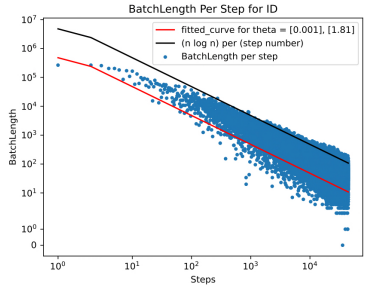


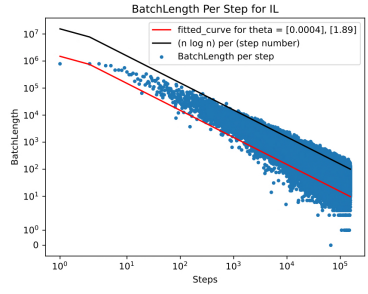


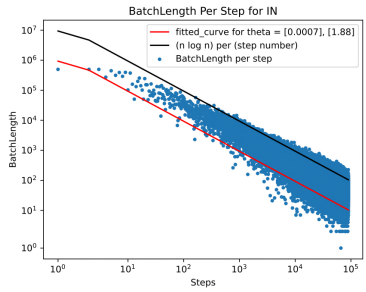


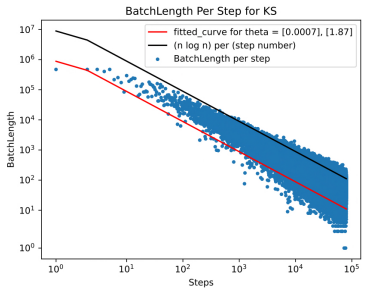


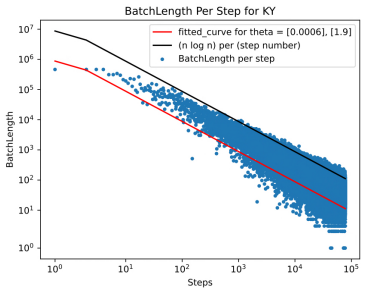


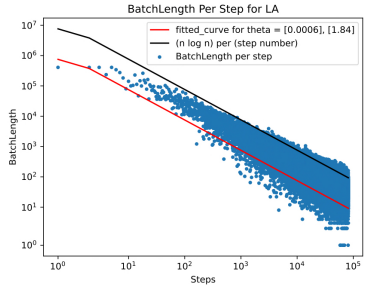


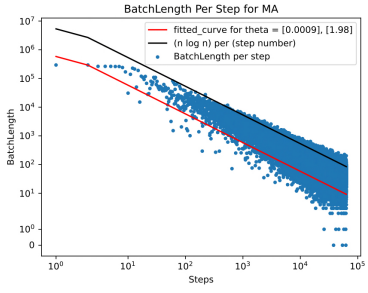


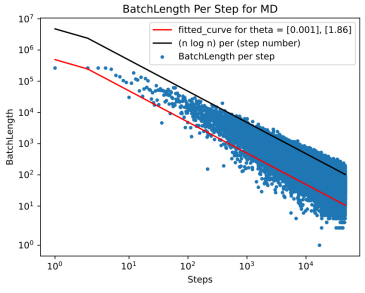

BatchLength Per Step for IN
fitted_curve for theta = [0.0007], [1.88]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

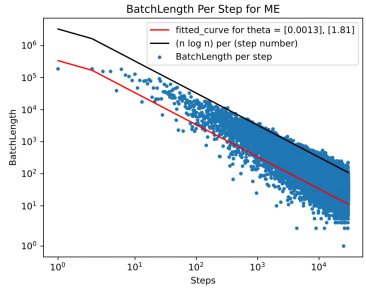

BatchLength Per Step for KS
fitted_curve for theta = [0.0007], [1.87]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

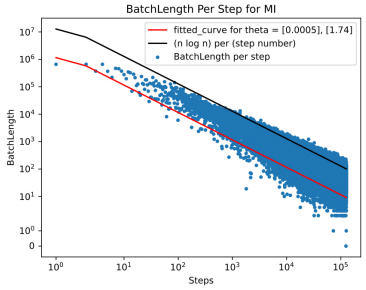

BatchLength Per Step for KY
fitted_curve for theta = [0.0006], [1.9]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

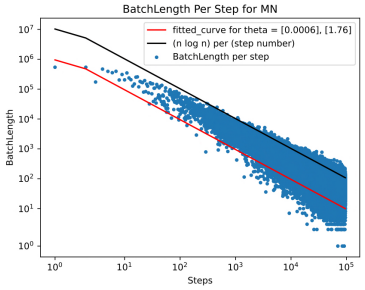

BatchLength Per Step for LA
fitted_curve for theta = [0.0006], [1.84]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

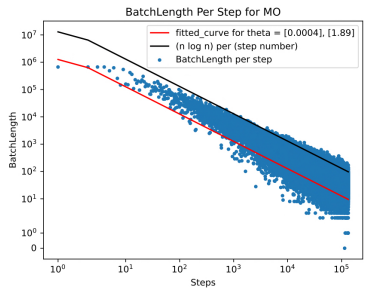

BatchLength Per Step for MA
fitted_curve for theta = [0.0009], [1.98]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

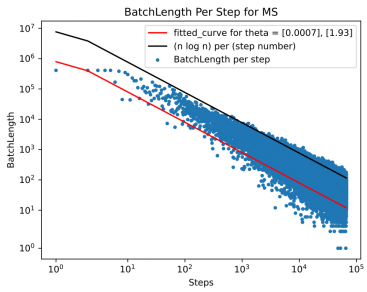

BatchLength Per Step for MD
fitted_curve for theta = [0.001], [1.86]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

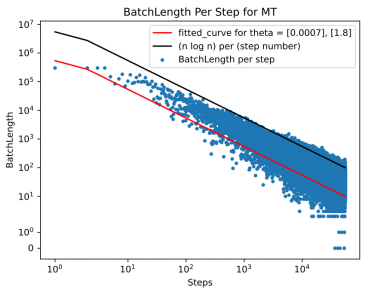

BatchLength Per Step for ME
fitted_curve for theta = [0.0013], [1.81]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

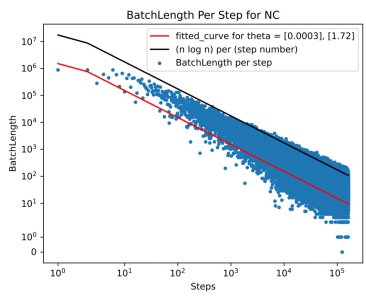

BatchLength Per Step for MI
fitted_curve for theta = [0.0005], [1.74]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

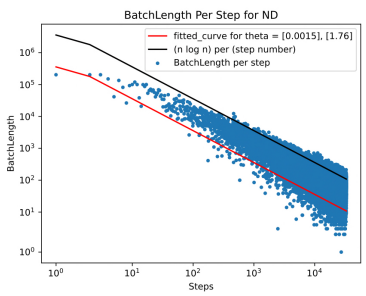

BatchLength Per Step for MN
fitted_curve for theta = [0.0006], [1.76]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

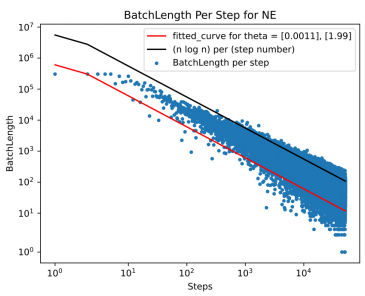

BatchLength Per Step for MO
fitted_curve for theta = [0.0004], [1.89]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

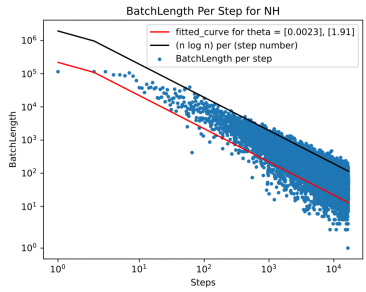

BatchLength Per Step for MS
fitted_curve for theta = [0.0007], [1.93]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

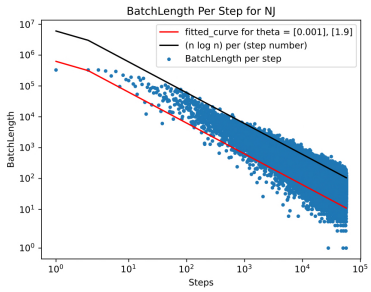

BatchLength Per Step for MT
fitted_curve for theta = [0.0007], [1.8]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

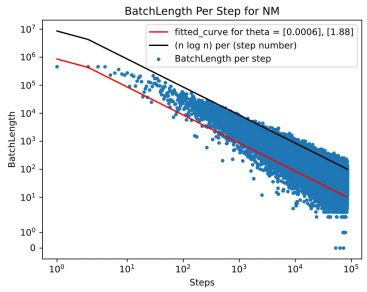

BatchLength Per Step for NC
fitted_curve for theta = [0.0003], [1.72]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

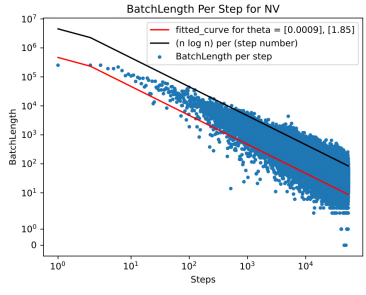

BatchLength Per Step for ND
fitted_curve for theta = [0.0015], [1.76]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

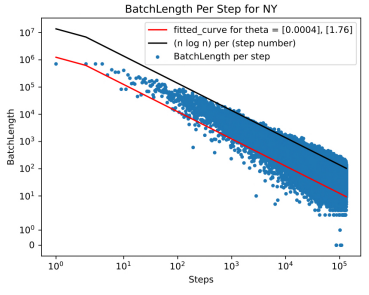

BatchLength Per Step for NE
fitted_curve for theta = [0.0011], [1.99]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

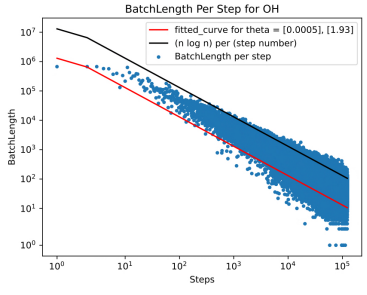

BatchLength Per Step for NH
fitted_curve for theta = [0.0023], [1.91]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

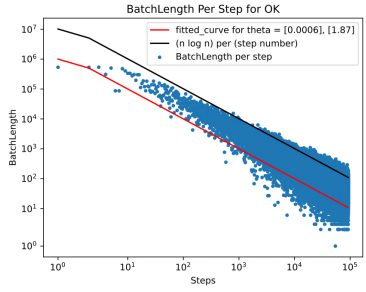

BatchLength Per Step for NJ
fitted_curve for theta = [0.001], [1.9]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

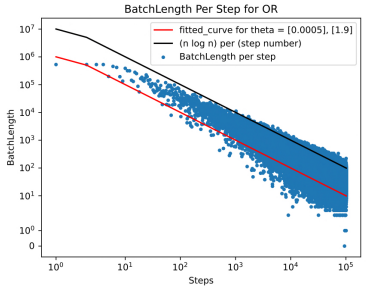

BatchLength Per Step for NM
fitted_curve for theta = [0.0006], [1.88]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

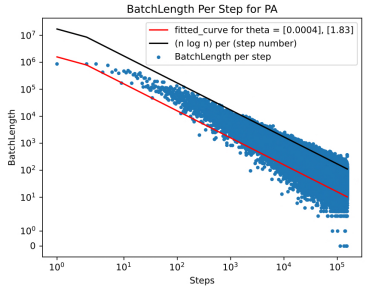

BatchLength Per Step for NV
fitted_curve for theta = [0.0009], [1.85]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

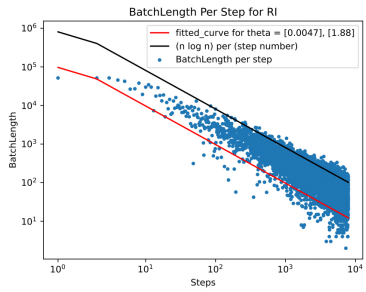

BatchLength Per Step for NY
fitted_curve for theta = [0.0004], [1.76]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

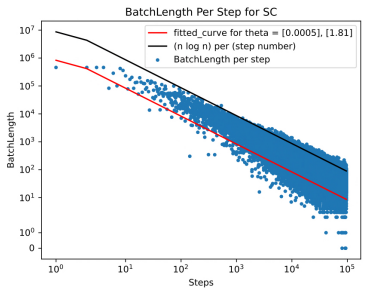

BatchLength Per Step for OH
fitted_curve for theta = [0.0005], [1.93]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

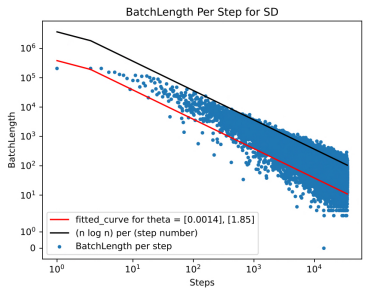

BatchLength Per Step for OK
fitted_curve for theta = [0.0006], [1.87]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

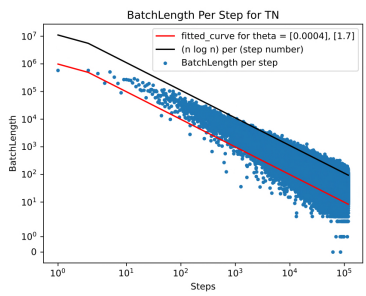

BatchLength Per Step for OR
fitted_curve for theta = [0.0005], [1.9]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

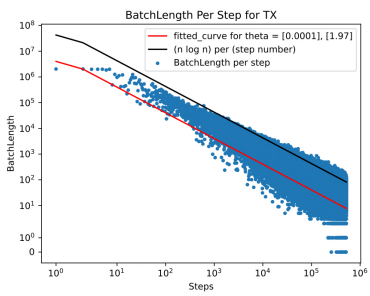

BatchLength Per Step for PA
fitted_curve for theta = [0.0004], [1.83]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

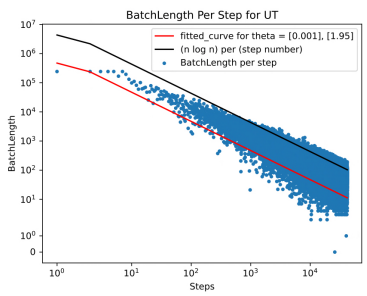

BatchLength Per Step for RI
fitted_curve for theta = [0.0047], [1.88]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

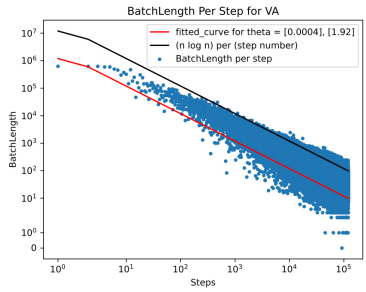

BatchLength Per Step for SC
fitted_curve for theta = [0.0005], [1.81]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

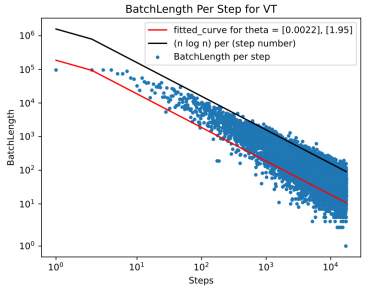

BatchLength Per Step for SD
fitted_curve for theta = [0.0014], [1.85]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

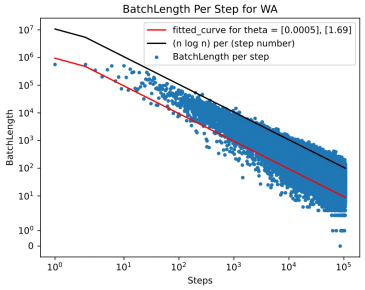

BatchLength Per Step for TN
fitted_curve for theta = [0.0004], [1.7]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

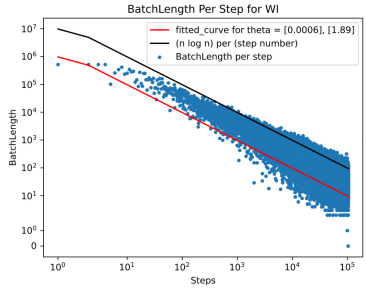

BatchLength Per Step for TX
fitted_curve for theta = [0.0001], [1.97]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

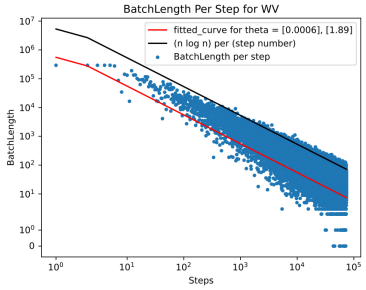

BatchLength Per Step for UT
fitted_curve for theta = [0.001], [1.95]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

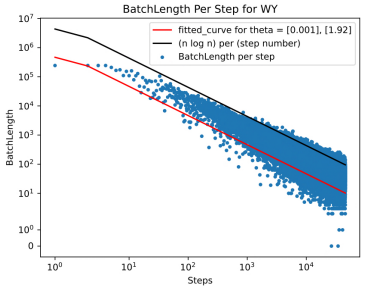

BatchLength Per Step for VA
fitted_curve for theta = [0.0004], [1.92]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

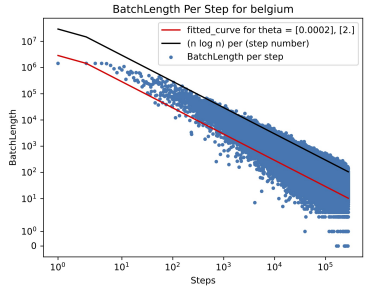

BatchLength Per Step for VT
fitted_curve for theta = [0.0022], [1.95]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

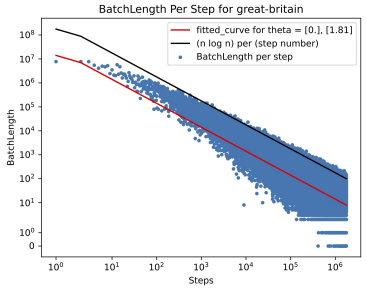

BatchLength Per Step for WA
fitted_curve for theta = [0.0005], [1.69]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

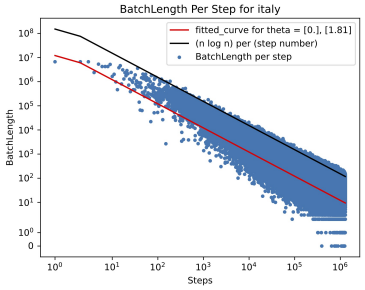

BatchLength Per Step for WI
fitted_curve for theta = [0.0006], [1.89]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

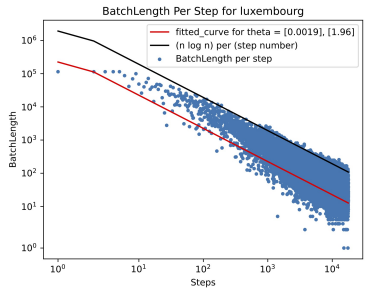

BatchLength Per Step for WV
fitted_curve for theta = [0.0006], [1.89]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

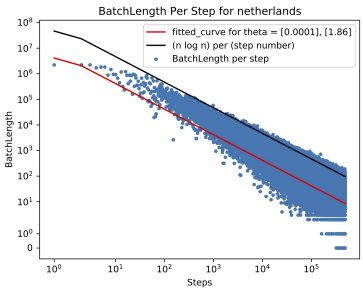

BatchLength Per Step for WY
fitted_curve for theta = [0.001], [1.92]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for belgium
fitted_curve for theta = [0.0002], [2.]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for great-britain
fitted_curve for theta = [0.], [1.81]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for italy
fitted_curve for theta = [0.], [1.81]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for luxembourg
fitted_curve for theta = [0.0019], [1.96]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps


BatchLength Per Step for netherlands
fitted_curve for theta = [0.0001], [1.86]
(n log n) per (step number)
BatchLength per step
BatchLength
Steps

Chapter 3

The Small-World Phenomenon in Road Networks

3.1 Introduction

The *small-world phenomenon* is the idea that all people are connected through a short chain of acquaintances that can be used to route messages. This phenomenon was popularized by the social-psychologist, Stanley Milgram, based on two experiments performed in the 1960s [88, 107], where a randomly-chosen group of people were given packages to send to someone in Massachusetts. Each participant was told that they should mail their package only to its target person if they knew them on a first-name basis; otherwise, they should mail their package to someone they knew who is more likely to know the target person. Remarkably, many packages made it to the target people, with the median number of hops being 6, which gave rise to the expression that everyone is separated by just "six degrees of separation" [70].

Subsequent to this pioneering research, many papers have been written on the small-world phenomenon, e.g., see [41], with a number of models having been proposed to explain it. Nevertheless, based on our review of the literature, the models proposed so far do not fully explain observations made by Milgram regarding his experiments [88, 107]. For example, Milgram observed that message routing occurred in a geographic setting with distances (measured in miles, presumably in the road network of the United States) roughly halving with each hop; see Figure 3.1.

