Mining Emerging Massive Scientific Sequence Data using Block-wise Decomposition Methods

Qi Zhang

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2009

Approved by:

Wei Wang, Advisor

Leonard McMillan, Co-principal Reader

Jan Prins, Co-principal Reader

Fernando Pardo Manuel de Villena, Reader

David Threadgill, Reader

Abstract

Qi Zhang: Mining Emerging Massive Scientific Sequence Data using Block-wise Decomposition Methods.
(Under the direction of Wei Wang.)

I present efficient data mining algorithms for knowledge discovery on two types of emerging large-scale sequence-based scientific datasets: 1) static sequence data generated from SNP diversity arrays for genomic studies, and 2) dynamic sequence data collected in streaming and sensor network systems for environmental studies. The massive, noisy nature of the SNP arrays and the distributive, online nature of sensor network data pose challenging issues for knowledge discovery such as scalability, robustness, and efficiency. Despite the different characteristics of the SNP arrays and streaming sensor data, when viewed as sequences of ordered observations, both can be efficiently mined using algorithms based on block-wise decomposition methods.

I present models and mining algorithms for inferring the genetic variation structure in genome-wide Single-Nucleotide Polymorphism (SNP) arrays. Genome-wide SNP arrays provide a comprehensive view of genome variation and serve as powerful resources for genetic and biomedical studies. Understanding the patterns of genetic variation in a population of individuals plays an important role in solving many genetics problems such as genealogy reconstruction and gene association studies. In this thesis, I propose data mining models and algorithms to efficiently infer genetic variation structure from the massive SNP panels of recombinant sequences resulting from meiotic recombination. I introduced the Minimum Segmentation Problem (MSP) to infer the segmentation structure of a single recombinant strain, as well as the Minimum Mosaic Problem (MMP) to infer the mosaic structure on a panel of recombinant strains. Both MSP and MMP estimate the ancestral polymorphism patterns exhibited in recombinant strains which provides important inputs for the subsequent association analysis. Efficient dynamic programming and graph algorithms based on block-wise decomposition are proposed which can solve MSP and MMP on genome-wide large-scale panels.

I present efficient algorithms for mining massive streaming and sensor network data for observational sciences such as ecology and environmental studies. I proposed efficient algo-

rithms using block-wise synopsis construction to capture the data distribution online for the dynamic sequence data collected in the sensor network and streaming systems including clustering analysis and order-statistics computation, which is critical for real-time monitoring, anomaly detection, and other domain specific analysis.

Acknowledgements

I would like to gratefully and sincerely thank my advisor Dr. Wei Wang for her guidance and support throughout the course of my Ph.D. study. I'd also express my gratitude to Dr. Leonard McMillan for his feedback and collaboration on several of my research projects since my second year. I'm also thankful to Dr. Fernando Pardo Manuel de Villena and Dr. David Threadgill for inspiring me and providing me the opportunity to explore the exciting area of Computational Genetics. My thanks also go to Dr. Jan Prins, who provided me helpful feedback and discussions on my dissertation and accomplishment of my Ph.D.

The research projects reported in this dissertation involved the efforts of several current or previous students of Data Mining and CompGen groups at UNC. I appreciate their help and support. I'm especially thankful to Dr. Jinze Liu and Yi Liu. Jinze has been a senior to me in the group when I joined and given me many useful advices. I have collaborated on several projects with both Jinze and Yi, including the projects for my dissertation. I would also like to thank Feng Pan, Li Guan, Liangjun Zhang, and many other UNC CS graduate students. I'm thankful for having received enormous help and support from my friends at UNC, especially Hao Wu, Liang Cai, and Yang Liu.

I am grateful to my parents and family for their love, encouragement and support.

Table of Contents

List of Tables . . . . . **xi**

List of Figures . . . . . **xii**

List of Abbreviations **1**

List of Symbols **1**

1 Introduction **1**

1.1 Mining Genomic Data 1

1.1.1 Genome-Wide SNP Arrays 2

1.1.2 Inferring Genetic Variation Patterns 4

1.2 Mining Environmental and Ecological Data 10

1.2.1 Streaming and Sensor Data 10

1.2.2 Challenges with Mining Streaming and Sensor Data 12

1.2.3 Capturing Data Distribution - Clustering and Order Statistics Computation 13

1.3 Thesis Statement 15

1.4 New Results 15

1.4.1 Mining Genome Data 16

1.5 Mining Environmental and Ecological Data 17

1.6 Organization 19

2 Inferring Segmentation Structure of Recombinant Genotype Sequences 20

2.1 Introduction 20

2.2 Related Work 21
2.3 The Minimum Segmentation Problem 22
2.4 Solutions for Genotype Input 25
2.4.1 Enforcing the Constraints and Modeling Noise 30
2.5 Experimental Results 35
2.5.1 Datasets 35
2.5.2 Segmentation Results 37
2.5.3 Running Time 38
2.5.4 Constraint on Segment Number Difference 39
2.5.5 Error Tolerance 40
3 Inferring Genome-wide Mosaic Structure 42
3.1 Introduction 42
3.2 Related Work 43
3.3 Problem Formulation 44
3.4 Inferring the Local Mosaic 46
3.4.1 Maximal Intervals 46
3.4.2 Finding Local Breakpoints 46
3.5 Finding Minimum Mosaic - A Graph Problem 50
3.6 Experimental Studies 53
3.6.1 Kreitman's ADH Data 53
3.6.2 Running Time and Scalability Analysis 55
4 Clustering Distributed Data Streams 58
4.1 Introduction 58
4.2 Related Work 60
4.2.1 Distributed Clustering 60