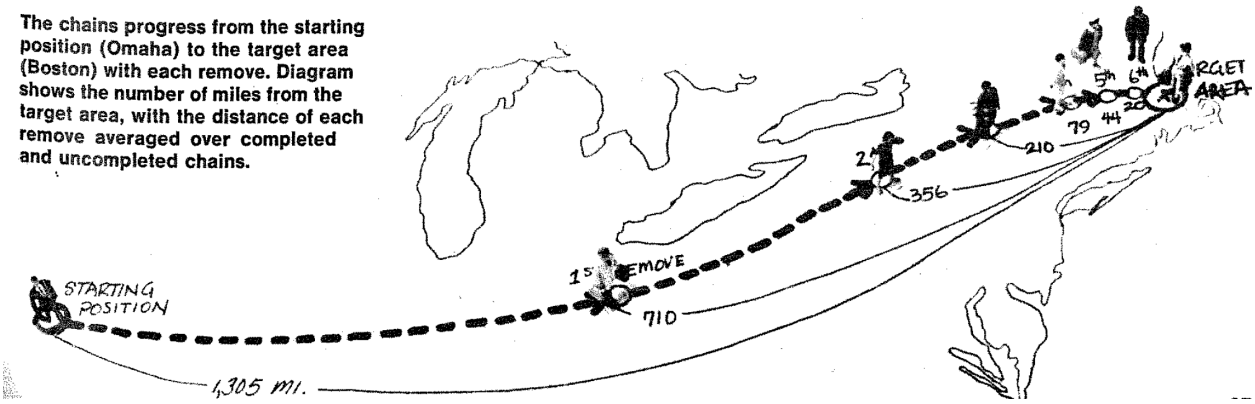


Figure 3.1: Illustration of geographic data from an original small-world experiment, from [88].

In spite of the geographic nature of the early small-world experiments,[1] we are not familiar with any previous work that models the small-world phenomenon with road networks. Thus, we are interested in this chapter in modeling the small-world phenomenon with road networks. For example, one of the surprising results in the original small-world experiments was that people were able to find very short paths among acquaintances with only a limited knowledge of the social network of acquaintances. This suggests that a model should explain how people can find short paths in a social network using a decentralized greedy algorithm, where individuals, who only have knowledge of their direct acquaintances, attempt to send a message towards a target along some path.

[1]The first experiment involved a group of people in Wichita, Kanasas who were asked to send a package to the wife of a divinity student in Cambridge, and the second experiment involved a group of people in Omaha, Nebraska (plus a small number of folks in Boston) who were asked to send a package to a stock broker who worked in Boston and lived in Sharon, Mass [88].

3.1.1 Related Prior Work

Arguably, the closest prior work on a model directed at explaining how small-world (social-network) greedy routing can work in a geographic setting is a well-known model by Jon Kleinberg [79]. Rather than using a road network, however, Kleinberg's model is built on a two dimensional $n \times n$ grid, where each grid point corresponds to a single person, with two types of connections—*local connections* and *long-range connections*. The local connections of the network are made by connecting each grid point to every other grid point within lattice distance $p \geq 1$. The long-range connections are made by connecting each grid point to $q \geq 0$ other grid points chosen randomly (typically with $q = 1$ or q being a small constant), such that the probability that grid point u is connected to grid point v is proportional to $[d_h(u, v)]^{-s}$, where $d_h(u, v)$ is the lattice distance between u and v, and s is the *clustering exponent* of the network. Kleinberg showed that in an $n \times n$ grid, a decentralized greedy algorithm, where each message holder forwards its message to an acquaintance that is closest to the target grid point, is able to achieve an expected path length of $O(\log^2 n)$ for $p = q = 1$ and $s = 2$, with a constant of at least 88 in the leading term in his Big-O analysis [79].

When attempting to model the original small-world experiments, however, there are a number of drawbacks with the Kleinberg model. First, it requires that the underlying distances are in the form of a grid, which is not compatible with how messages were sent in the original small-world experiments, where messages were sent using the U.S. road network. Second, the upper bound $O(\log^2 n)$, with a hidden constant that is at least 88, for the expected number of hops between vertices does not match the average hop length of six obtained in the original small-world experiments. For example, if $n = 9{,}000$, then $88 \log^2 n$ is approximately 15,000. Finally, as we show in Section 3.6, when acquaintanceship links are viewed as bidirectional, the maximum degree in the resulting network for the Kleinberg model is quite small. Having a degree distribution with a heavier tail might be more realistic for a social network. Moreover, these high-degree vertices might improve the performance of the model

during the routing step. Indeed, Milgram noted that in one of his experiments half of the successfully delivered packages were routed through three "key" individuals; see Figure 3.2.

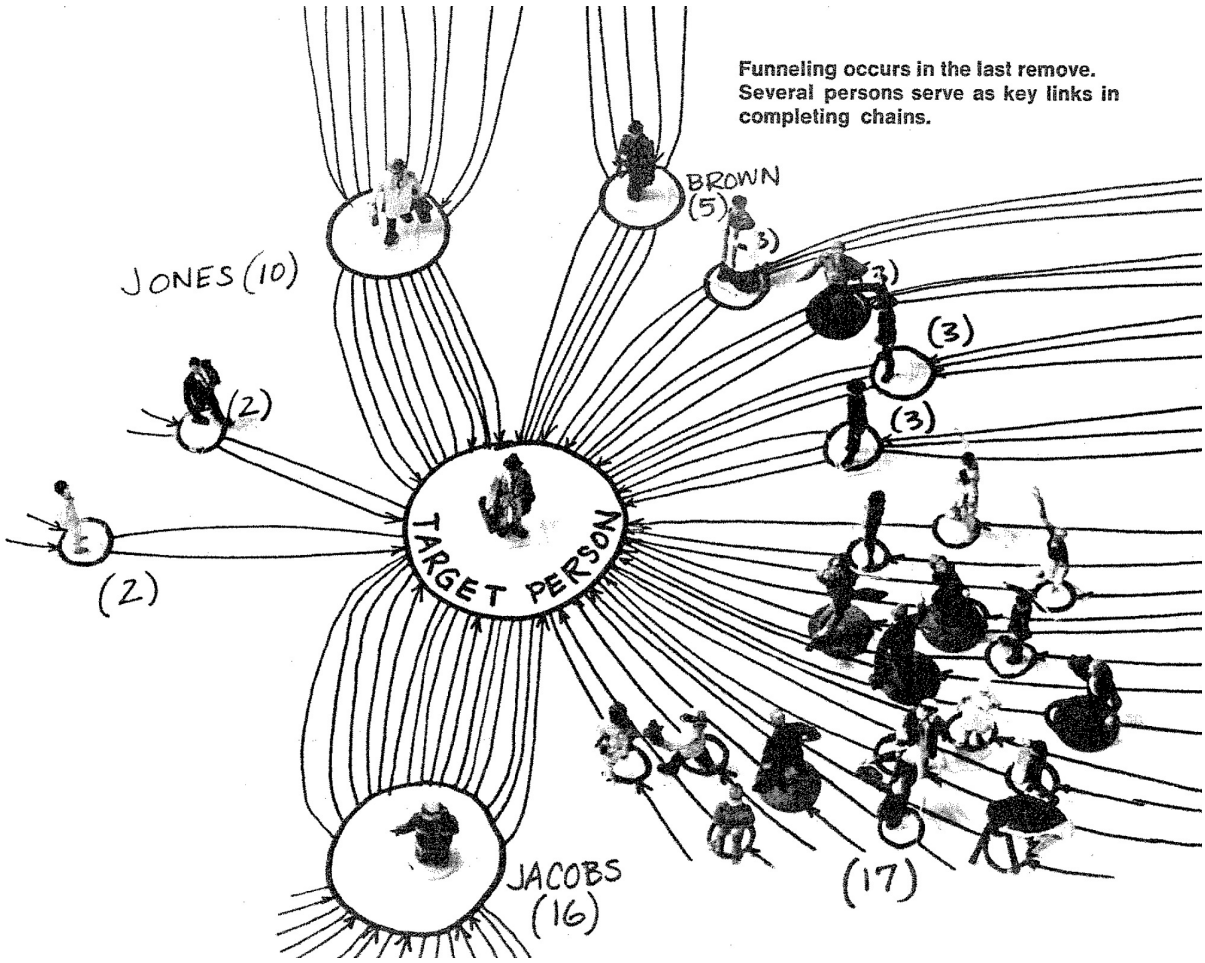


Figure 3.2: Final hops for the paths of delivered packages for people in an original small-world experiment, from [88]. Roughly half of the paths were routed through three "key" individuals, Jacobs, Jones, and Brown.

Another well-known social-network model is the *preferential attachment* model, which is a random graph model for non-geographic social networks, such as the World Wide Web. This model traces its roots back roughly 100 years, e.g., see [120, 37, 101], and was popularized and formalized by Barabási and Albert [18], who also coined the term *scale-free*, which describes networks where the fraction of vertices with degree d follows a power law, $d^{-\alpha}$, where $\alpha > 1$. A graph in the preferential attachment model is constructed incrementally, starting from a constant-sized "seed" graph, adding vertices one-at-a-time, such that when a vertex, v, is

added one adds a fixed number, m, of edges incident to v, where each other neighbor is chosen with probability proportional to its degree at that time, e.g., see [20]. This is often called a "rich-get-richer" process, and a rigorous analysis on the degree distribution and diameter of this model was studied by Bollabas and Riordan [26]. Further, Dommers, Hofstad and Hooghiemstra [40] investigated the diameters of several variations of the preferential attachment model, proving that, for each variant, when the power law exponent exceeds 3, the diameter is $\Omega(\log n)$, and when the power law exponent is in $(2, 3)$, the diameter is $\Omega(\log \log n)$.

To our knowledge, there does not exist any prior work combining a preferential attachment model with Kleinberg's model. In terms of the most relevant prior work, Flaxman, Frieze, and Vera [49] introduce a random graph model that combines preferential attachment graphs with geometric random graphs, with points created randomly on a unit sphere one-at-a-time, such that for each added vertex, m neighbors that are within a fixed distance, r, of that vertex are chosen with probability proportional to their degrees. Flaxman, Friex, and Vera show that with high probability the vertex degrees in this model follow a power law assuming r is sufficiently large, and they prove that the diameter of this graph model is $O(\ln n/r)$ w.h.p., but they do not study its ability to support efficient greedy routing. Indeed, when $r \geq \pi/2$, this model is just the preferential attachment model.

3.1.2 Additional Prior Work

Ever since being popularized by Milgram's experiments and the subsequent work by other researchers on complex networks, the small-world phenomenon has found applications in a wide array of research fields, including rumor spreading, epidemics, electronic circuits, wireless networks, the World Wide Web, network neuroscience, and biological networks. For an overview of the small-world phenomenon and its applications, the reader can refer to [114].

Incidentally, and not surprisingly, there has been a significant amount of additional prior work that analyzes the small-world phenomenon on different types of social network models, e.g., see [83, 102, 81]). Liben-Nowell, Novak, Kumar, Raghavan, and Tomkins [83] introduce a geographic social network model, which uses *rank-based friendships*, where the probability of assigning long-range connections from any person u to person v is inversely proportional to the number of people in the network who are geographically closer to u than v. The social network is modeled based on a 2D grid representation of the surface of earth, where each grid point has a positive population value, and has local connections to its immediate neighbors on the grid. Each grid point is then connected to a fifth neighbor based on their rank. Liben-Nowell *et al.* prove an upper bound of $O(\log^3 n)$ for the expected hop length of paths formed by this model, which, of course, is worse than the expected $O(\log^2 n)$ hop lengths in Kleinberg's model.

Kleinberg's model and its extensions have also been studied extensively. Martel and Nguyen [86] proved the expected diameter of the resulting graph is $\Omega(\log n)$, but that a greedy routing strategy cannot find such short paths, as they show that Kleinberg's $O(\log^2)$ analysis for greedy routing is tight. They extend Kleinberg's model by assuming each vertex has some additional (unrealistic) knowledge of the network. For example, they show that when each node u knows the long-range contacts of the $\log n$ nodes closest to u in the grid, the expected number of hops is $O(\log^{3/2} n)$. Fraigniaud, Gavoille and Paul [50] provide a similar extension, and they prove a bound of $O(\log^{1+1/d} n))$ expected hops in the general $d-$dimensional mesh, and show that this bound is tight for a variety of greedy algorithms, including those that have global knowledge of the network.

3.1.3 Our Contributions

In this chapter, we study the small-world phenomenon with road networks, which is motivated by the fact that, as mentioned above, the network of connections in the original small-world experiments were as much geographic as they were social [88, 107]. We introduce a new small-world model, which we call the *Neighborhood Preferential Attachment* model, which blends elements from the preferential attachment model of Barabási and Albert [18] and Kleinberg's model [79], but with underlying distances defined by a road network rather than a square grid.

In a nutshell, our model generates a random social network starting from a road network. We add the vertices to our model one-at-a-time at random from the vertices of the underlying road network (whose vertices stand in as the participants in our social network). When we add a new vertex, v, to our model, we create a fixed contant number, $m \geq 1$, of additional edges from v to existing vertices, with each other neighbor, w, chosen with a probability proportional to the ratio of the current degree of w (counting just the added edges) and $d(v, w)^2$, where $d(v, w)$ is the distance from v to w in the road network.

By using the constant, m, as parameter, we guarantee that the average degree in the network is a constant, which matches another observation made by Milgram for his experiments [88]. Interestingly, researchers have observed that an upper bound of $O(\log n)$ on the expected hop length in Kleinberg's model can be achieved by having an unrealistic $O(\log n)$ outgoing links for *every* vertex instead of a small constant, e.g., see [86]. Thus, our model tests whether short paths can be found using greedy routing in a social network with constant average degree, but with a few vertices having degrees higher than this, as was the case for the few "key" individuals, Jacobs, Jones, and Brown, in an original small-world experiment [88].

One of the main goals in our design of the Neighborhood Preferential Attachment model is to introduce a model that brings the average hop length for greedy routing closer to

the six degrees-of-separation found in the original small-world experiments, while keeping the average degree of the network bounded by a constant. To test this, we experimentally evaluate instances of our model using road networks for various U.S. states. We empirically compare the performance of greedy routing in our model to the performance for a variant of Kleinberg's model, where links are chosen with probability proportional to the inverse squared road-network distances of vertices (rather than a grid), as well as with the well-known Barabási-Albert preferential-attachment model. Interestingly, our experiments show that the Neighborhood Preferential Attachment model outperforms both the Barabási-Albert preferential-attachment model and the road-network variant of Kleinberg's model. Moreover, our experimental results show that our model has a scale-free degree distribution, which is arguably a better representation of real-world social networks than Kleinberg's model while also being geographic, unlike the preferential-attachment model of Barabási and Albert.

3.2 Preliminaries

We view road networks as undirected, weighted, and connected graphs, where each vertex corresponds to a road junction or terminus, and each edge corresponds to road segments that connect two vertices. In our social network model, each junction or terminus in a road network represents a single person, and each road segment represents a social connection between two people, which we consider to be the local connections of the network. Intuitively, our social network model can be seen as a mapping of each person in the population to the road network vertex that is geographically closest to their address. Likewise, an edge (u, v) in the road network represents the existence of social connections between people who were mapped to vertices u and v. This is admittedly an approximation for a population distribution, but we feel it is reasonable for most geographic regions, since population density

correlates with road-network density, e.g., see [30, 19, 71]. Certainly, it is is more realistic than modeling population density using a uniform $n \times n$ grid, as in Kleinberg's model [79].

The distance between two vertices $u, v \in V$ is denoted as $d(u, v)$ and is the total weight of the shortest path between u and v in the underlying road network. The hop distance between two vertices is denoted as $d_h(u, v)$ and is the minimum number of hops required to reach v from u, without considering edge weights and including both road-network edges and additional edges added during model formation. In all of the social network models we mention in this chapter, we assume all edges are undirected for the sake of distance computations, which reflects the notion that friendships are bidirectional.

We define $\deg_G(v)$ to be the degree of v in a graph, $G = (V, E)$, that is, the number of v's adjacent vertices in G. If G is understood from context, then we may drop the subscript.

3.3 The Road-Network Kleinberg Model

In this section, we introduce a variant of Kleinberg's small-world model adapted so that it works with weighted road networks rather than $n \times n$ grids. We denote this model throughout this chapter as the *KL model*. Interestingly, as we show in our empirical analysis, although this model is not as effective for performing greedy routing as our Neighborhood Preferential Attachment model, it nevertheless is much more efficient in practice than the theoretical analysis of Kleinberg [79] that is based on using $n \times n$ grids would predict.

As mentioned above, Kleinberg's network model begins by defining a set of vertices as the lattice points in an $n \times n$ grid, i.e., $\{(i, j) \mid i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots n\}\}$, so that the distance between any two vertices $u = (i, j)$ and $v = (k, l)$ is the Manhattan distance, $d(u, v) = |k - i| + |l - j|$. Each vertex, u, has an edge to every vertex within distance $p \geq 1$, called the *local contacts* (typically, we just take $p = 1$, so these are just grid-neighbor

connections), and each vertex has edges to $m \geq 1$ other vertices selected at random, called the *long-range contacts*, such that the probability that there exists an edge from u to v is $d(u,v)^{-s}/z$, where $s \geq 0$ is called the *clustering exponent* and z is a normalizing factor that ensures we have a probability distribution. Then, a decentralized *greedy algorithm* is used to route messages between a source and target vertex as follows: at each step, the current message holder forwards its message to a contact that has the smallest Manhattan distance to the target vertex.

We now adapt this model to the KL model that works on weighted road networks. We start with the set of vertices and edges of a road network, where each edge corresponds to a *local* connection. Then, for each vertex, u, we add $m \geq 1$ long-range edges randomly, where the probability that there exists a long-range connection between u and a vertex, v, is $d(u,v)^{-s}/z$, where $d(u,v)$ is the road-network distance between u and v (in miles or kilometers), $s \geq 0$ is the clustering exponent, and z is a normalizing factor that ensures we have a probability distribution. See Algorithm 3, noting that we call it for a road network, $G = (V, E)$, and parameter, $m \geq 1$, for the number of long-range connections to add for each vertex.

Algorithm 3: CONSTRUCT-KL(V, E, s, m)

$E' \leftarrow \emptyset$
for *each* $v \in V$ **do**
  $P \leftarrow \{1/d(v,u)^s \mid u \in V, u \neq v\}$
  $z_v \leftarrow \sum_{p \in P} p$
  Normalize P by dividing each $p \in P$ by z_v
  $S \leftarrow$ sample m vertices according to their probabilities in P
  $E' \leftarrow E' \cup \{(v,w) \mid w \in S\}$
return $G = (V, E \cup E')$

For his original model (on an $n \times n$ grid), Kleinberg [79] showed that the optimal value for the clustering exponent s is 2, for which the decentralized greedy routing algorithm is able to find paths of length $O(\log^2 n)$ in expectation, and that for any other value of $s \neq 2$, the

greedy algorithm would only be able to find a path with length that is lower bounded by a polynomial in $|V|$. Following Kleinberg, we usually select $s = 2$ for the weighted road-network variant, KL, of this model, as well as for the Neighborhood Preferential Attachment model, and we include some experiments that show the effect of varying this parameter for the latter model on different road networks.

In the routing algorithm for the KL model, we use a weighted version of the decentralized greedy algorithm, such that at each step, the current message holder forwards its message to a directly adjacent contact in the social network that has the smallest road-network distance to the target vertex (which could have easily been estimated in the 1960s using a road atlas of the United States and which can be determined in modern times from any navigation app, such as Google Maps, OpenStreetMap, Apple Maps, or Waze). We denote this greedy algorithm as Weighted−Decentralized−Routing.

3.4 A Road-Network Preferential Attachment Model

In this section, we give a brief description of the preferential-attachment model; see, e.g., [89, 18, 40, 26]. This model is defined by an algorithm to generate random graphs whose degree distribution follows a power law. The algorithm is based on a preferential attachment mechanism, where vertices with larger degrees are more likely to receive new links.

The algorithm for building an instance of the preferential-attachment model starts with a set, V, of n vertices, and an initial clique of $m + 1$ vertices from V.[2] It then selects the remaining vertices from V in random order, with each vertex, v, getting connected to m existing vertices, where the probability that v connects to vertex u is proportional to u's degree at the time v is added. In the case of $m \geq 2$, edges for a particular vertex are added through independent trials, i.e., previous edges do not affect the degree counts when choosing

[2]There are other variations for the starting "seed" graph, but the results in the limit are similar [89].

later edges for the same vertex. The algorithm stops when it has constructed a graph with n vertices. Note that the number of added edges is exactly nm. See Algorithm 4.

Algorithm 4: CONSTRUCT-BA(V, E, m)

```
Select subset M ⊆ V of size m + 1 by sampling vertices u.a.r.
E' ← {(u, v) | u, v ∈ M, u ≠ v}
for each v ∈ V \ M in random order do
    P ← {deg_{G'}(u) | u ∈ V, u ≠ v}, where G' = (V, E')
    z_v ← Σ_{p∈P} p
    Normalize P by dividing each p ∈ P by z_v
    S ← sample m vertices according to their probabilities in P
    E' ← E' ∪ {(v, w) | w ∈ S}
return G = (V, E ∪ E')
```

Although the preferential attachment model is defined as a non-geographic model, if the vertices in the model have geographic coordinates, such as determined in a road network, we can nevertheless apply the same distributed greedy routing algorithm as for the KL model. Specifically, if we take the set of candidate vertices in the preferential attachment model to be vertices in a road network and we union the edges of the final preferential attachment model with the edges of the road network for the corresponding vertices (as shown in Algorithm 4), then we can construct an instance of a preferential-attachment graph embedded in a road network. This allows each participant to forward their message to a direct contact (including both added edges and road-network edges) that is closest to the target (using road-network distance). Indeed, for our experiments, this is what we refer to as the *BA model*.

3.5 The Neighborhood Preferential Attachment Model

We now introduce our *Neighborhood Preferential Attachment* (NPA) model. We start with the same set of local connections as for the road-network Kleinberg model, KL, except now we distribute long-range connections according to a combination of vertex degrees and road-network distances between vertices. Thus, our model combines elements of the KL and BA

models. Surprisingly, as we show below, rather than achieving a performance somewhere between the KL and BA models, our NPA model outperforms both the KL model and BA model.

To generate the network of long-range connections, we consider the vertices in random order, adding new (long-range) edges, based on degrees, distances, and an input parameter, $m \geq 1$. Let $G = (V, E)$ be a road network of n vertices. We begin by selecting a subset, $M \subseteq V$, of $m + 1$ vertices from G and we add all possible edges between them, so that every initial vertex has an initial degree equal to m. That is, we start by forming a clique of size $m + 1$ of randomly chosen vertices from V. We then repeatedly randomly consider the remaining vertices from V, until we have considered all the vertices from V. When we process a vertex, v, we connect v to m other vertices, where the probability that there is an edge between a new vertex v and another vertex u is proportional to the ratio $\frac{\deg(u)}{d(v,u)^s}$, normalized by normalizing factor,

$$z_v = \sum_{w \neq v} \frac{\deg(w)}{d(v, w)^s},$$

for v, such that $\deg(v)$ is the degree of vertex v considering only added edges and $d(v, u)$ is road-network distance. Typically, we choose $s = 2$. When $m \geq 2$, edges for a particular vertex are added through independent trials. See Algorithm 5 and Figure 3.3.

Algorithm 5: Construct-NPA(V, E, s, m)

```
Select subset M ⊆ V of size m + 1 by sampling vertices u.a.r.
E' ← {(u, v) | u, v ∈ M, u ≠ v}
foreach v ∈ V \ M in random order do
    P ← {deg_{G'}(u)/d(v, u)^s | u ∈ V, u ≠ v}, where G' = (V, E')
    z_v ← Σ_{p∈P} p
    Normalize P by dividing each p ∈ P by z_v
    S ← sample m vertices according to their probabilities in P
    E' ← E' ∪ {(v, w) | w ∈ S}
return G = (V, E ∪ E')
```

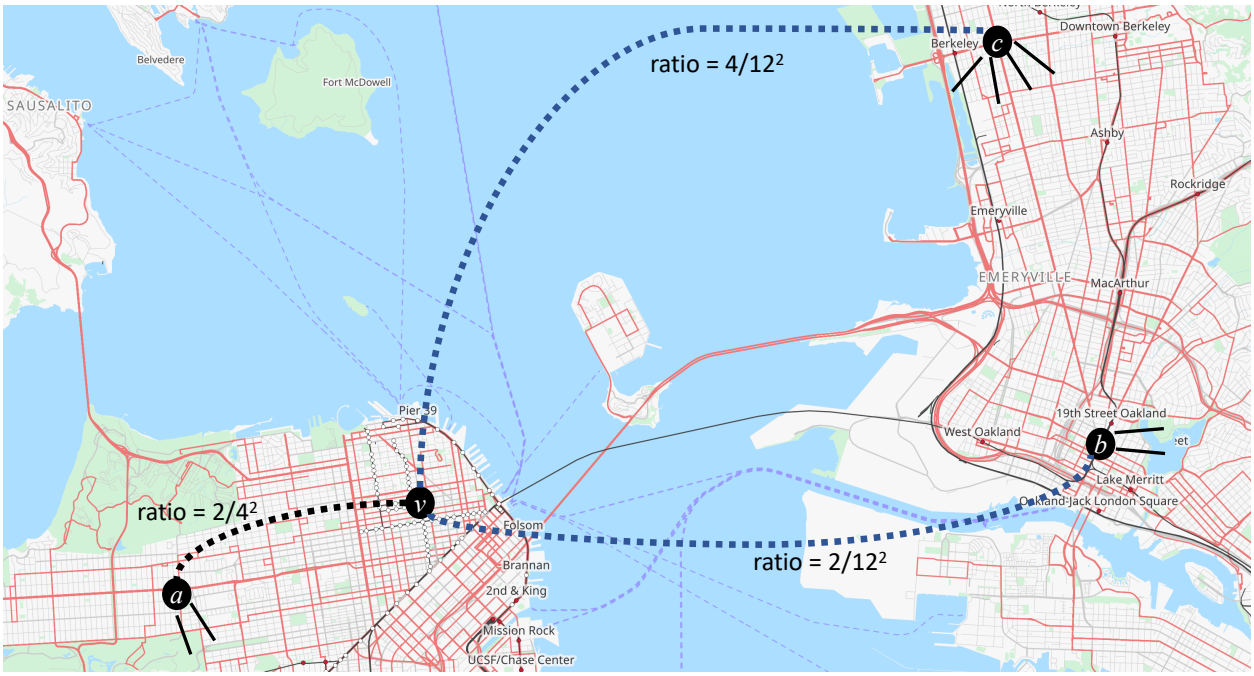


Figure 3.3: How edges are chosen in the Neighborhood Preferential Attachment model, illustrated with the road network of San Francisco, Berkeley, and Oakland. When vertex v is added, the ratio for the probability for a is $2/16(= 0.125)$, the ratio for the probability for b is $2/144(= 0.014)$, the ratio for the probability for c is $4/144(= 0.028)$. Thus, even though b and c are the same distance from v, c is twice as likely as b to be chosen, and a is 4.5 times more likely to be chosen than c, because c's degree of 4 is twice that of b or a, but a's squared distance is 9 times smaller than that of b and c. (Background image is from OpenStreetMap and is licensed under the Open Data Commons Open Database License (ODbL) by the OpenStreetMap Foundation (OSMF).)

Once the model-construction is finished, we add the local road-network connections back in. Since we add m edges for each vertex in the network, and since road networks themselves have a constant maximum degree, the average degree for our network model is a constant when m is a constant. We refer to this as the *NPA model*. For the routing phase, we run the same decentralized greedy routing algorithm for the NPA model as for the KL and BA models.

3.6 Experimental Analysis

Intuitively, the BA model tries to capture how popularity is often distributed according to a power law, with the "rich getting richer" as more people are added to a group, but it completely ignores geography in forming friendship connections. That is, in the BA model, if there is a popular person, u, in New York and an equally popular person, w, in Los Angeles, a newly-added person, v, in San Diego is just as likely to form a long-range connection to u as to w.

The KL model, on the other hand, tries to capture how friendship is correlated with geographic distance, but it completely ignores popularity. That is, in the KL model, if there is a popular person, u, in Hollywood and an unpopular person, w, who is also in Hollywood, a newly-added person, v, in San Diego is just as likely to form a long-range connection to u as to w.

In contrast to both of these extremes, as illustrated above in Figure 3.3, our NPA model tries to capture how friendship is correlated with both popularity and geographic distance. That is, in the NPA model, if there is a popular person, u, in New York and an equally popular person, w, in Los Angeles, a newly-added person, v, in San Diego is more likely to form a long-range connection to w than to u. Furthermore, if there is a popular person, u, in Hollywood and an unpopular person, w, who is also in Hollywood, a newly-added person, v, in San Diego is more likely to form a long-range connection to u than to w.

Intuition aside, however, we are interested in this chapter in determining how effective the BA, KL, and NPA models are at greedy routing. For example, which of these models is the best at greedy routing and can any of them achieve the six-degrees-of-separation phenomenon shown in the original small-world experiments [88, 107]?

3.6.1 Experimental Framework

To answer the above question, we implemented the BA, KL and NPA models in C++ (using an open-source routing library [38] to find shortest paths), randomly sampled 1000 source/target pairs, then ran Weighted—Decentralized—Routing on each pair and measured the average hop length. The datasets we used are road networks for 50 U.S. states and Washington, D.C., obtained from the formatted TIGER/Line dataset available from the 9th DIMACS Implementation Challenge website.[3] For each road network, only the largest connected component was considered. The sizes of the road networks we used range from 9,522 to 2,037,156 vertices. As a preprocessing step, we normalized edge weights so that the smallest edge weight is 1.

3.6.2 Hop Counts with Few Long-Range Links

The first set of experiments that we performed was to test the effectiveness of each of the three models on each road-network data set assuming that we add only a small number of long-range links. In particular, we tested each model for the cases when $m = 1, 2, 3, 4$. We show the results of these experiments in Figure 3.4, which show that the NPA model outperforms both the KL and BA models for each of these small values for m. For example, even for $m = 1$, the number of hops for the NPA model tends to be half the numbers for the BA and KL models. Once $m \geq 2$, the KL model shows improved performance over the BA model, with the KL model achieving degrees-of-separation values that are roughly half those for the BA model. Nevertheless, for $m \geq 2$, the NPA model still beats the KL model, with hop-counts that are between a third and a half better than the KL model. Further, as would be expected, all the models tend to do better as we increase the value of m. For example, when $m = 1$, the NPA model achieves a degrees-of-separation value of between 40 and 60,

[3] `http://www.diag.uniroma1.it/~challenge9/data/tiger/`

whereas when we increase m to just 4, the NPA model achieves a degrees-of-separation value of between 10 and 20. Admittedly, this still isn't 6, but it is getting closer, and it shows what can be achieved with just a few added long-range links.

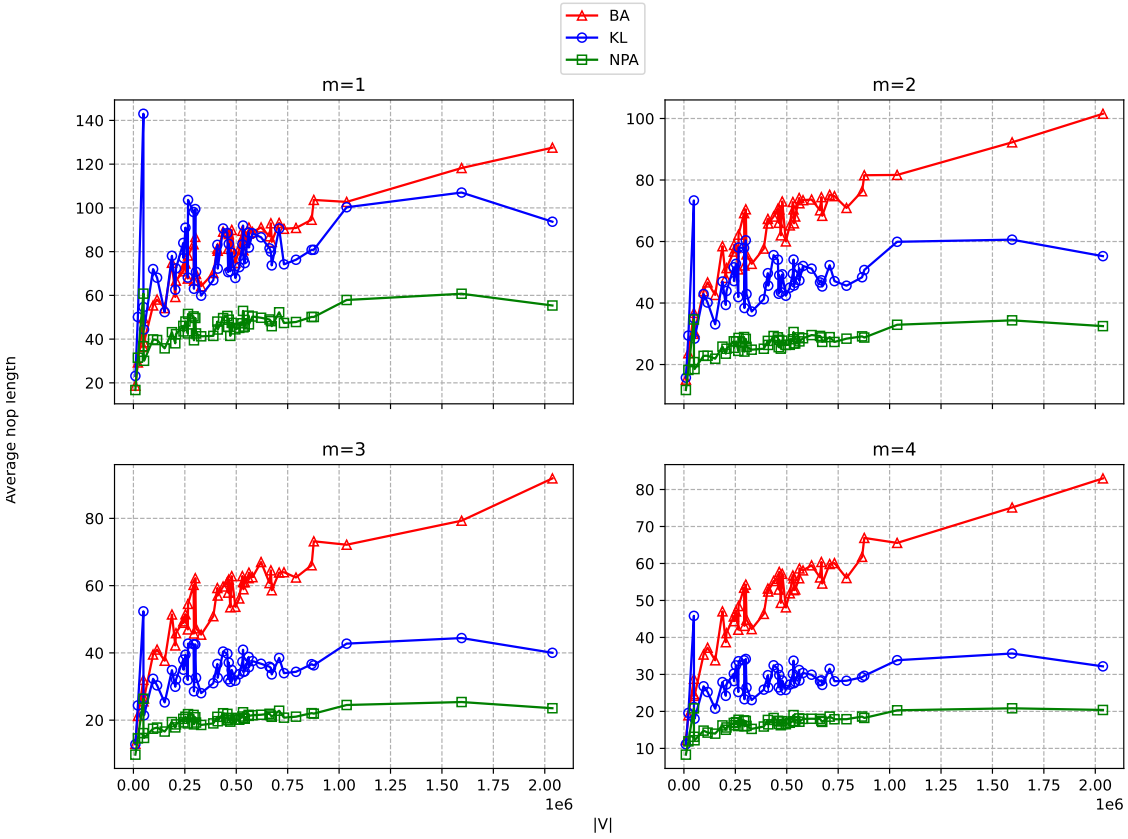


Figure 3.4: Average hop lengths over 1000 runs of Weighted—Decentralized—Routing for 50 U.S. states and Washington, DC.

3.6.3 Dropouts

There is another aspect of the original small-world experiments, which (like most prior research on the small-world phenomenon) we have heretofore ignored. Namely, as participants perform greedy routing in the real world there is a probably that someone will simply drop out of the experiment and not forward the package to anyone. For example, in one of the original small-world experiments [107], Travers and Milgram observed a dropout probability

of roughly $p = 0.2$ at each step in a routing operation. That is, in the original small-world experiment, it was observed that some amount of messages never ended up reaching the target person, e.g., due to recipients refusing to participate or not having anyone to forward the message to. The longer a source-to-target path gets, the more likely it is that at least one person will drop the message, so we expect that the average path length would decrease as the probability of dropping messages increases. To see whether this could have contributed to the small average hop length observed in the original small-world experiment, we ran a variant of Weighted−Decentralized−Routing on the KL and NPA models, such that each message holder has a fixed probability p of dropping the message. Our results can be seen in Figure 3.5, for $m = 4$. As expected, these experiments show that the average hop counts for successful paths decrease as we increase the dropout probability, p, but we still are not quite achieving six degrees of separation for these values.

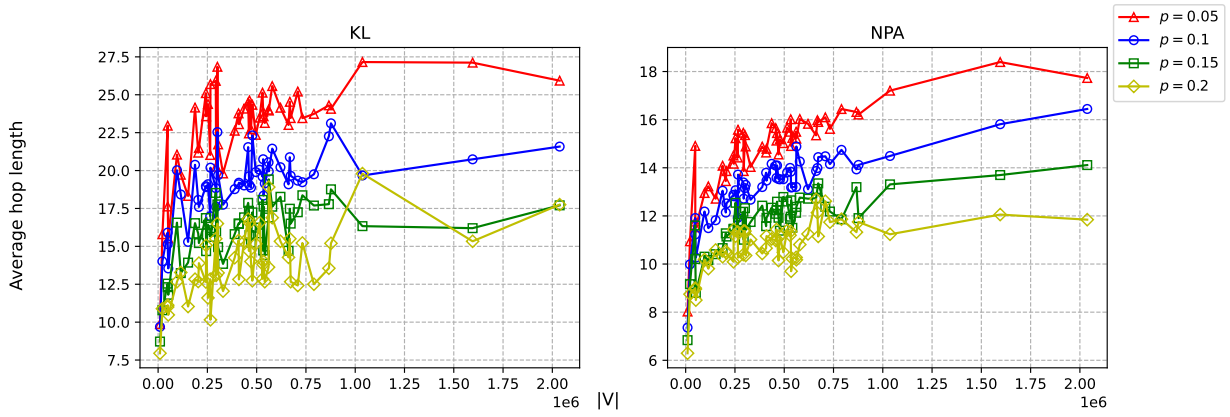


Figure 3.5: Effect of varying the probability p of dropping the message at each step during Weighted−Decentralized−Routing for the KL and NPA models, with $m = 4$.

3.6.4 Six Degrees of Separation

We can, in fact, achieve six degrees of separation in the NPA model, just by slightly increasing the value of m. In particular, we provide experimental results in Figure 3.6 for the NPA model with $m = 30$ with different dropout probabilities. As this result shows, even with $p = 0$

(no dropouts), we can achieve 7 degrees of separation for modestly sized road networks (and 8 degrees of separation for the three largest road networks). With $p = 0.2$, for the majority of road networks, we get average hop counts that match the findings in the original small-world experiments, where the average hop length was found to be 6. For the largest road networks, we get average hop counts that are between 6 and 7.

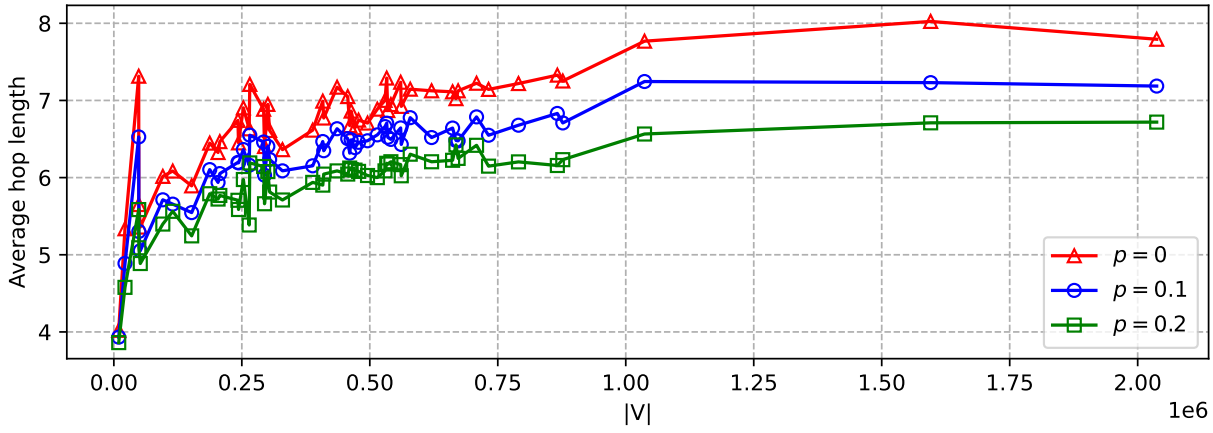


Figure 3.6: Average hop length of the NPA model with $m = 30$ for different dropout probabilities.

Intuitively, setting $m = 30$ is equivalent to assuming that people participating in a small-world experiment would consult their address books when deciding who to send a package to next and that the average number of entries in each address book is 30, which we feel is a reasonable assumption.

3.7 Diving Deeper

We are actually interested in more than just showing that the NPA model can achive six degrees of separation and thereby match the performance of the original small-world experiments. In this section, we take a deeper dive into the models we introduce in this chapter, with an eye towards trying to better understand what is going on during the greedy routing done in each model.

3.7.1 Degree Distributions

Comparing the degree distributions of the three models, which are shown in Figure 3.7, we see that the KL model has a light-tailed distribution, whereas our model seems to be scale-free, similar to the BA model. These results indicate that the NPA model, similar to the KL model, is able to utilize local clustering when finding long-range contacts, while still having the scale-free property.

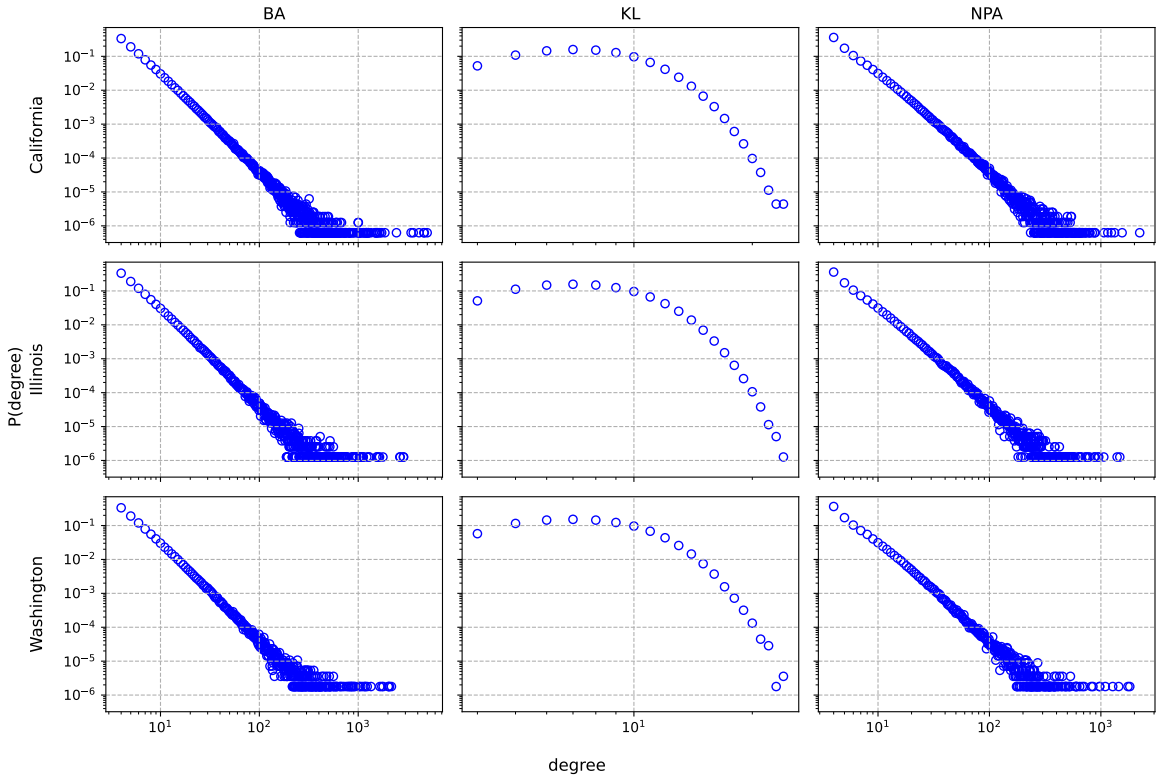


Figure 3.7: Degree distributions of the three main models with $m = 4$ on road networks of different sizes.

3.7.2 How Distances to the Target Decrease

As shown above, we observe that the NPA model outperforms both of the KL and BA models in terms of the average hop length. We also see that the KL model performs significantly better than Kleinberg's theoretical upper bound [79] on the grid, which was $c \log^2 n$ for

$c > 88$. Still, Kleinberg's theoretical analysis was based on an interesting proof technique that was inspired from Milgram's figure showing how distances to the target tend to halve with each hop, as shown above in Figure 3.1. At a high level, Kleinberg's proof for his $O(\log^2 n)$ bound is based on finding that the probability that the distance from the current vertex to the target is halved at any step is $\Theta(1/\log n)$; hence, this is a constant after $\Theta(\log n)$ hops, and we can reach the target by repeating this argument $O(\log n)$ times.

We provide experimental results in Figure 3.8 showing how the remaining distance to the target changes for the NPA model over multiple runs of Weighted−Decentralized−Routing. We see that for most runs, the distance typically gets halved every few steps, as Milgram observed.

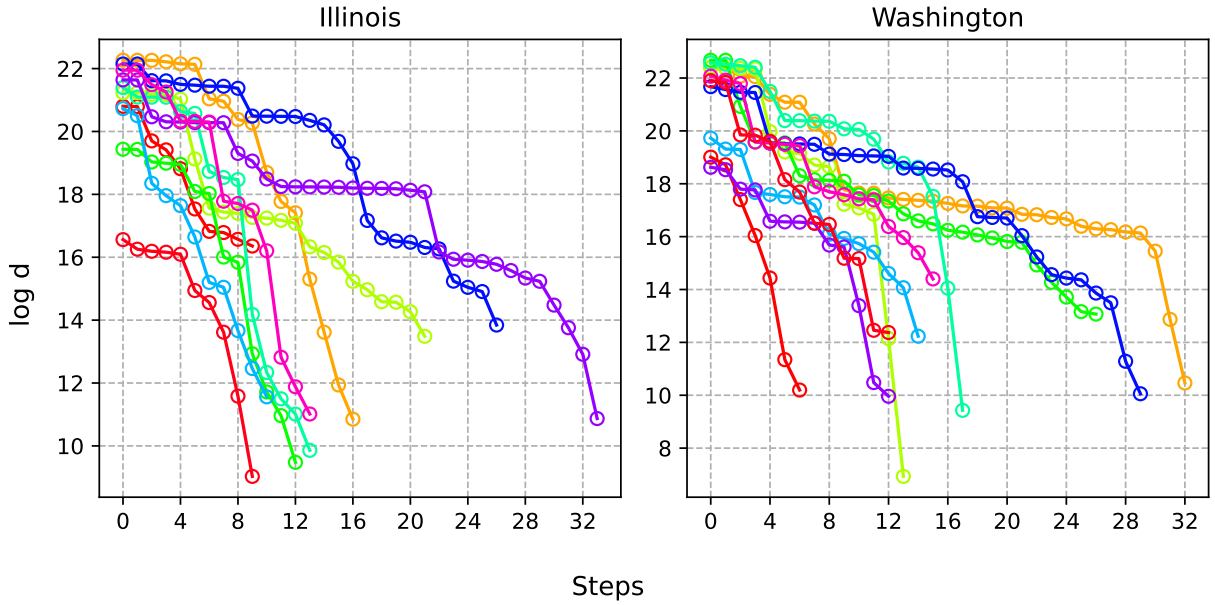


Figure 3.8: Remaining distance to target, denoted as d, during 10 runs of Weighted-Decentralized-Routing on two road networks, with $m = 4$. Each line corresponds to a separate run of Weighted-Decentralized-Routing, with the markers on each line corresponding to the remaining distance at a particular step. The last data point for each run corresponds to the penultimate step, i.e. when the message holder is one hop away from the target.

3.7.3 Varying the Clustering Coefficient

In Figure 3.9, we see how varying the clustering coefficient affects the average hop length in the NPA model for the HI and CA road networks. Though $s = 2$ is not the best-performing clustering exponent for either road network in our experiments, the results indicate that the best-performing clustering exponent seems to move towards 2 when the input size gets larger, which suggests that the asymptotically optimal clustering exponent could still be 2. A similar effect could be observed in Kleinberg's original model as well, since the lower bounds that are proved for $s \neq 2$ are $\Omega(n^{(2-s)/3})$ for $s < 2$ and $\Omega(n^{(s-2)/(s-1)})$ for $s > 2$, both of which require input sizes that are orders of magnitude larger than real-world road networks to be able to experimentally observe the optimality of $s = 2$.

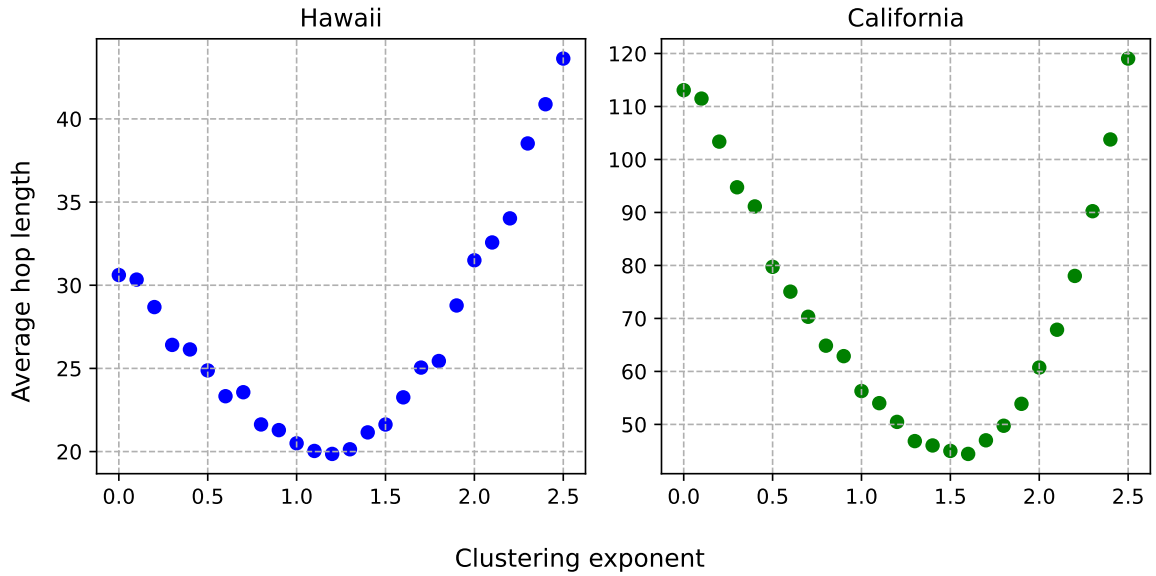


Figure 3.9: Effect of varying the clustering coefficient on the average hop length in the NPA model for the road networks of Hawaii ($|V| = 21\,774$) and California ($|V| = 1\,595\,577$), for $m = 1$.

3.7.4 Capping the Maximum Degree

We considered another variation of the NPA model, where we cap the maximum degree such that only vertices of degree less than c are considered when choosing long-range contacts.

We call this the *NPA-cap* model. We choose $c = \log n$ and $c = 150$ as possible maximum degree caps. Intuitively, the cap on the maximum degree is like a cap on the size of someone's address book during a small-world experiment. We provide experimental results comparing the models KL, NPA, and NPA-cap (for $c = \log n$ and $c = 150$), with $m = 4$, in Figure 3.10.

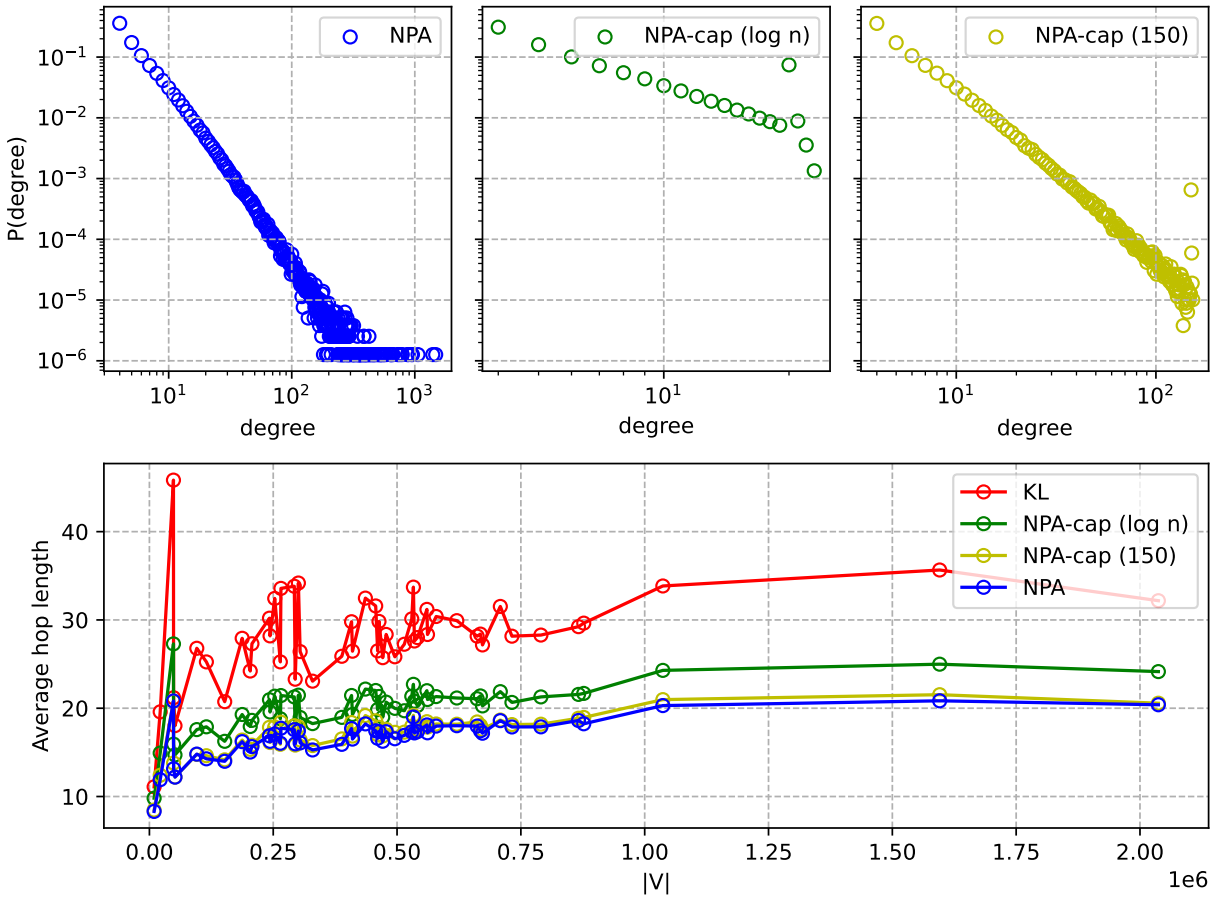


Figure 3.10: Comparing the average hop lengths of the NPA, KL, and the NPA-cap models, and the degree distribution of the NPA and NPA-cap models for Illinois, with $m = 4$.

In Figure 3.11, we compare the models NPA and NPA-cap (for $c = 150$), when there is a dropout probability of $p = 0.2$, with $m = 30$.

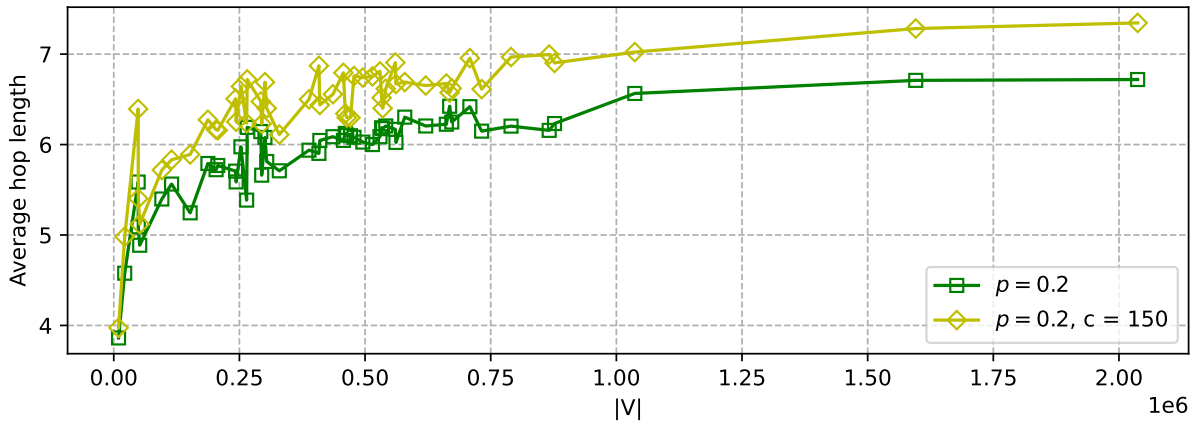


Figure 3.11: Comparing the average hop lengths of the NPA and NPA-cap (150) models with a dropout probability of 0.2 and $m = 30$.

3.7.5 Routing Across Multiple States

The experiments we have performed so far have been limited to the road networks of individual states. However, Milgram's small-world experiments were performed across multiple states. For this reason, we also performed experiments on the combined road networks of Virginia, Washington, D.C., Maryland, Delaware, New Jersey, New York, Connecticut, and Massachusetts.

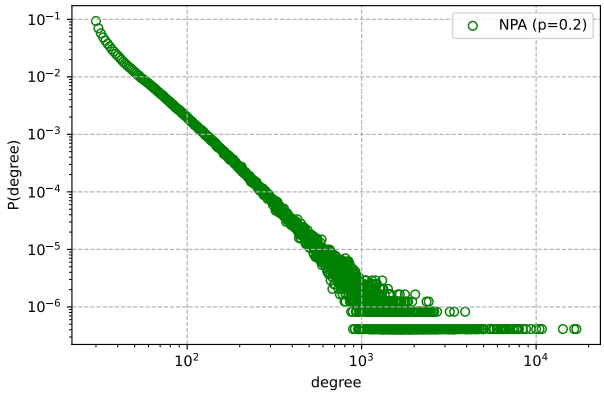


Figure 3.12: Degree distribution in the multi-state road network, using the NPA model with a dropout probability of 0.2 and $m = 30$.

For $m = 30$, we found that the average hop length was ≈ 8.06, and when we introduced a dropout probability of $p = 0.2$, the average hop length was ≈ 7.15. In Figure 3.12, we provide the resulting degree distribution of this road network when the NPA model with a dropout of $p = 0.2$ was used.

3.7.6 Key Participants

We also considered the importance of key participants in performing greedy routing, as shown in Figure 3.2, which motivated the NPA model in the first place. Having a long-tailed degree distribution could be benefiting the routing phase, as we know that having more links per vertex improves the asymptotic bound of Kleinberg's model. In Figure 3.13, we compare the degree distribution of vertices that were used during the routing phase with the degree distribution of the whole network for both the NPA and BA models. We can see that for the NPA model, high-degree vertices are being better utilized during an instance of the routing algorithm compared to the BA model.

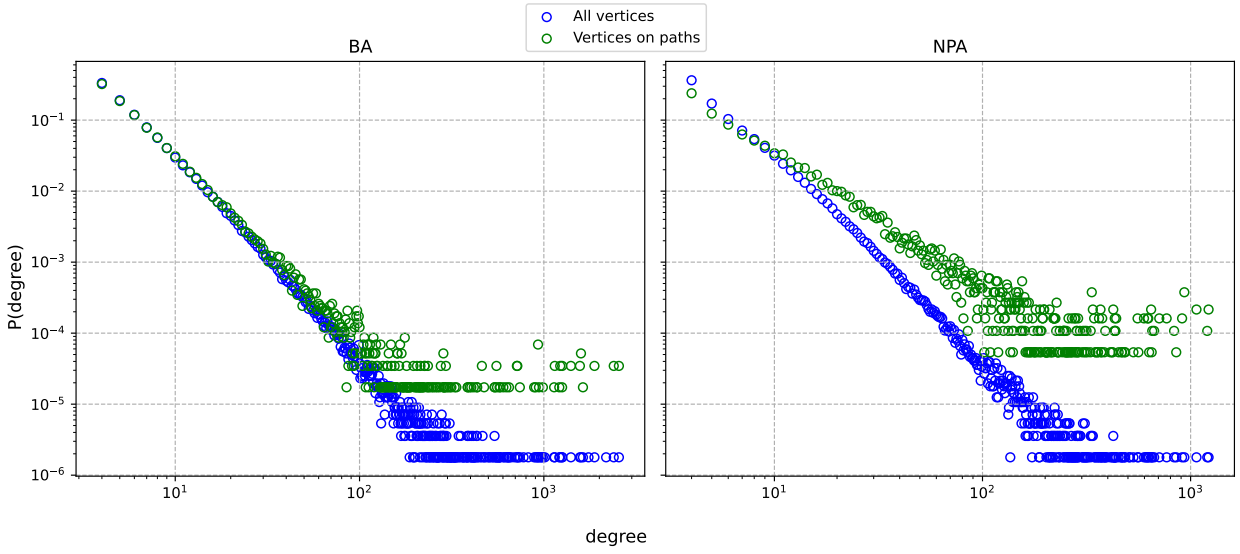


Figure 3.13: Degree distributions in the Washington road network for vertices in the whole network, and vertices visited during the routing phase, using the BA and NPA models with $m = 4$.

3.8 Conclusion

We introduced a new small world model, the Neighborhood Preferential Attachment model, which combines elements of both Kleinberg's model and the Barabási-Albert model, and experimentally outperforms both models in terms of the average hop length. Importantly, our model is built using real-world distances from nodes in a road network rather than vertices in a square grid or random points on a sphere.

3.8.1 Future Work

For future work, given our experimental results, it would be interesting to perform a mathematical analysis of our model, e.g., to see whether our model has an asymptotic bound on the expected hop length that is $o(\log^2 n)$. Another interesting question is whether the power law exponent of the degree distribution differs from the Barabási-Albert model in the limit of the size of the network, or what the diameter of graphs generated by our model is. Yet another interesting problem is whether Kleinberg's lower bounds for the standard model when the clustering coefficient is $\neq 2$ still holds for our model.

Chapter 4

Data Oblivious Noisy Sorting

4.1 Introduction

Given an array, A, of n distinct comparable elements, we study the problem of efficiently sorting A subject to noisy probabilistic comparisons. In this framework, which has been extensively studied [27, 85, 78, 54, 56, 55, 46, 91, 73, 112, 76, 82], the comparison of two elements, x and y, results in a true result independently according to a fixed probability, and otherwise returns the opposite (false) result. In the case of *persistent* errors [27, 78, 54, 56, 55], the result of a comparison of two given elements, x and y, always returns the same result. In the case of *non-persistent* errors [46, 91, 73, 112, 76, 82], however, the probabilistic determination of correctness is determined independently for each comparison, even if it is for a pair of elements, (x, y), that were previously compared.

Motivation for sorting with comparison errors comes from multiple sources, including applied cryptography scenarios where cryptographic comparison protocols can fail with known probabilities (see, e.g., [84, 48, 117]). In such cases, reducing the noise from comparison errors can be computationally expensive, and the framework advanced in this chapter offers

an alternative, possibly more efficient approach, where a higher error rate is tolerated while still achieving the ultimate goal of sorting or near-sorting with high probability. Further, other applications of sorting with comparison errors include ranking objects in online forums via group A/B testing [116].

Since it is not possible to always correctly sort an array, A, subject to persistent comparison errors, we follow the formulation of Geissmann *et al.* [55, 54, 56], and define the *dislocation* of an element, x, in an arary, A, as the absolute value of the difference between x's index in A and its index in the correctly sorted permutation of A. Further, define the *maximum dislocation* of A as the maximum dislocation for the elements in A, and let the *total dislocation* of A be the sum of the dislocations of the elements in A. By known lower bounds [55, 54, 56], the best a sorting algorithm can achieve under persistent comparison errors is a maximum dislocation of $O(\log n)$ and a total dislocation of $O(n)$. Thus, coming close to such asymptotic maximum and total dislocation guarantees should be the goal for a sorting algorithm in the presence of persistent comparison errors.

Given the cryptographic applications of noisy comparisons, we desire sorting algorithms that are *data oblivious*, which support privacy-preserving cryptographic protocols. A sorting algorithm is data oblivious if its memory access pattern does not reveal any information about the data values being sorted. Unfortunately, existing efficient algorithms for sorting with noisy comparisons are not data oblivious. Indeed, they all make use of noisy binary search [55], which is a data-sensitive random walk in a binary search tree, e.g., see Geissmann, Leucci, Liu, and Penna [55], Feige, Raghavan, Peleg, and Upfal [46], and Leighton, Ma, and Plaxton [82]. Instead, we desire efficient sorting algorithms that tolerate noisy comparisons and avoid the use of noisy binary search, so as to be data oblivious (i.e., privacy preserving if comparisons are done according to a data-hiding protocol).

Related Prior Results. Problems involving probabilistic comparison errors can trace their roots back to a classic problem by Rényi [95] of playing a two-person game where player A poses yes/no questions to a player B who lies with a given probability; see a survey by Pelc [92]. Notable prior results include a paper by Pippenger [93] on computing Boolean functions with probabilistically noisy gates and work by Yao and Yao [118] on sorting networks built from noisy comparators. There is also considerable work on searching when the total number of faulty comparisons is bounded rather than considering probabilistic noisy comparisons, including the work by Kenyon-Mathieu and Yao [75] and Rivest, Meyer, Kleitman, Winklmann, and Spencer [97]. Also of note is work by Karp and Kleinberg [73], who study binary searching for a value $x \in [0, 1]$ in an array of biased coins ordered by their biases.

Braverman and Mossel [27] introduce a persistent-error model, where comparison errors are persistently wrong with a fixed probability, $p < 1/2 - \varepsilon$, and they achieve a sorting algorithm that in our framework runs in $O(n^{3+f(p)})$ time, where $f(p)$ is some function of p, with maximum expected dislocation $O(\log n)$ and total dislocation $O(n)$. Klein, Penninger, Sohler, and Woodruff [78] improve the running time to $O(n^2)$, but with $O(n \log n)$ total dislocation w.h.p. The running time for sorting in the persistent-error model optimally with respect to maximum and total dislocation was subsequently improved to $O(n^2)$, $O(n^{3/2})$, and ultimately to $O(n \log n)$, in a sequence of papers by Geissmann, Leucci, Liu, and Penna [55, 56, 54], all of which are not data oblivious because they make extensive use of noisy binary searching, which amounts to a random walk in a binary search tree.

Our Results. In this chapter, we provide data-oblivious sorting algorithms that tolerate persistent noisy comparisons. In addition, we empirically compare our algorithms to other sorting algorithms, including the worst-case optimal algorithm, Riffle sort, by Geissmann, Leucci, Liu, and Penna [55], which is not data oblivious, but it achieves an optimal maximum

and total dislocation under noisy comparisons. It runs in $O(n \log n)$ time, but it makes use of noisy binary search.

In addition to providing theoretical analysis for some of our algorithms, we empirically study all of our algorithms by measuring the effect of changing the amount of noise and the input size on the amount of dislocation, inversions, and number of comparisons. Our experiments show that for all of the data-oblivious algorithms we provide in this chapter, the maximum and total dislocations are comparable to the optimal bounds of $O(\log n)$ and $O(n)$ respectively for the best algorithms that are not data oblivious. Moreover, we include experimental results for some standard sorting algorithms such as insertion sort, quick sort, and shell sort, for which we provide emipirical evidence that all of our algorithms significantly outperform these other algorithms in terms of the maximum and total dislocation metrics. These results indicate that our algorithms are able to combine the properties of both having a good tolerance to noisy comparisons while also being data-oblivious.

4.2 Window-Sort

Our first sorting algorithm is a version of window-sort [54], which will be useful as a subroutine in our other algorithms. We describe the pseudo-code at a high level in Algorithm 6, for approximately sorting an array of size n that has maximum dislocation at most $d_1 \leq n$ so that it will have maximum dislocation at most $d_2 = d_1/2^k$, for some integer $k \geq 1$, with high probability as a function of d_2.

In addition to implementing window-sort data obliviously, we provide a new analysis of window-sort, which allows us to apply it in new contexts. We begin this new analysis with the following lemma, which establishes the progress made in each iteration of window-sort.

Algorithm 6: Window-Sort($A = \{a_0, a_1, \ldots, a_{n-1}\}, d_1, d_2$)

1 **for** $w \leftarrow 2d_1, d_1, d_1/2, \ldots, 2d_2$ **do**
2 **foreach** $i \leftarrow 0, 1, 2, \ldots, n-1$ **do**
3 $r_i \leftarrow \max\{0, i-w\} + |\{a_j < a_i \ : \ |j-i| \leq w\}|$
4 Sort A (deterministically) by nondecreasing r_i values (i.e., using r_i as the comparison key for a_i)
5 **return** A

Lemma 4.2.1. *Suppose the comparison error probability, p_e, is at most $1/16$. If an array, A, has maximum dislocation at most d' prior to an iteration of window-sort for $w = 2d'$ (line 1 of Algorithm 6), then after this iteration, A will have maximum dislocation at most $d'/2$ with probability at least $1 - n2^{-d'/8}$.*

Proof. Let a_i be an element in A. Let W denote the window of elements in A for which we perform comparisons with a_i in this iteration; hence, $2d' \leq |W| \leq 4d'$. Because A has maximum dislocation d', by assumption, there are no elements to the left (resp., right) of W that are greater than a_i (resp., less than a_i). Thus, a_i's dislocation after this iteration depends only on the comparisons between a_i and elements in its window. Let X be a random variable that represents a_i's dislocation after this iteration, and note that $X \leq Y$, where Y is the number of incorrect comparisons with a_i performed in this iteration. Note further that we can write Y as the sum of $|W|$ independent indicator random variables and that $\mu = E[Y] = p_e|W| \leq d'/4$. Thus, if we let $R = d'/2$, then $R \geq 2\mu$; hence, we can use a Chernoff bound as follows:

$$\Pr(X > d'/2) \leq \Pr(Y > d'/2) = \Pr(Y > R) \leq 2^{-R/4} = 2^{-d'/8}.$$

Thus, with the claimed probability, the maximum dislocation for all elements of A will be at most $d'/2$, by a union bound. □

This implies the following.

Theorem 4.2.2. *Suppose the comparison error probability, p_e, is at most $1/16$. If an array, A, of size n has maximum dislocation at most $d_1 \geq \log n$, then executing Window-Sort(A, d_1, d_2) runs in $O(d_1 n)$ time. Further, we can execute Window-Sort(A, d_1, d_2) data-obliviously to result in A having maximum dislocation of $d_2/2$ with probability at least $1 - 2n2^{-d_2/8}$, where $d_2 = d_1/2^k$, for some integer $k \geq 1$.*

Proof. For the running time and data obliviousness, note that we can perform the deterministic sorting step using a data-oblivious sorting algorithm (e.g., see [63]) in $O(n \log n)$ time. The windowed comparison steps (step 3 of Algorithm 6) are already data-oblivious and their running times form a geometric sum adding up to $O(d_1 n)$; hence, the total time for all the deterministic sorting steps (step 4 of Algorithm 6) is $O((\log(d_1/d_2))n \log n)$, which is at most $O(d_1 n)$ for $d_1 \geq \log n$.

For the maximum dislocation bound, note once $w = 2d_2$ and the array A prior to this iteration has maximum dislocation at most d_2, then it will result in having maximum dislocation at most $d_2/2$ with probability at least $1 - n2^{-d_2/8}$, by Lemma 4.2.1. Thus, by a union bound, the overall failure probability is at most

$$n\left(2^{-d_2/8} + 2^{-2d_2/8} + 2^{-4d_2/8} + \cdots + 2^{-d_1/8}\right) \quad < \quad n2^{-d_2/8} \sum_{i=0}^{\infty} 2^{-i} \quad = \quad 2n2^{-d_2/8}.$$

□

In terms of efficiency, we note that our data-oblivious implementation of window-sort is only time-efficient for small subarrays; hence, we need to do more work to design an efficient data-oblivious sorting algorithm.

4.3 Window-Merge-Sort

In this section, we describe a simple algorithm for sorting with noisy comparisons, which achieves a maximum dislocation of $O(\log n)$. Our window-merge-sort method is a windowed version of merge sort; hence, it is deterministic but not data oblivious. Nevertheless, it does avoid using noisy binary search.

Suppose we are given an array, A, of n elements (we use n to denote the original size of A, and N to denote the size of the subproblem we are currently working on recursively). Our method runs in $O(n \log^2 n)$ time and we give the pseudo-code for this method in Algorithm 7, with $d = c \log n$ for a constant $c \geq 1$ set in the analysis.

Algorithm 7: Window-Merge-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n, d$)

```
if N ≤ 6d then
    return Window-Sort(A, 4d, d)
Divide A into two subarrays, A_1 and A_2, of roughly equal size
Window-Merge-Sort(A_1, n, d)
Window-Merge-Sort(A_2, n, d)
Let B be an initially empty output list
while |A_1| + |A_2| > 6d do
    Let S_1 be the first min{3d, |A_1|} elements of A_1
    Let S_2 be the first min{3d, |A_2|} elements of A_2
    Let S ← S_1 ∪ S_2
    Window-Sort(S, 4d, d)
    Let B' be the first d elements of (the near-sorted) S
    Add B' to the end of B and remove the elements of B' from A_1 and A_2
Call Window-Sort(A_1 ∪ A_2, 4d, d) and add the output to the end of B
return B
```

Our method begins by checking if the current problem size, N, satisfies $N \leq 6d$, in which case we're done. Otherwise, if $N > 6d$, then we divide A into 2 subarrays, A_1 and A_2, of roughly equal size and recursively approximately sort each one. For the merge of the two sublists, A_1 and A_2, we inductively assume that A_1 and A_2 have maximum dislocation at most $3d/2 = (3c/2) \log n$. We then copy the first $3d$ elements of A_1 and the first $3d$ elements

of A_2 into a temporary array, S, and we note that, by our induction hypothesis, S contains the smallest $3d/2$ elements currently in A_1 and the smallest $3d/2$ elements currently in A_2. We then call Window-Sort$(S, 4d, d)$, and copy the first d elements from the output of this window-sort to the output of the merge, removing these same elements from A_1 and A_2. Then we repeat this merging process until we have at most $6d$ elements left in $A_1 \cup A_2$, in which case we call window-sort on the remaining elements and copy the result to the output of the merge. The following lemma establishes the correctness of this algorithm.

Lemma 4.3.1. *If A_1 and A_2 each have maximum dislocation at most $3d/2$, then the merge of A_1 and A_2 has maximum dislocation at most $3d/2$ with probability at least $1 - N2^{-d/8}$.*

Proof. By Lemma 4.2.1 and a union bound, each of the calls to window-sort performed during the merge of A_1 and A_2 will result in an output with maximum dislocation at most $d/2$, with at least the claimed probability. So, let us assume each of the calls to window-sort performed during the merge of A_1 and A_2 will result in an output with maximum dislocation at most $d/2$. Consider, then, merge step i, involving the i-th call to Window-Sort$(S, 4d, d)$, where S consists of the current first $3d$ elements in A_1 and the current first $3d$ elements in A_2, which, by assumption, contain the current smallest $3d/2$ elements in A_1 and current smallest $3d/2$ elements in A_2. Thus, since this call to window-sort results in an array with maximum dislocation at most $d/2$, the subarray, B_i, of the d elements moved to the output in step i includes the $d/2$ current smallest elements in $A_1 \cup A_2$. Moreover, the first $d/2$ elements in B_i have no smaller elements that remain in S. In addition, for the $d/2$ elements in the second half of B_i, let S' denote the set of elements that remain in S that are smaller than at least one of these $d/2$ elements. Since the output of Window-Sort$(S, 4d, d)$ has maximum dislocation at most $d/2$, we know that $|S'| \leq d/2$ Moreover, the elements in S' are a subset of the smallest $d/2$ elements that remain in S and there are no elements in $(A_1 \cup A_2) - S$ smaller than the elements in S' (since S includes the $3d/2$ smallest elements in A_1 and A_2, respectively. Thus, all the elements in S' will be included in the subarray, B_{i+1}, of d

elements output in merge step $i+1$. In addition, a symmetric argument applies to the first $d/2$ elements with respect to the d elements in B_{i-1}. Therefore, the output of the merge of A_1 and A_2 will have maximum dislocation at most $3d/2$ with the claimed probability. □

Window-merge-sort clearly runs in $O(n \log^2 n)$ time. This gives us the following.

Theorem 4.3.2. *Given an array, A, of n distinct comparable elements, one can deterministically sort A in $O(n \log^2 n)$ time subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of $O(\log n)$ w.h.p., assuming that the block size B is at least $\log n$.*

This method is not data oblivious, however. For example, in a merge of two subarrays, A_1 and A_2, if each element in A_1 is less than all the elements in A_2, then with high probability the merge will take almost all the elements from A_1 before taking any elements from A_2.

4.4 Window-Oblivious-Merge-Sort

In this section, we describe a deterministic data-oblivious sorting algorithm that can tolerate noisy comparisons, which uses our data-oblivious window-sort only for small subarrays. Our method is an adaptation of the classic odd-even merge-sort algorithm [21] to the noisy comparison model, and it runs in $O(n \log^3 n)$ time, and achieves a maximum dislocation of $O(\log n)$, set in the analysis. We give our algorithm in Algorithm 8, with $d = c \log n$, where c is a constant set in the analysis.

Note that, assuming d is $O(\log n)$, the running time for window-merge is characterized by the recurrence, $T(n) = 2T(n/2) + n \log n$, which is $O(n \log^2 n)$; hence, the running time for window-odd-even-sort is characterized by the recurrence, $T(n) = 2T(n/2) + n \log^2 n$, which is $O(n \log^3 n)$.

Algorithm 8: Window-Odd-Even-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n, d$)

if $N \leq 6d$ **then**
  return Window-Sort($A, 4d, d$)
Divide A into two subarrays, A_1 and A_2, of roughly equal size
Window-Odd-Even-Sort(A_1, n, d)
Window-Odd-Even-Sort(A_2, n, d)
$B \leftarrow$ Window-Merge(A_1, A_2, d)
return B

Window-Merge(A_1, A_2, d):
if $|A_1| + |A_2| \leq 6d$ **then**
  return Window-Sort($A_1 \cup A_2, 4d, d$)
Let A_1^o (resp., A_1^e) be the subarray of A_1 of elements at odd (resp., even) indices
Let A_2^o (resp., A_2^e) be the subarray of A_2 of elements at odd (resp., even) indices
$B_1 \leftarrow$ Window-Merge(A_1^e, A_2^e, d)
$B_2 \leftarrow$ Window-Merge(A_1^o, A_2^o, d)
Let B be the shuffle of B_1 and B_2, so its even (resp., odd) indices are B_1 (resp., B_2)
for $i = 0, 1, 2, \ldots, |B|/d$ **do**
  Window-Sort($B[id : id + 6d], 4d, d$)
return B

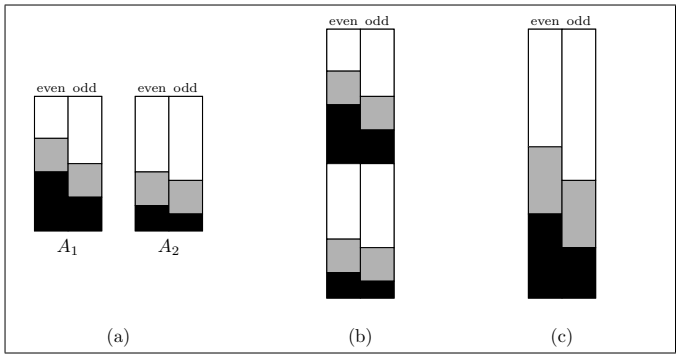


Figure 4.1: `Window-Merge` (a) Subarrays A_1 and A_2. (b) A before the merge. (c) A after the merge.

The correctness of window-merge is proved using induction and the 0-1 principle, which is that if a data-oblivious algorithm can sort an array of 0's and 1's, then it can sort any array[1] [80]. Let n be a power of 2, and consider the elements of each of A_1 and A_2 arranged in two columns with even indices in the left column and odd indices in the right column. (See Figure 4.1.) By the 0-1 principle, if A_1 and A_2 each have maximum dislocation at most d, then, for each arrangement of A_1 and A_2, the difference between the number of 1's in the left column and the number of 1's in the right column is at most $d+1$.

Next stack the two-column arrangement of A_1 on top of that for A_2 and note that our window-merge algorithm recursively sorts each column, which, by induction will each have maximum dislocation d. That is, by the 0-1 principle, each column will consist of a contiguous sequence of 0's, followed by a sequence of length at most $2d$ comprising a mixture of 0's and 1's, followed by a contiguous sequence of 1's. Further, by how we began our arrangement, the difference between the number of 1's in the left column and the number of 1's in the right column in the full arrangement of A_1 and A_2 is at most $2d+2$. Thus, all the unsortedness is confined to a region of at most $4d+2$ consecutively-indexed elements in the merged sequence, which are then completely contained in a region of $5d$ consecutively-indexed elements that begin at a multiple of d. Our window-merge method is guaranteed to call window-sort for this region of elements, bringing its maximum dislocation to be at most d. We observe that there are other calls to window-sort as well, but these will not degrade the sortedness of this region. Thus, the result is that the maximum dislocation of the merged list is at most d.

This gives us the following.

Theorem 4.4.1. *Given an array, A, of n distinct comparable elements, one can deterministically and data-obliviously sort A in $O(n \log^3 n)$ time, subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of $O(\log n)$ w.h.p.*

[1] It is straightforward to show that the 0-1 principle holds for our noisy sorting setting as well.

We note that the only randomization here is in the comparison model. The algorithm for Theorem 4.4.1 is deterministic. If we are willing to use a randomized algorithm, however, we can achieve a faster running time.

4.5 Randomized Shellsort

In this section, we describe a randomized data-oblivious sorting method that runs in $O(n \log n)$ time. The method is the simple randomized Shellsort algorithm of Goodrich [61], which we review in an appendix in Algorithm 9. It is based on peforming region compare-exhanges between subarrays of equal size, which, for a constant $c \geq 1$ set in the analysis, consists of constructing c random matchings between the elements of the two subarrays and performing compare-exchange operations between the matched elements. We study the dislocation reduction properties of randomized Shellsort empirically.

4.6 Annealing Sort

We briefly review here the annealing sort algorithm (see Algorithm 10 in an appendix), first introduced by Goodrich [62], which is a randomized data-oblivious sorting algorithm, and uses the simulated annealing meta-heuristic that involves following an **annealing schedule** defined by a **temperature sequence** $T = (T_1, T_2, \ldots T_t)$ and a **repetition sequence** $R = (r_1, r_2, \ldots, r_t)$. This algorithm essentially uses a randomized round-robin strategy of scanning the input array A and performing, for each $i = 1, 2, \ldots, n$, a compare-exchange operation between $A[i]$ and $A[s]$ where s is a randomly chosen index not equal to i. At each round j, the temperature T_j is then used to determine how far apart the candidate comparison elements with indices i and s should be at each time step. Following the simulated annealing metaheuristic, the temperatures in the annealing schedule decrease over time, and each

random choice is repeated r_j number of times in round j. In our experiments, we follow the same three-phase annealing schedule used in the analysis of this algorithm in [62].

4.7 Experiments

To empirically test the performance of our algorithms under persistent noisy errors, we implemented each of the algorithms described in Sections 4.2 to 4.6, along with RIFFLESORT, which is a non-data-oblivious noisy sorting algorithm introduced by Geissman, Leucci, Liu, and Penna [55] that we review in an appendix in Algorithm 11. We also compare our algorithms to the standard and well-known insertion sort, randomized quicksort, and Shellsort [100] algorithms, e.g., see [35, 66]. For completeness, we include pseudo-code for these classic algorithms in an appendix in Algorithm 12.

We have also considered a variant of randomized Shellsort, which we denote by RANDOMIZEDSHELLSORTNO2S3S that does not include the 2 hop and 3 hop passes (lines 7-8 in Algorithm 9), as we do not think that they are necessary for the algorithm to perform well in practice. For standard Shell sort, we used the Pratt sequence [94], which uses a gap sequence consisting of all products of powers of 2 and 3 less than the array size, and we denote this algorithm by SHELLSORTPRATT.

Parameter configurations. The RIFFLESORT algorithm uses a parameter c to determine the group sizes during noisy binary search. Geissmann, Leucci, Liu, and Penna [55] assume $c = 10^3$ in their analysis; however, we set $c = 5$ so that the algorithm works with the input sequence sizes we use. We also set a parameter, h, of riffle-sort, which affects the height of the noisy binary search tree, to be $\log(\lfloor \frac{n+1}{5d} \rfloor)$, where d is the maximum dislocation of the input sequence given to the noisy binary search tree. For all other parameters, we follow the values used in [55]. We have also made a significant and potentially risky modification to the noisy

binary search algorithm described by Geissmann, Leucci, Liu, and Penna [55] so that it works in a practical setting. In particular, while the original description of this subroutine fixes an upper bound $\tau = \lfloor 240 \log n \rfloor$ on the total number of steps performed in a noisy binary search random walk, we found that this resulted in unreasonably long running times for the input sequence sizes we used, and we instead lowered this upper bound to $\tau = \lfloor 7 \log n \rfloor$ in our implementation. Despite using lower τ, the algorithm surprisingly produces good dislocation bounds, while taking significantly less time. Because of its reliance on noisy binary searching, riffle-sort is not data-oblivious, so we used its performance as the best achievable empirical dislocation bounds, which our data-oblivious methods compare against.

For annealing sort, we follow the annealing schedule and constants used in [42], which defines additional parameters h, g_{scale}, and finds suitable values for them alongside the existing parameters c and q defined by Goodrich [62], all of which affect the temperature and repetition schedules used in the algorithm; hence, we set $h = 1$, $g_{scale} = 0$, $c = 10$, and $q = 1$.

For WINDOWMERGESORT and WINDOWODDEVENMERGESORT, we set $d = \log n$. Though a larger constant multiple of $\log n$ is required for the theoretical proofs of these algorithms, we found that this wasn't necessary in practice; in fact we observed that $d = \log n$ resulted in lower inversions and dislocations in our experiments.

Lastly, for WINDOWSORT, we set $d_1 = n/2$ and $d_2 = \log n$, and for RANDOMIZEDSHELLSORT, we set $c = 4$.

Experimental setup. We implemented each algorithm in C++[2], and compared the performance of each algorithm by measuring the total dislocations, maximum dislocations, and the number of inversions of the output arrays, as well as the total number of pairwise com-

[2]Our implementations of all algorithms can be found at `https://github.com/UC-Irvine-Theory/NoisyObliviousSorting`.

parisons that were done. Each data point in the following plots correspond to the average of 5 runs of the algorithm with random input sequences of integers.

To implement noisy persistent comparisons, we make use of tabulation hashing [90, 105]. In our tabulation hashing setting, we let f denote the number of bits to be hashed, and $s \leq f$ be a block size, and $t = \lceil f/s \rceil$ be the number of blocks. We initialize a two dimensional $t \times 2^s$ array, A, with random q bit integers. Given a key, c, with f bits, we partition f into t blocks of s bits. For our experiments, we set $f = 64$, $s = 8$, and $q = 14$. If c_i represents the i-th block, the hash value $h(c)$ will be derived using the lookup table as follows:

$$h(c) = A[0][c_0] \oplus A[1][c_1] \oplus \cdots \oplus A[t][c_t]$$

For simulating a noisy comparison, given two 4-Byte Integers, $x < y$, we first concatenate the numbers to get the key, $c = (x \cdot 2^{32}) + y$. Then, we hash c to derive $h(c)$, a random $q = 14$ bit integer. We determine that the comparison of these two numbers is noisy if and only if $h(c) \leq p \cdot 2^q$, where p is the noise probability, and output the result of the comparison accordingly.

We performed two sets of experiments: one with a varying probability p of comparison error and fixed input size $n = 32768$, and the other with varying input size n and a fixed probability $p = 0.03$ of comparison error. In our experiments, p takes on values $(2^{-1}, 2^{-2} \ldots, 2^{-10})$, and n takes on values $(2^{16}, 2^{15} \ldots, 2^7)$.

Results and analysis. We first consider experiments with varying p, and compare the maximum dislocations, total dislocations and inversions between each algorithm. We see from Figure 4.2 that all of the data-oblivious algorithms we describe in this chapter have max-

imum and total dislocations that are inline with the theoretical optimal bounds of $O(\log n)$ and $O(n)$ respectively, as well as RIFFLESORT, particularly when $p < 0.1$.

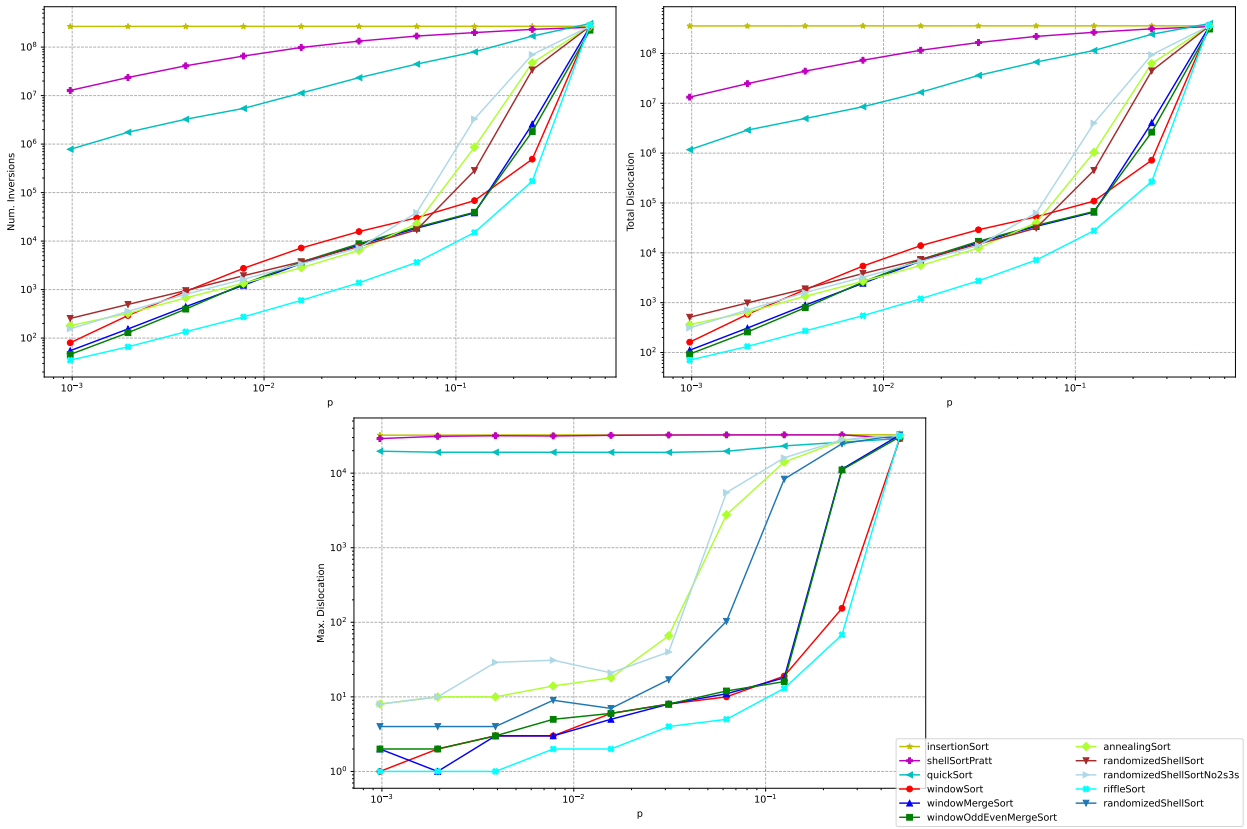


Figure 4.2: Effect of varying the comparison error probability p on the inversion and dislocation counts, with input sequences of size 32768.

For example, we see that WINDOWODDEVENMERGESORT tends to be the best-performing data-oblivious algorithm for different values of p, achieving a total dislocation of at most $\approx 35\,300$, a maximum dislocation of at most 12, and at most $\approx 19\,200$ total inversions for values of $p < 0.1$.

We see that all of the non-standard algorithms tend to form an S-shaped curve, in terms of their dislocation bounds, such that as p starts to increase, the number of dislocations and inversions start to increase slowly, then there is a sharper increase after we reach $p > 0.1$. As expected, we see that the highest dislocation and inversions is when $p = 0.5$, which is the worst-case scenario for p (for any value of $p > 0.5$, reversing the output should result

in a sequence with lower dislocation). In particular, as p goes from $1/32$ to $1/2$, all of the non-standard algorithms go from having up to 100 maximum dislocation and $\approx 28\,900$ total dislocation to having up to $\approx 30\,000$ maximum dislocation and ≈ 345 million total dislocation. These results match our theoretical analyses in this chapter, as we assume that $p \leq 1/16$ in order to prove bounds for the dislocation. The proof for the version of RIFFLESORT we use assumes similar bounds for p [55]. On the other hand, we see that our implementations of insertion sort, quick sort, and Shell sort do not have the tendency to form an S-curve, and their inversion and dislocation counts are significantly higher compared to our algorithms.

In Figure 4.3, we see the effect of varying p and n on the number of comparisons made during the algorithm.

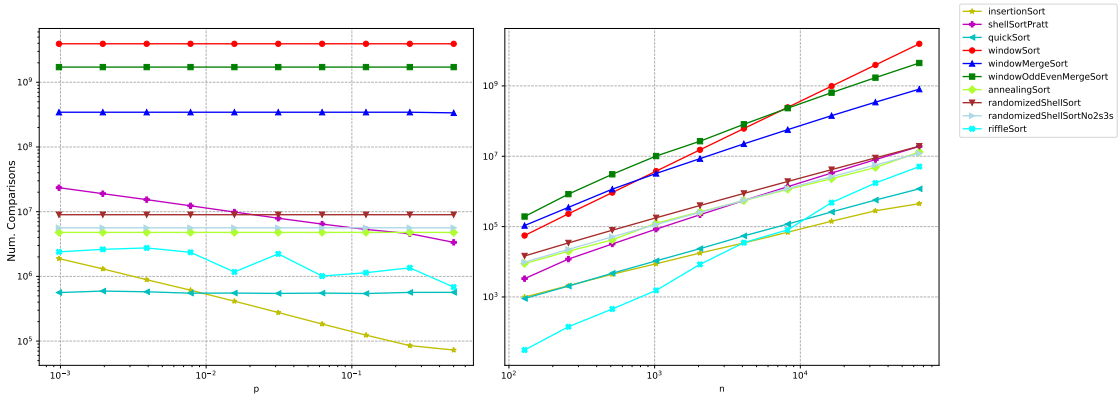


Figure 4.3: Effect of varying the comparison error probability p and the input size n on the number of comparisons.

We see that the number of comparisons tends to grow smaller as p increases in RIFFLESORT, INSERTIONSORT and SHELLSORTPRATT. When the input size is varied, we see that RIFFLESORT, INSERTIONSORT and QUICKSORT use the fewest number of comparisons. Notably, we see that RANDOMIZEDSHELLSORT (and its variant without 2 and 3-hop passes), as well as ANNEALINGSORT, are the best-performing data-oblivious algorithms in terms of the number of comparisons. Overall, we found that RIFFLESORT was the best-performing algorithm in

both sets of experiments; however, it uses the noisy binary search subroutine and is thus not a data-oblivious algorithm.

We also consider how the dislocation is distributed across the output array for each algorithm. In Figure 4.4, we see the average dislocation across different array indices for 5 runs of each algorithm with input sequences of size 16384, and $p = 0.03$. For each output array, we grouped the indices into 128 bins and took the average dislocation inside each bin. From this figure we can see the significant difference in dislocation counts between the standard sorting algorithms insertion sort, Shellsort and quick sort, compared to the other algorithms we implemented. All of the standard sorting algorithms have bins with over 2000 dislocation on average, whereas none of the other algorithms have any bins with over 1.2 dislocations on average, with RIFFLESORT having less than 0.2 dislocations on average accross all of its bins.

While the distribution of dislocation is similar accross most algorithms, we see that insertion sort has most of its dislocation at the two ends of the array, whereas QUICKSORT has a few bins with high dislocation and has lower dislocation for most of the remaining bins.

4.8 Conclusions and Future Work

We introduced the sorting algorithms Window-Merge-Sort and Window-Odd-Even-Sort, both of which are tolerant to noisy comparisons, with the latter also being data-oblivious, with the key difference from existing algorithms being that we do not require use of a noisy binary search subroutine for either algorithms. We then provided both theoretical and experimental analyses, comparing our algorithms to some standard well-known sorting algorithms, and saw that our algorithms perform well in a practical setting as well. Interestingly, we found that the data-oblivious algorithms Annealing sort and Randomized Shellsort per-

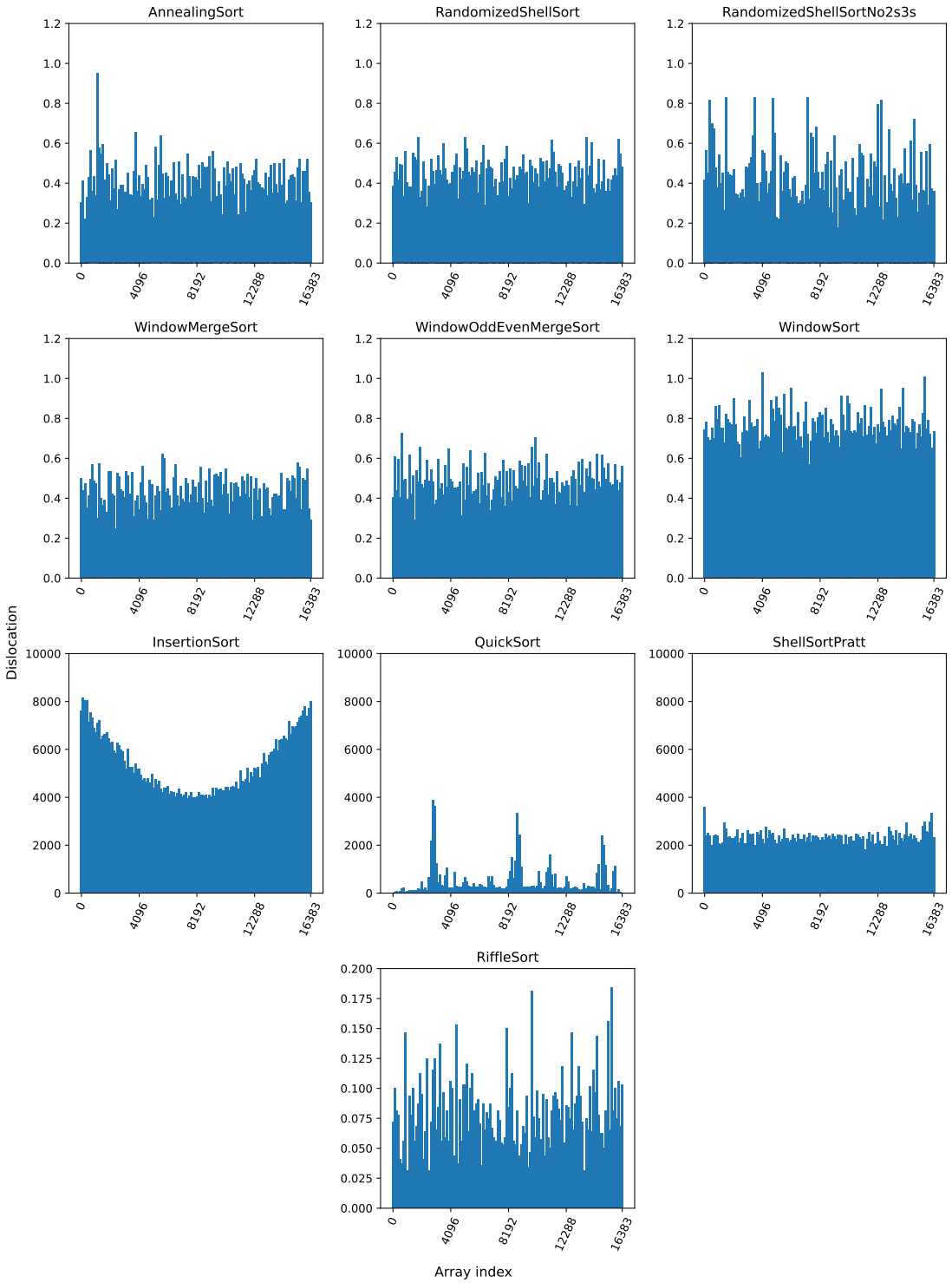


Figure 4.4: Averaged dislocation counts at different array indices over 5 runs for each algorithm on input sequences of size 16384, and $p = 0.03$. Each bar in the histogram corresponds to a bin of 128 indices.

formed quite well under noisy comparisons in our experiments, though we have not provided a theoretical analysis for either of these algorithms. Therefore one possible direction for future work could be to prove similar bounds for these two algorithms.

4.9 Descriptions of Some Existing Sorting Algorithms

In this section, we review some existing sorting algorithms that we included in our tests.

4.9.1 Randomized Shellsort

The first existing sorting algorithm we review is the randomized Shellsort of Goodrich [61], which we give in Algorithm 9. This algorithm is data oblivious.

Algorithm 9: Random-Shellsort($A = \{a_0, a_1, \ldots, a_{n-1}\}$)

```
for o = n/2, n/2^2, n/2^3, ..., 1 do
    Let A_i denote subarray A[io .. io + o − 1], for i = 0, 1, 2, ..., n/o − 1.
    begin a shaker pass
        Region compare-exchange A_i and A_{i+1}, for i = 0, 1, 2, ..., n/o − 2.
        Region compare-exchange A_{i+1} and A_i, for i = n/o − 2, ..., 2, 1, 0.
    begin an extended brick pass
        Region compare-exchange A_i and A_{i+3}, for i = 0, 1, 2, ..., n/o − 4.
        Region compare-exchange A_i and A_{i+2}, for i = 0, 1, 2, ..., n/o − 3.
        Region compare-exchange A_i and A_{i+1}, for even i = 0, 1, 2, ..., n/o − 2.
        Region compare-exchange A_i and A_{i+1}, for odd i = 0, 1, 2, ..., n/o − 2.
```

4.9.2 Annealing Sort

The next existing sorting algorithm we review is the annealing-sort method of Goodrich [62], which we review in Algorithm 10. This algorithm is also data oblivious.

Algorithm 10: Annealing-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n, T, R$)

```
for j = 1, 2, ..., t do
    for i = 1, ..., n − 1 do
        for k = 1, 2, ..., r_j do
            Let s be a random integer in the range [i + 1, min(n, i + T_j)]
            if A[i] > A[s] then
                Swap A[i] and A[s]
    for i = n, n − 1, ... 2 do
        for k = 1, 2, ..., r_j do
            Let s be a random integer in the range [max(1, i − T_j), i − 1]
            if A[s] > A[i] then
                Swap A[i] and A[s]
```

4.9.3 Riffle Sort

We include pseudo-code for the riffle-sort method of Geissman, Leucci, Liu, and Penna [55], which we review in Algorithm 11, for $k = (\log n)/2$ and $\gamma = 2020$. The pseudo-code uses a subroutine $\textsc{test}(x, v)$ (see [55], Definition 1), which checks whether some element x approximately belongs to the interval pointed to by some node v in the noisy binary search tree, which is the main place where this algorithm is not data oblivious.

4.9.4 Well-known Sorting Algorithms

For the sake of completeness, we also include pseudo-code for the well-known insertion-sort, quick-sort, and Shellsort algorithms, in Algorithm 12. None of these three algorithms are data oblivious. One can modify insertion-sort to be data oblivious, however, by continuing the compare-and-swap inner loop process to the beginning of the array in every iteration. Likewise, the Shellsort algorithm can also be modified to be data oblivious in the same manner, since its inner loop is essentially an insertion-sort carried out across elements separated by the gap distance in each iteration.

Algorithm 11: Riffle-Sort($A = \{a_0, a_1, \ldots, a_{n-1}\}$)

$T_0, T_1, \ldots, T_k \leftarrow$ Partition(A)
$S_0 \leftarrow$ WindowSort($T_0, \sqrt{n}, 1$)
for $j = 1, \ldots, k+1$ **do**
    $S_i \leftarrow$ Merge(S_i, T_{i-1})
    $S_i \leftarrow$ WindowSort($S_i, 9\gamma \log n, 1$)
return S_{k+1}

Partition(A) :
for $i = k, \ldots, 1$ **do**
    $T_i \leftarrow 2^{i-1}\sqrt{n}$ elements chosen u.a.r. from $A \setminus \{T_{i+1}, \ldots, T_k\}$
$T_0 \leftarrow$ remaining $\sqrt{n}$ elements in A
return $T_0, \ldots, T_k$

Merge(A, B) :
foreach $x \in B$ **do**
    $rank_x \leftarrow$ NoisyBinarySearch(A, x)
Insert simultaneously all elements $x \in B$ according to $rank_x$ into A
return A

NoisyBinarySearch(A, x) :
Construct noisy binary search trees T_0, T_1 as described in [55], section 3.1.
for $j = 0, 1$ **do**
    $t \leftarrow 7\lfloor \log |A| \rfloor$
    $curr \leftarrow T_j.root$
    while $t > 0$ **do**
        if *curr is a leaf of* T_j **then**
            return *curr*
        Call test(x, c) for each child c of node *curr*.
        if exactly one of the calls pass for some child node c **then**
            $curr \leftarrow c$
        else
            // all tests have failed
            $curr \leftarrow curr.parent$
        $t \leftarrow t - 1$
return *an arbitrary index* // both walks have timed out

Algorithm 12: Well-known sorting algorithms, assuming the input array, A, is of size n and indexed starting at 0. We sort A by calling Insertion-Sort(A, n), Quick-sort$(A, 0, n-1)$, or Shell-sort(A, n, G), where G is a non-increasing *gap sequence* of positive integers less than n, such as the Pratt sequence [94], which consists of all products of powers of 2 and 3 less than n.

```
Insertion-sort(A, n):
for i ← 1, ..., n − 1 do
    j ← i
    while j > 0 and A[j − 1] > A[j] do
        Swap A[j] and A[j − 1]
        j ← j − 1

Quick-sort(A, l, h):
if l < h then
    Choose x uniformly at random from the subarray A[l..h]
    Partition A into A[l..p − 1], A[p], and A[p + 1..h], where A[i] < x for i ∈ [l, p − 1],
      A[p] = x, and A[i] ≥ x for i ∈ [p + 1, h] (if these subarrays exist)
    Quick-sort(A, l, p − 1)
    Quick-sort(A, p + 1, h)

Shell-sort(A, n, G):
foreach g ∈ G do
    for i ← g, ..., n − 1 do
        j ← i
        while j ≥ g and A[j − g] > A[j] do
            Swap A[j] and A[j − g]
            j ← j − g
```

Chapter 5

External-Memory Noisy Sorting

5.1 Introduction

Given n distinct comparable elements, we study the problem of efficiently sorting them subject to noisy probabilistic comparisons. In this framework, which has been extensively studied in internal-memory settings [27, 85, 78, 54, 56, 55, 46, 91, 73, 112, 76, 82], the comparison of two elements, x and y, results in a true and accurate result independently according to a fixed probability, $p < 1/2$, and otherwise returns the opposite (false) result. In the case of *persistent* errors [27, 78, 54, 56, 55], the result of a comparison of two given elements, x and y, always returns the same result. In the case of *non-persistent* errors [46, 91, 73, 112, 76, 82], however, the probabilistic determination of correctness is determined independently for each comparison, even if it is for a pair of elements, (x, y), that were previously compared.

Motivation for sorting with comparison errors comes from multiple sources, including ranking objects online via A/B testing [116], which evaluates the impact of a new technology or technological choice by executing a system in a real production environment and testing

two instances of its performance (an "A" and "B") on a random subset of the users of the platform. Such systems can involve many users and choices to compare via A/B testing, e.g., see [59, 111]; hence, we feel that managing such implementations could benefit from external-memory solutions.

Since one cannot always correctly sort an array, A, subject to persistent comparison errors, we follow Geissmann *et al.* [55, 54, 56], and define the *dislocation* of an element, x, in an array, A, as the absolute value of the difference between x's index in A and its index in the correctly sorted permutation of A. Further, define the *maximum dislocation* of A as the maximum dislocation for the elements in A, and define the *total dislocation* of A is the sum of the dislocations of the elements in A. By known lower bounds [55, 54, 56], the best a sorting algorithm can achieve under persistent comparison errors is a maximum dislocation of $O(\log n)$ and a total dislocation of $O(n)$.

In this chapter, we are interested in sorting algorithms that are in the *external-memory* model. Unfortunately, the existing algorithms for sorting with noisy comparisons are not easily converted into efficient external-memory algorithms, because they all make use of noisy binary search, which involves a random walk in a binary search tree [55, 46, 82]. Instead, we desire efficient sorting algorithms that tolerate noisy comparisons and have an efficient number of input/output operations, primarily for the persistent model, since we can sort an array with maximum dislocation of $O(\log n)$ in the non-persistent model by a single scan where we repeat each comparison in internal memory $O(\log n)$ times.

Intuitively, the main disadvantage of relying on noisy binary search is that it is cache inefficient, in that it requires performing memory accesses for widely-distributed storage locations. Large-scale applications need to minimize the number of input/output (I/O) operations to external memory. Thus, we also desire sorting algorithms that tolerate noisy comparisons and minimize the number of I/Os. In external-memory applications, I/Os occur in terms of memory blocks. In this context, we use M to denote the size of internal memory and B to

denote the size of a block of memory, and we note that the best I/O bound that is possible for sorting an array of size N in external-memory is $\Theta((N/B)\log_{M/B}(N/B))$, see, e.g., [110]. Thus, we also desire sorting algorithm that tolerate noisy comparisons and have this bound on their number of I/Os. Moreover, we desire solutions that are either cache-aware (taking advantage of knowledge of the parameters M and B) or cache-oblvious (which don't know the parameters M and B).

Related Prior Results. The non-persistent error model traces back to a classic problem by Rényi [95] of playing a game involving posing questions to someone who lies with a given probability; see, e.g., the survey by Pelc [92]. Braverman and Mossel [27] introduced the persistent-error model, where comparison errors are persistently wrong with a fixed probability, $p < 1/2 - \varepsilon$, and they achieved a running time of $O(n^{3+f(p)})$ time with maximum expected dislocation $O(\log n)$ and total dislocation $O(n)$. Klein, Penninger, Sohler, and Woodruff [78] improve the running time to $O(n^2)$, but with $O(n \log n)$ total dislocation w.h.p. The internal-memory running time for sorting in the persistent-error model optimally with respect to maximum and total dislocation was subsequentially improved to $O(n^2)$, $O(n^{3/2})$, and ultimately to $O(n \log n)$, in a sequence of papers by Geissmann, Leucci, Liu, and Penna [55, 56, 54].

Feige, Raghavan, Peleg, and Upfal [46] provide a parallel algorithm for sorting with non-persistent errors that, with high probability, runs in $O(\log n)$ time and $O(n \log n)$ work in the CRCW PRAM model, and Leighton, Ma, and Plaxton [82] show how to achieve these bounds in the EREW PRAM model.

None of these prior algorithms translate into efficient external-memory algorithms, however, where we focus on optimizing the number of input/output (I/O) operations. The main reason is that they all use noisy binary searching, which is a random walk in a binary search

tree, where each step involves a noisy comparison. As an external-memory algorithm, this search algorithm unfortunately involves far-flung comparisons; hence, it causes a lot of I/Os.

Frigo, Leiserson, Prokop and Ramachandran [51] introduced the notion of cache-oblivious algorithms, which are algorithms that do not have any variables dependent on hardware parameters such as cache or block size that need to be tuned for it to perform optimally. The authors also introduced the *(M,B) ideal-cache cache model* to analyze cache oblivious algorithms, and defined the *cache complexity* $Q(n)$ and *work complexity* $W(n)$ of an algorithm with input size n, which respectively measure the number of cache misses the algorithm incurs in the ideal-cache model, and the conventional running time of the algorithm in a RAM model. The authors then introduced a cache-oblivious sorting algorithm, Funnelsort, and showed, assuming $M = \Omega(B^2)$ (also known as the *tall-cache assumption*), that Funnelsort is cache-oblivious, has work complexity $O(n \log n)$ and cache complexity $O(1 + \frac{n}{B}(1 + \log_M n))$, which matches the $\Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$ lower bound for sorting in the external-memory model.

Our Results. In this chapter, we provide efficient sorting algorithms in the external-memory model that tolerate noisy comparisons. All our algorithms utilize an optimal number of I/Os. In particular, we provide solutions for either the persistent or non-persistent error models, and for the cache-aware and cache-oblivious external-memory models. Our algorithms avoid using noisy binary searching by instead utilizing a generalized subroutine that is an external-memory version of window-sort. This allows us to then design windowed versions of external-memory merge-sort and funnel-sort. Both algorithms run in time $O(n \log^2 n)$ in internal memory, or in external memory with an optimal $O(n/B) \log_{M/B}(n/B)$ I/O's, subject to comparison errors with probability $p_e < 1/2$ so as to have a maximum dislocation of $O(\log n)$ w.h.p. For both algorithms, we assume that the block size is at least logarithmic in the problem size, i.e., $B = \Omega(\log n)$. Our windowed version of funnel-sort will

also use the tall-cache assumption, i.e., $M = \Omega(B^2)$. In the sections that follow, we describe our algorithms for sorting with comparison errors.

5.2 Window-Sort

We begin with a version of window-sort [55], which will be useful as a subroutine in our algorithms. We provide the pseudo-code in Algorithm 6, for approximately sorting an array of size n that has maximum dislocation at most $d_1 \leq n$ so that it will have maximum dislocation at most $d_2 = d_1/2^k$, for some integer $k \geq 1$, with high probability as a function of d_2.

We note that determining the r_i values can be done by scans; hence, that step is I/O efficient in either cache-aware or cache-oblivious settings. Moreover, the sorting step can be done with an I/O efficient algorithm, in either the cache-aware or cache-oblivious settings, e.g., see [28, 51, 29]. We note that to simplify our presentation, we assume $p_e \leq 1/16$, however this constraint can be relaxed to any $p_e < 1/2$ to obtain the same asymptotic results.

We implement window-sort in external memory, as follows.

Theorem 5.2.1. *Suppose the comparison error probability, p_e, is at most $1/16$. If an array, A, of size n has maximum dislocation at most $d_1 \geq \log n$, then executing Window-Sort(A, d_1, d_2) runs in $O(d_1 n)$ time in internal memory. It can be implemented in external memory with $O(n/B)$ I/Os if $n \leq M$; otherwise, it can be implemented with $O((nd_1/B) + (\log(d_1/d_2))(n/B)\log_{M/B}(n/B))$ I/Os. Executing Window-Sort(A, d_1, d_2) results in A having maximum dislocation of $d_2/2$ with probability at least $1 - 2n2^{-d_2/8}$, where $d_2 = d_1/2^k$, for some integer $k \geq 1$.*

Proof. For the internal-memory running time, note that we can perform the deterministic sorting step using any efficient sorting algorithm in $O(n \log n)$ time. The running times for the windowed comparison steps (step 3 of Algorithm 6) form a geometric sum adding up to $O(d_1 n)$ and the total time for all the deterministic sorting steps (step 4 of Algorithm 6) is $O((\log(d_1/d_2))n \log n)$, which is at most $O(d_1 n)$ for $d_1 \geq \log n$. For the external-memory model in both the cache-aware and cache-oblivious settings, a cache-efficient sorting algorithm can be used, requiring at most $O(\log(d_1/d_2))(n/B) \log_{M/B}(n/B))$ I/Os for all the sorting steps. The scanning step can also be done in an cache efficient way, requiring at most $O(nd_1/B)$ I/Os.

The remainder of the proof follows the same steps in Theorem 4.2.2. □

5.3 Window-Merge-Sort

In this section, we describe a simple external-memory algorithm for sorting with noisy comparisons, which achieves a maximum dislocation of $O(\log n)$. The number of I/Os for this algorithm is optimal. As is common (see, e.g., [23]), we assume that the block size is at least logarithmic in the problem size, i.e., $B \geq \log n$.

We start from the internal-memory version of this algorithm, which we have already described and proved the correctness of in Section 4.3, that runs in $O(n \log^2 n)$ time and then we show how to generalize this method to an efficient external-memory method that uses an optimal number of I/Os. The pseudo-code for this method is in Algorithm 7, with $d = c \log n$ for a constant $c \geq 1$ set in the analysis.

To convert this algorithm to an external-memory one, we just need to make a few changes. First, rather than divide A into 2 subarrays for the recursive calls, we divide A into $m = \Theta(M/B) \geq 2$ subarrays, $A_1, A_2, \ldots, A_m$, each of roughly equal size, and recursively sort

each one. For the merge step, we bring the first $\max\{3d, |A_i|\}$ elements from each A_i, group them together into a list, S, and call Window-Sort$(S, 4md, d)$ on this list, performing this computation entirely in internal memory (so it does not require any additional I/Os). Then we output the first d elements from this window-sort, and continue as in Algorithm 7. This implies the following.

Lemma 5.3.1. *If $A_1, A_2, \ldots, A_m$ each have maximum dislocation at most $3d/2$, then the result of the their merge has maximum dislocation at most $3d/2$ with probability at least $1 - 6mN2^{-d/8}$.*

Proof. The proof follows by similar arguments used in the proof of Lemma 4.3.1. □

This gives us the following.

Theorem 5.3.2. *Given an array, A, of n distinct comparable elements, one can deterministically sort A in internal memory in $O(n \log^2 n)$ time or in external memory with $O((n/B) \log_{M/B}(n/B))$ I/Os subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of $O(\log n)$ w.h.p., assuming $B \geq \log n$.*

5.4 Window Funnelsort

In this section we describe WindowFunnelSort (see Algorithm 13), a noise-tolerant version of the Funnelsort algorithm that sorts n distinct comparable elements so as to have at most $O(\log n)$ maximum dislocation, with $W(n) = O(n \log^2 n)$ work complexity and $Q(n) = O(1 + (n/B)(1 + \log_M n))$ cache complexity, which matches the lower bound of $\Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$ for sorting in the external-memory model. In our pseudocode, n denotes the original input size, while N denotes the size of the input array given to each function call, which can be less than n during recursive calls.

We require a stronger assumption on the cache size for our algorithm: in addition to the tall-cache assumption $M = \Omega(B^2)$, we also require that the block size B be at least logarithmic in the problem size, i.e. $B \geq \gamma \log n$ for some constant $\gamma > 0$ that will be determined in the analysis. In our analysis, we follow the same general proof structure used in [51] with the ideal cache model. For the remainder of this section, we assume that the maximum dislocation bound we would like to obtain, d, is $(3c/2) \log n$ for some constant $c > 0$ that will be determined later. The main difference in our analysis compared to [51] is that in the recursive definition of a k-merger, we define the base cases differently such that each base case k-merger will now use a similar merging method to the one used in Algorithm 7, and our base cases are defined over multiple values of k, instead of just $k = 2$ as done in [51].

Algorithm 13: Window-Funnel-Sort($A = \{a_0, a_1, \ldots, a_{N-1}\}, n$)

```
if N ≤ (c log n)^{3/2} then
    return WindowSort(A, N, c log n)
Divide A into N^{1/3} subarrays, A_1 ..., A_{N^{1/3}}, each of size N^{2/3}
for i = 1, ..., N^{1/3} do
    A_i = Window-Funnel-Sort(A_i)
A ← output of merging A_1 ..., A_{N^{1/3}} using a N^{1/3}-merger, as described in
  Section 5.4
return A
```

We first describe how to construct a k-merger, which is defined recursively in terms of smaller mergers. We follow the same general structure for constructing a k-merger in the original Funnelsort algorithm [51], however in the recursive definition of a k-merger, instead of having $k = 2$ as the base case, we view k-mergers with $\sqrt{c \log n} \leq k < c \log n$ as base cases. As an invariant, each k-merger outputs the next k^3 elements of the approximately sorted sequence obtained by merging its k input sequences.

Our base case k-merger works similarly to the merging procedure in Algorithm 7. We read in $3c \log n$ elements from each of the k inputs into an array S, call WindowSort($S, 4kc \log n, c \log n$), then output the last $c \log n$ elements from this call. Then, we replace the $c \log n$ elements in the k-merger that were just written to the output as follows: for each element e written

to the output, we read into the k-merger a new element from the input queue that e belonged to. We then call WINDOWSORT again on this updated set of elements, and repeat this process until the k-merger has outputted k^3 elements.

For all other values of $k \geq c \log n$, a k-merger will work the same way as in [51], which we describe here for completeness. A (non-base case) k-merger is built recursively out of $\sqrt{k}$-mergers by first partitioning the k inputs into $\sqrt{k}$ sets of $\sqrt{k}$ elements, which forms the input to $\sqrt{k}$ left mergers $L_1, L_2, \ldots, L_{\sqrt{k}}$, each of which is a $\sqrt{k}$-merger. Each L_i is connected to an output buffer i, implemented as a circular queue that can hold up to $2k^{3/2}$ elements. Each buffer is then connected as input to R, which is another $\sqrt{k}$-merger. The output of R then becomes the output of the whole k-merger. Following our invariant, in order to output k^3 elements, the k-merger will invoke R $k^{3/2}$ times. Since the input queues connected to R might become empty, the k-merger first fills all buffers that have less than $k^{3/2}$ elements before each invocation of R, which is done by invoking the corresponding left merger L_i that connects to buffer i. Since each left merger invocation will output $k^{3/2}$ elements to the corresponding buffer, each L_i will only need to be invoked at most once before each invocation of R.

Let us first consider the cache complexity of WINDOWFUNNELSORT. Following the proof in [51], we first consider how much space a k-merger requires.

Lemma 5.4.1. *A k-merger requires at most $O(k^2)$ contiguous memory locations when $k \geq c \log n$.*

Proof. A k-merger with $k \geq c \log n$ requires $O(k^2)$ memory locations for the buffers, and it also requires space for its $\sqrt{k}+1$ $\sqrt{k}$-mergers. Thus, the space $S(k)$ required by a k-merger satisfies the recurrence relation

$$S(k) \leq (\sqrt{k}+1)S(\sqrt{k}) + \beta k^2,$$

for some constant $\beta > 0$. We prove inductively that $S(k) \leq Zk^2$ for some constant Z. For k-mergers with $\sqrt{c \log n} \leq k < c \log n$, we will read in $c \log n$ elements from k input queues, then perform WINDOWSORT on them, requiring $S(k) = O(k \log n)$ space for some constant $\beta > 0$. Thus for $c \log n \leq k < (c \log n)^2$, we have $S(k) \leq (\sqrt{k} + 1)O(\sqrt{k} \log n) + \beta k^2 \leq Zk^2$ for sufficiently large Z.

For $k \geq (c \log n)^2$, we inductively have

$$\begin{aligned} S(k) &\leq (\sqrt{k} + 1)S(\sqrt{k}) + \beta k^2 \\ &\leq (\sqrt{k} + 1)Zk + \beta k^2 \leq Zk^2 \end{aligned}$$

for sufficiently large Z. Thus we have $S(k) = O(k^2)$ for any $k \geq c \log n$. □

Any k-merger with $\sqrt{c \log n} \leq k < c \log n$ reads in $3c \log n$ elements from less than $c \log n$ inputs, and will require $O(\log^2 n)$ space in total. Therefore we require that the block size B is at least $\gamma \log n$ for an appropriate constant $\gamma > 0$ such that after applying the tall-cache assumption $M = \Omega(B^2)$, any k-merger with $\sqrt{c \log n} \leq k < c \log n$ will fit inside the cache. Therefore, more generally, through Lemma 5.4.1, any k-merger with $\sqrt{c \log n} \leq k \leq c \log n \leq \alpha\sqrt{M}$, where α is a sufficiently small constant, will also fit inside the cache and run without any additional cache misses.

The following lemma, which is proved in [51], shows that the $\sqrt{k}$ buffers used in a k-merger can be managed cache-efficiently.

Lemma 5.4.2 (Lemma 4.2. in [51])**.** *Performing r insert and remove operations on a circular queue causes $O(1 + r/B)$ cache misses if two cache lines are available for the buffer.*

We now bound the cache complexity Q_k of one invocation of a k-merger.

Lemma 5.4.3. *One invocation of a k-merger incurs*

$$Q_k = O(k + k^3/B + k^3 \log_M k/B)$$

cache misses.

Proof. We first consider the case $\sqrt{c \log n} \leq k \leq \alpha\sqrt{M}$. From Lemma 5.4.1 and our assumption on the cache size, we know that any k-merger with $\sqrt{c \log n} \leq k \leq \alpha\sqrt{M}$ will fit inside the cache and run with no additional cache misses. Each k-merger has k input queues, and loads a total of $O(k^3)$ elements. Let r_i denote the number of elements extracted from the ith queue. Since $k \leq \alpha\sqrt{M}$ and $B = O(\sqrt{M})$, there are at least $M/B = \Omega(k)$ cache lines available for the input buffers. Thus, through Lemma 5.4.2, the total number of cache misses for accessing the input queues is

$$\sum_{i=1}^{k} O(1 + r_i/B) = O(k + k^3/B).$$

Similarly, the cache complexity of writing to the output queue is $O(1+k^3/B)$. The k-merger incurs an additional $O(k^2/B)$ cache misses through using its internal data structures, for a total of $Q_k = O(k + k^3/B)$ cache misses.

We then consider the case $k > \alpha\sqrt{M}$. We prove by induction that $Q(k) \leq (Zk^3 \log_M k)/B - A(k)$ for some constant $Z > 0$, where $A(k) = o(k^3)$. We first verify that values of $\alpha M^{1/4} < k \leq \alpha\sqrt{M}$ also satisfy this inequality: from the first case, we have $Q(k) = O(k + k^3/B) = O(k^3/B)$ since $B = O(\sqrt{M}) = O(k^2)$ and $k = \Omega(1)$.

For $k > \alpha\sqrt{M}$, for a k-merger to output k^3 elements, the number of times the left mergers are invoked is bounded by $k^{3/2} + 2\sqrt{k}$. The right merger R is also invoked $k^{3/2}$ times. The

k-merger also has to check before each invocation of R whether any of the buffers are empty. This requires at most $\sqrt{k}$ cache misses and is repeated exactly $k^{3/2}$ times, for a total of at most k^2 cache misses. Therefore the cache complexity Q_k of a k-merger satisfies the following recurrence relation:

$$\begin{aligned} Q_k &\leq (2k^{3/2} + 2\sqrt{k})Q_{\sqrt{k}} + k^2 \\ &\leq (2k^{3/2} + 2\sqrt{k})(\frac{Zk^{3/2}\log_M k}{2B} - A(\sqrt{k})) + k^2 \\ &\leq \frac{Z}{B}k^3 \log_M k + k^2(1 + \frac{Z}{B}\log_M k) - (2k^{3/2} + 2\sqrt{k})A(\sqrt{k}), \end{aligned}$$

which is at most $(Zk^3 \log_M k)/B - A(k)$ if $A(k) = k(1 + (2Z \log_M k)/B)$. □

Theorem 5.4.4. WindowFunnelSort *incurs at most $Q(n)$ cache misses, where*

$$Q(n) = O(\frac{n}{B}\log_{M/B}\frac{n}{B}).$$

Proof. If $n \leq \alpha M$ for a sufficiently small constant α, the algorithm will incur at most $O(1+n/B)$ cache misses, since only one k-merger will be active at any time, and the largest possible k-merger will require $O(n^{2/3}) < O(n)$ space. This case also covers the base case in Line 1 of Algorithm 13 through our assumption on the cache size.

If $n > \alpha M$, have the recurrence

$$Q(n) = n^{1/3}Q(n^{2/3}) + Q_{n^{1/3}}.$$

From Lemma 5.4.3, we have $Q_{n^{1/3}} = O(n^{1/3} + n/B + (n \log_M n)/B)$. Therefore the recurrence simplifies to

$$Q(n) = n^{1/3}Q(n^{2/3}) + O((n \log_M n)/B),$$

which has solution $Q(n) = O(1 + (n/B)(1 + \log_M n))$ by induction, which matches the $\Omega(\frac{n}{B} \log_{M/B} \frac{n}{B})$ lower bound for sorting in the external-memory model. □

We now prove that WINDOWFUNNELSORT is tolerant to persistent comparison errors.

Lemma 5.4.5. *Given k input queues with maximum dislocation at most $\frac{3}{2}c \log n$ for some constant $c > 0$, one invocation a k-merger outputs k^3 elements with maximum dislocation at most $\frac{3}{2}c \log n$ with probability at least $1 - 2Zk^3(c \log n)^5 2^{-(c \log n)/8}$ for some constant $Z > 0$.*

Proof. We first consider k-mergers with $\sqrt{c \log n} \leq k < c \log n$. Each such k-merger will call WINDOWSORT $\frac{k^3}{c \log n} < (c \log n)^2$ times, with each call working on at most $(c \log n)^2$ elements. Therefore, using a similar argument to Lemmas 4.3.1 and 5.3.1 and a union bound, the resulting sequence after $(c \log n)^2$ calls to WINDOWSORT will have maximum dislocation at most $(3c/2) \log n$ with probability at least $1 - 2(c \log n)^4 2^{-(c \log n)/8}$.

We then consider the case $k \geq c \log n$. We have $\sqrt{k}$ left $\sqrt{k}$-mergers, along with a $\sqrt{k}$-merger R. Each left merger inductively outputs $k^{3/2}$ elements with dislocation at most $\frac{3}{2}c \log n$, which is used as the input to the $\sqrt{k}$-merger R that also inductively outputs $k^{3/2}$ elements with dislocation at most $\frac{3}{2}c \log n$. Using a similar argument to Lemma 4.3.1, the output queue of the k-merger will also have dislocation at most $\frac{3}{2}c \log n$. To find the success probability, we consider the number of times WINDOWSORT is called. Since the number of invocations of smaller k-mergers is bounded by $2k^{3/2} + 2\sqrt{k}$, the number of invocations of WINDOWSORT, $I(k)$, satisfies the recurrence relation

$$I(k) = \begin{cases} (2k^{3/2} + 2\sqrt{k})I(\sqrt{k}) & k \geq c \log n \\ 1 & \sqrt{c \log n} \leq k < c \log n, \end{cases}$$

which has solution $I(k) = Zk^3 \log k$ for some constant $Z > 0$ using a similar derivation to the one in Lemma 5.4.3. Therefore, using a union bound, the probability that a k-merger outputs

k^3 elements with maximum dislocation at most $\frac{3}{2}c\log n$ is at least $1-2Zk^3(c\log n)^5 2^{-(c\log n)/8}$.

□

Theorem 5.4.6. *Given an array A of n distinct comparable elements, and assuming $B = \Omega(\log n)$, one can deterministically sort A subject to comparison errors with probability $p_e \leq 1/16$, so as to have maximum dislocation of at most $\frac{c}{2}\log n$ for some constant $c > 0$ w.h.p., with at most $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$ cache misses in the cache-oblivious model, and taking $O(n\log^2 n)$ time in a RAM model.*

Proof. By induction, each of the $n^{1/3}$ input sequences given to the $n^{1/3}$-merger has maximum dislocation at most $\frac{3c}{2}\log n$ w.h.p. From Lemma 5.4.5, we have that a $n^{1/3}$ merger outputs n elements with maximum dislocation at most $\frac{3c}{2}\log n$ with probability at least $1 - 2Zn(c\log n)^5 2^{-(c\log n)/8}$ for some constant $Z > 0$. Choosing an appropriate value for c establishes this theorem. □

We now bound the work complexity of WindowFunnelSort, by first bounding the work complexity W_k of a k-merger.

Lemma 5.4.7. *The work complexity W_k of one invocation of a k-merger is $O(k^3\log^2 n)$.*

Proof. We first consider k-mergers with $\sqrt{c\log n} \leq k < c\log n$. The k-merger reads $3c\log n$ elements from k input queues, each of which have maximum dislocation at most $O(\log n)$ from Theorem 5.4.6, for a total of $3kc\log n$ elements, then performs window-sort on these elements, which takes $O(k\log^2 n)$ time. To output k^3 elements, the k-merger needs to repeat this $O(\frac{k^3}{\log n})$ times, taking a total of $O(k^4\log n)$ time, which is bounded by $O(k^3\log^2 n)$ since $k < c\log n$.

For k-mergers where $k \geq c\log n$, to output k^3 elements, the left mergers and right merger are invoked at most $k^{3/2} + 2\sqrt{k}$ and $k^{3/2}$ times respectively. The k-merger also has to check before each invocation of R whether any of the buffers are empty. This takes $O(\sqrt{k})$ time and

is repeated exactly $k^{3/2}$ times, for a total of $O(k^2)$ time. Therefore the total work complexity $W(k)$ of a k-merger satisfies the following recurrence relation:

$$W_k \leq (2k^{3/2} + 2\sqrt{k})W_{\sqrt{k}} + O(k^2).$$

Using a derivation similar to the one in Lemma 5.4.3, we can show that $W_k = O(k^3 \log^2 n)$ by induction. □

Theorem 5.4.8. *The work complexity $W(n)$ of* windowFunnelSort *is $O(n \log^2 n)$ for any input sequence of n elements.*

Proof. We have the recurrence

$$W(n) = n^{1/3}W(n^{2/3}) + W_{n^{1/3}}.$$

From Lemma 5.4.7, we have $W_{n^{1/3}} = O(n \log^2 n)$. Therefore the recurrence simplifies to

$$W(n) = n^{1/3}W(n^{2/3}) + O(n \log^2 n),$$

which has solution $W(n) = O(n \log^2 n)$ by induction. □

5.5 Conclusions and future work

We provided efficient sorting algorithms that tolerate noisy comparisons and are cache efficient in both cache-aware and cache-oblivious external memory models. In [51], the authors introduced another cache-oblivious sorting algorithm based on distribution-sort, that has the same work and cache complexities as funnel-sort. One direction for future work could be

to design and analyze a windowed version of the cache-oblivious distribution sort algorithm that has similar bounds on the work and cache complexities.

Bibliography

[1] M. Abrahamsen, G. Bodwin, E. Rotenberg, and M. Stöckel. Graph reconstruction with a betweenness oracle. In N. Ollinger and H. Vollmer, editors, *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[2] P. Afshani, M. Agrawal, B. Doerr, C. Doerr, K. G. Larsen, and K. Mehlhorn. The query complexity of finding a hidden permutation. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 1–11, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] R. Afshar, A. Amir, M. T. Goodrich, and P. Matias. Adaptive exact learning in a mixed-up world: Dealing with periodicity, errors and jumbled-index queries in string reconstruction. In C. Boucher and S. V. Thankachan, editors, *String Processing and Information Retrieval - 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2020.

[4] R. Afshar, M. B. Dillencourt, M. T. Goodrich, and E. Ozel. Noisy sorting without searching: Data oblivious sorting with comparison errors. In L. Georgiadis, editor, *21st International Symposium on Experimental Algorithms, SEA 2023, July 24-26, 2023, Barcelona, Spain*, volume 265 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[5] R. Afshar, M. T. Goodrich, P. Matias, and M. C. Osegueda. Reconstructing binary trees in parallel. In C. Scheideler and M. Spear, editors, *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, pages 491–492. ACM, 2020.

[6] R. Afshar, M. T. Goodrich, P. Matias, and M. C. Osegueda. Reconstructing biological and digital phylogenetic trees in parallel. In F. Grandoni, G. Herman, and P. Sanders, editors, *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*, volume 173 of *LIPIcs*, pages 3:1–3:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[7] R. Afshar, M. T. Goodrich, P. Matias, and M. C. Osegueda. Parallel network mapping algorithms. In K. Agrawal and Y. Azar, editors, *SPAA '21: 33rd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, 6-8 July, 2021*, pages 410–413. ACM, 2021.

[8] R. Afshar, M. T. Goodrich, P. Matias, and M. C. Osegueda. Mapping networks via parallel kth-hop traceroute queries. In P. Berenbrink and B. Monmege, editors, *39th International Symposium on Theoretical Aspects of Computer Science, STACS 2022, March 15-18, 2022, Marseille, France (Virtual Conference)*, volume 219 of *LIPIcs*, pages 4:1–4:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[9] R. Afshar, M. T. Goodrich, and E. Ozel. Efficient exact learning algorithms for road networks and other graphs with bounded clustering degrees. In C. Schulz and B. Uçar, editors, *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*, volume 233 of *LIPIcs*, pages 9:1–9:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[10] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, sep 1988.

[11] N. Alon and V. Asodi. Learning a hidden subgraph. *SIAM J. Discret. Math.*, 18(4):697–712, 2005.

[12] N. Alon, R. Beigel, S. Kasif, S. Rudich, and B. Sudakov. Learning a hidden matching. *SIAM J. Comput.*, 33(2):487–501, 2004.

[13] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[14] D. Angluin. Queries and concept learning. *Mach. Learn.*, 2(4):319–342, 1987.

[15] D. Angluin and J. Chen. Learning a hidden hypergraph. *J. Mach. Learn. Res.*, 7:2215–2236, 2006.

[16] D. Angluin and J. Chen. Learning a hidden graph using o(logn) queries per edge. *J. Comput. Syst. Sci.*, 74(4):546–556, 2008.

[17] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.

[18] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[19] M. Barthélemy and A. Flammini. Co-evolution of density and topology in a simple model of city formation. *Networks and Spatial Economics*, 9(3):401–425, 2009.

[20] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[21] K. E. Batcher. Sorting networks and their applications. In *Proc. of the Spring Joint Computer Conference (AFIPS)*, pages 307–314. ACM, 1968.

[22] Z. Beerliova, F. Eberhard, T. Erlebach, A. Hall, M. Hoffmann, M. Mihalák, and L. S. Ram. Network discovery and verification. *IEEE J. Sel. Areas Commun.*, 24(12):2168–2181, 2006.

[23] M. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[24] M. W. Bern, D. Eppstein, and F. F. Yao. The expected extremes in a delaunay triangulation. In J. L. Albert, B. Monien, and M. Rodríguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, volume 510 of *Lecture Notes in Computer Science*, pages 674–685. Springer, 1991.

[25] A. Bernasconi, C. Damm, and I. Shparlinski. Circuit and decision tree complexity of some number theoretic problems. *Information and Computation*, 168(2):113 – 124, 2001.

[26] B. Bollobás and O. M. Riordan. Mathematical results on scale-free random graphs. In S. Bornholdt and H. G. Schuster, editors, *Handbook of Graphs and Networks: From the Genome to the Internet*, chapter 1, pages 1–34. Wiley, 2002.

[27] M. Braverman and E. Mossel. Noisy sorting without resampling. In *19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 268–276, 2008.

[28] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *29th Int. Colloq. on Automata, Languages and Programming (ICALP)*, pages 426–438. Springer, 2002.

[29] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *ACM J. Exp. Algorithmics*, 12:1–23, 2008.

[30] X. Cai, Z. Wu, and J. Cheng. Using kernel density estimation to assess the spatial pattern of road density and its impact on landscape fragmentation. *International Journal of Geographical Information Science*, 27(2):222–230, 2013.

[31] M. Chase and S. Kamara. Structured encryption and controlled disclosure. In M. Abe, editor, *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, volume 6477 of *Lecture Notes in Computer Science*, pages 577–594. Springer, 2010.

[32] S.-S. Choi and J. H. Kim. Optimal query complexity bounds for finding graphs. *Artificial Intelligence*, 174(9):551 – 569, 2010.

[33] F. Cicalese. *Fault-Tolerant Search Algorithms: Reliable Computation with Unreliable Information*. 01 2013.

[34] P. Corcoran, M. Jilani, P. Mooney, and M. Bertolotto. Inferring semantics from geometry: the case of street networks. In J. Bao, C. Sengstock, M. E. Ali, Y. Huang, M. Gertz, M. Renz, and J. Sankaranarayanan, editors, *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, pages 42:1–42:10. ACM, 2015.

[35] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 4/e edition, 2022.

[36] L. Dall'Asta, J. I. Alvarez-Hamelin, A. Barrat, A. Vázquez, and A. Vespignani. Exploring networks with traceroute-like probes: Theory and simulations. *Theor. Comput. Sci.*, 355(1):6–24, 2006.

[37] D. J. De Solla Price. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American Society for Information Science*, 27(5):292–306, 1976.

[38] J. Dibbelt, B. Strasser, and D. Wagner. Customizable contraction hierarchies. *ACM J. Exp. Algorithmics*, 21(1):1.5:1–1.5:49, 2016.

[39] S. Dobzinski and J. Vondrak. From query complexity to computational complexity. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 1107–1116, New York, NY, USA, 2012. ACM.

[40] S. Dommers, R. van der Hofstad, and G. Hooghiemstra. Diameters in preferential attachment models. *Journal of Statistical Physics*, 139(1):72–107, 2010.

[41] D. Easley and J. Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge university press, 2010.

[42] K. V. Ebbesen. On the practicality of data-oblivious sorting. Master's thesis, Aurhus Univ., Denmark, 2015.

[43] D. Eppstein and M. T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. *CoRR*, abs/0808.3694, 2008.

[44] D. Eppstein and S. Gupta. Crossing patterns in nonplanar road networks. In E. G. Hoel, S. D. Newsam, S. Ravada, R. Tamassia, and G. Trajcevski, editors, *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*, pages 40:1–40:9. ACM, 2017.

[45] M. Erwig. The graph voronoi diagram with applications. *Networks*, 36(3):156–163, 2000.

[46] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23(5):1001–1018, 1994.

[47] I. Finocchi and G. Italiano. Sorting and searching in the presence of memory faults (without redundancy). pages 101–110, 06 2004.

[48] M. Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In D. Naccache, editor, *Topics in Cryptology CT-RSA*, pages 457–471. Springer, 2001.

[49] A. D. Flaxman, A. M. Frieze, and J. Vera. A geometric preferential attachment model of networks. *Internet Math.*, 3(2):187–205, 2007.

[50] P. Fraigniaud, C. Gavoille, and C. Paul. Eclecticism shrinks even small worlds. In S. Chaudhuri and S. Kutten, editors, *23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 169–178, 2004.

[51] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1), jan 2012.

[52] S. Funke, R. Schirrmeister, and S. Storandt. Automatic extrapolation of missing road network data in openstreetmap. In I. Katakis, F. Schnitzler, T. Liebig, D. Gunopulos, K. Morik, G. L. Andrienko, and S. Mannor, editors, *Proceedings of the 2nd International Workshop on Mining Urban Data co-located with 32nd International Conference on Machine Learning (ICML 2015), Lille, France, July 11th, 2015*, volume 1392 of *CEUR Workshop Proceedings*, pages 27–35. CEUR-WS.org, 2015.

[53] S. Funke and S. Storandt. Provable efficiency of contraction hierarchies with randomized preprocessing. In K. M. Elbassioni and K. Makino, editors, *Algorithms and Computation - 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, volume 9472 of *Lecture Notes in Computer Science*, pages 479–490. Springer, 2015.

[54] B. Geissmann, S. Leucci, C.-H. Liu, and P. Penna. Sorting with recurrent comparison errors. In Y. Okamoto and T. Tokuyama, editors, *28th Int. Symp. on Algorithms and Computation (ISAAC)*, volume 92 of *LIPIcs*, pages 38:1–38:12, 2017.

[55] B. Geissmann, S. Leucci, C.-H. Liu, and P. Penna. Optimal sorting with persistent comparison errors. In M. A. Bender, O. Svensson, and G. Herman, editors, *27th European Symposium on Algorithms (ESA)*, volume 144 of *LIPIcs*, pages 49:1–49:14, 2019.

[56] B. Geissmann, S. Leucci, C.-H. Liu, and P. Penna. Optimal dislocation with persistent errors in subquadratic time. *Theory of Computing Systems*, 64(3):508–521, 2020. This work appeared in preliminary form in STACS'18.

[57] E. Ghosh, S. Kamara, and R. Tamassia. Efficient graph encryption scheme for shortest path queries. In J. Cao, M. H. Au, Z. Lin, and M. Yung, editors, *ASIA CCS '21:*

ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021, pages 516–525. ACM, 2021.

[58] O. Gila, E. Ozel, and M. Goodrich. Highway preferential attachment models for geographic routing. In W. Wu and J. Guo, editors, *Combinatorial Optimization and Applications*, pages 56–80, Cham, 2024. Springer Nature Switzerland.

[59] A. Gilotte, C. Calauzènes, T. Nedelec, A. Abraham, and S. Dollé. Offline A/B testing for recommender systems. In *Eleventh ACM International Conference on Web Search and Data Mining (WSDM)*, pages 198—-206, New York, NY, USA, 2018.

[60] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[61] M. T. Goodrich. Randomized shellsort: A simple data-oblivious sorting algorithm. *J. ACM*, 58(6):27:1–27:26, Dec. 2011.

[62] M. T. Goodrich. Spin-the-bottle sort and annealing sort: Oblivious sorting via round-robin random comparisons. *Algorithmica*, pages 1–24, 2012.

[63] M. T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in $O(n \log n)$ time. In *46th ACM Symposium on Theory of Computing (STOC)*, page 684–693, 2014.

[64] M. T. Goodrich and E. Ozel. Modeling the small-world phenomenon with road networks. *CoRR*, abs/2209.09888, 2022.

[65] M. T. Goodrich and E. Ozel. External-memory sorting with comparison errors. In P. Morin and S. Suri, editors, *Algorithms and Data Structures - 18th International Symposium, WADS 2023, Montreal, QC, Canada, July 31 - August 2, 2023, Proceedings*, volume 14079 of *Lecture Notes in Computer Science*, pages 493–506. Springer, 2023.

[66] M. T. Goodrich and R. Tamassia. *Algorithm Design and Applications*, volume 363. Wiley, 2015.

[67] V. Grebinski and G. Kucherov. Optimal query bounds for reconstructing a hamiltonian cycle in complete graphs. In *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*, pages 166–173, 1997.

[68] V. Grebinski and G. Kucherov. Reconstructing a hamiltonian cycle by querying the graph: Application to DNA physical mapping. *Discret. Appl. Math.*, 88(1-3):147–165, 1998.

[69] V. Grebinski and G. Kucherov. Optimal reconstruction of graphs under the additive model. *Algorithmica*, 28(1):104–124, 2000.

[70] J. Guare. *Six Degrees of Separation: A Play.* Vintage, 1990.

[71] M. Iacono and D. Levinson. Mutual causality in road network growth and economic development. *Transport Policy*, 45(C):209–217, 2016.

[72] S. Kannan, C. Mathieu, and H. Zhou. Graph reconstruction and verification. *ACM Trans. Algorithms*, 14(4), Aug. 2018.

[73] R. M. Karp and R. Kleinberg. Noisy binary search and its applications. In *18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 881–890, 2007.

[74] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994.

[75] C. Kenyon-Mathieu and A. C. Yao. On evaluating boolean functions with unreliable tests. *International Journal of Foundations of Computer Science*, 1(01):1–10, 1990.

[76] K. Khadiev, A. Ilikaev, and J. Vihrovs. Quantum algorithms for some strings problems based on quantum string comparator. *Mathematics*, 10(3):377, 2022.

[77] V. King, L. Zhang, and Y. Zhou. On the complexity of distance-based evolutionary tree reconstruction. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 444–453. ACM/SIAM, 2003.

[78] R. Klein, R. Penninger, C. Sohler, and D. P. Woodruff. Tolerant algorithms. In C. Demetrescu and M. M. Halldórsson, editors, *European Symposium on Algorithms (ESA)*, pages 736–747. Springer, 2011.

[79] J. M. Kleinberg. The small-world phenomenon: an algorithmic perspective. In F. F. Yao and E. M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 163–170. ACM, 2000.

[80] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[81] R. Kumar, D. Liben-Nowell, and A. Tomkins. Navigating low-dimensional and hierarchical population networks. In Y. Azar and T. Erlebach, editors, *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 480–491. Springer, 2006.

[82] T. Leighton, Y. Ma, and C. G. Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *Journal of Computer and System Sciences*, 54(2):265–304, 1997.

[83] D. Liben-Nowell, J. Novak, R. Kumar, P. Raghavan, and A. Tomkins. Geographic routing in social networks. *Proc. Natl. Acad. Sci. USA*, 102(33):11623–11628, 2005.

[84] W. Liu, S.-S. Luo, and P. Chen. A study of secure multi-party ranking problem. In *Eighth ACIS Int. Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, volume 2, pages 727–732, 2007.

[85] C. Mao, J. Weed, and P. Rigollet. Minimax rates and efficient algorithms for noisy sorting. In F. Janoos, M. Mohri, and K. Sridharan, editors, *Proceedings of Algorithmic Learning Theory*, volume 83 of *Proceedings of Machine Learning Research*, pages 821–847, 2018.

[86] C. U. Martel and V. Nguyen. Analyzing Kleinberg's (and other) small-world models. In S. Chaudhuri and S. Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 179–188. ACM, 2004.

[87] C. Mathieu and H. Zhou. A simple algorithm for graph reconstruction. In P. Mutzel, R. Pagh, and G. Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 68:1–68:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[88] S. Milgram. The small world problem. *Psychology Today*, 1(1):61–67, 1967.

[89] M. Mitzenmacher. A brief history of generative models for power law and lognormal distributions. *Internet Mathematics*, 1(2):226–251, 2004.

[90] M. Pătraşcu and M. Thorup. The power of simple tabulation hashing. *J. ACM*, 59(3):1–50, jun 2012.

[91] A. Pelc. Searching with known error probability. *Theoretical Computer Science*, 63(2):185–202, 1989.

[92] A. Pelc. Searching games with errors—fifty years of coping with liars. *Theoretical Computer Science*, 270(1):71–109, 2002.

[93] N. Pippenger. On networks of noisy gates. In *26th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 30–38, 1985.

[94] V. R. Pratt. *Shellsort and sorting networks*. PhD thesis, Stanford University, Stanford, CA, USA, 1972.

[95] A. Rényi. On a problem in information theory. *Magyar Tud. Akad. Mat. Kutató Int. Közl.*, 6:505–516, 1961. See `https://mathscinet.ams.org/mathscinet-getitem?mr=0143666`.

[96] L. Reyzin and N. Srivastava. On the longest path algorithm for reconstructing trees from distance matrices. *Inf. Process. Lett.*, 101(3):98–100, 2007.

[97] R. Rivest, A. Meyer, D. Kleitman, K. Winklmann, and J. Spencer. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20(3):396–404, 1980.

[98] G. Rong, W. Li, Y. Yang, and J. Wang. Reconstruction and verification of chordal graphs with a distance oracle. *Theor. Comput. Sci.*, 859:48–56, 2021.

[99] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, pages 37–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[100] D. L. Shell. A high-speed sorting procedure. *Comm. ACM*, 2(7):30–32, July 1959.

[101] H. A. Simon. On a Class of Skew Distribution Functions. *Biometrika*, 42(3-4):425–440, 12 1955.

[102] A. Slivkins. Distance estimation and object location via rings of neighbors. In M. K. Aguilera and J. Aspnes, editors, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005*, pages 41–50. ACM, 2005.

[103] R. Solomonoff. A formal theory of inductive inference. part i. *Information and Control*, 7(1):1–22, 1964.

[104] G. Tardos. Query complexity, or why is it difficult to separate $NP^A \cap coNP^A$ from P^A by random oracles A? *Combinatorica*, 9(4):385–392, Dec 1989.

[105] M. Thorup. Fast and powerful hashing using tabulation. *Commun. ACM*, 60(7):94—-101, jun 2017.

[106] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

[107] J. Travers and S. Milgram. An experimental study of the small world problem. *Sociometry*, 32(4):425–443, 1969.

[108] S. Ulam and S. Ulam. *Adventures of a Mathematician*. University of California Press, 1991.

[109] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, nov 1984.

[110] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[111] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, and D. L. Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, page 839–848, New York, NY, USA, 2018. Association for Computing Machinery.

[112] Z. Wang, N. Ghaddar, and L. Wang. Noisy sorting capacity. *arXiv*, abs/2202.01446, 2022.

[113] M. Waterman, T. Smith, M. Singh, and W. Beyer. Additive evolutionary trees. *Journal of Theoretical Biology*, 64(2):199–213, 1977.

[114] D. J. Watts. *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press, 1999.

[115] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[116] Y. Xu, N. Chen, A. Fernandez, O. Sinno, and A. Bhasin. From infrastructure to culture: A/B testing challenges in large scale social networks. In *21th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 2227–2236, 2015.

[117] A. C. Yao. Protocols for secure computations. In *23rd IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 160–164, 1982.

[118] A. C. Yao and F. F. Yao. On fault-tolerant networks for sorting. *SIAM Journal on Computing*, 14(1):120–128, 1985.

[119] A. C.-C. Yao. Decision tree complexity and Betti numbers. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 615–624, New York, NY, USA, 1994. ACM.

[120] G. U. Yule. A mathematical theory of evolution, based on the conclusions of dr. j. c. willis, f. r. s. *Phil. Trans. R. Soc. Lond.*, 1925.