4.2.2 Approximate In-network Aggregation 61
4.2.3 Clustering Single Stream 61
4.2.4 Coreset and Streaming k-median 62
4.3 Preliminaries and Background 62
4.3.1 Problem Definition 62
4.3.2 Local Summary Structure 63
4.3.3 Topology Dependence 64
4.4 Algorithm 65
4.4.1 Topology-oblivious Algorithm 66
4.4.2 Height-aware algorithm 72
4.4.3 Path-aware algorithm 74
4.5 Experiments and Analysis 76
4.5.1 Benchmark Data 76
4.5.2 Results and Analysis 77
5 Fast Algorithms for Approximate Order-Statistics Computation in Data Streams 83
5.1 Introduction 83
5.2 Related Work 84
5.3 Approximate Quantile Computation 86
5.3.1 Algorithms 86
5.3.2 Implementation and Resuts 95
5.4 Approximate Baised-Quantile Computation 99
5.4.1 Preliminary 99
5.4.2 Algorithms 100
5.4.3 Implementation and Results 112

6 Conclusions 117

6.1 Mining Genomic Datasets 117

6.1.1 Inferring Segmentation Structure of Recombinant Genotype Sequences 118

6.1.2 Inferring Genome-wide Mosaic Structure 119

6.2 Mining Streaming and Sensor Network Environmental Datasets 120

6.2.1 Clustering Distributed Data Streams 120

6.2.2 Fast Algorithms for Approximate Order-Statistics Computation in Data Streams 121

Bibliography 123

List of Tables

2.1 Effect of Enforcing the Constraint on the Segment Number Difference . . 40

3.1 The result on genome-wide 51-strain mouse dataset 57

5.1 This table shows the memory size requirements of the Generalized algorithm (with unknown size) for large data streams with an error of 0.001. Each tuple consists of a data value, and its minimum and maximum rank in the stream, totally 12 bytes. Observe that the block size is less than a MB and fits in the L2 cache of most CPUs. Therefore, the sorting will be in-memory and can be conducted very fast. Also, the maximum memory requirement for the algorithm is a few MB even for handling streams of 1 peta data. 92

5.2 This table shows the properties of the different biased quantile (BQ) and uniform quantile (UQ) algorithms. All the algorithms do not make assumptions on stream sizes or input data ranges except CKMS06 which requires knowledge of input data ranges. Moreover, all the biased quantile algorithms except CKMS05 specify pruning operations to add error on existing summaries and can be applied to sensor networks. Our biased quantile algorithm is both general and applicable to sensor networks. Moreover, it achieves higher performance in terms of quantiles per second (qps) than prior BQ algorithms running on similar hardware. * Performance numbers of CKMS05 and CKMS06 are obtained from [3,4]. 116

List of Figures

1.1 Illustration of 3 SNP sites in a panel of DNA sequences from four strains 3

1.2 Recombination event during meiosis 4

1.3 Point mutation (base "C" changed to "A") 4

1.4 The Collaborative Cross breeding funnel (courtesy of Prof. Leonard Mcmillan). S, T, U, V, W, X, Y, and Z denote the 8 founders. The breeding funnel consists of 2 generations of crosses (G1 and G2), followed by 20 generations of inbreeding (G2:F1 - G2:F20) 5

1.5 Illustration of a mosaic structure on a SNP panel. Gray and blue cells in (a) represent majority and minority alleles, respectively. The black vertical bars in (b) represent the recombination breakpoints that result in a mosaic structure. 9

1.6 Illustration of a wireless sensor network. The edges with arrows between sensor nodes represent the routing structure. 11

2.1 An example subregions. F_1-F_4 are four founder sequences. G is the genotype sequence to be segmented. There are 15 sites in all sequences, where site 10 is the only heterozygous site. $R_1 : [1, 9]$ and $R_2 : [11, 15]$ are the homozygous regions. Δ_1-Δ_5 are the maximal shared intervals in R_1. Δ_6 and Δ_7 are the maximal shared intervals in R_2. $r_1 - r_8$ are the subregions for the entire sequence, out of which r_6 is the heterozygous subregions, and the remaining are the homozygous subregions. 28

2.2 The Collaborative Cross breeding funnel. S, T, U, V, W, X, Y, and Z denote the 8 founders. The breeding funnel consists of 2 generations of crosses (G1 and G2), followed by 20 generations of inbreeding (G2:F1 - G2:F20) 36

2.3 The 8 founder strains chosen for the Collaborative Cross breeding funnel shown in Fig. 2.2 37

2.4 The segmentation result of the proposed algorithm on a G2:F1 animal OR65f18 from Collaborative Cross. The colors of different segments represent different founders shown in Fig. 2.3. 37

2.5 The segmentation result of the proposed algorithm on a Pre-CC animal 13m72 from Collaborative Cross. The colors of different segments represent different founders shown in Fig. 2.3. 38

2.6 Running time with varying parameters. 39

2.7 Segmentation results on data with noise. 41

3.1 Neighboring blocks B_L, B_R fall inside overlapping/adjacent maximal intervals I_L, I_R respectively. The dots in the shaded region represent incompatible SNP pairs of I_L and I_R. 47

3.2 Neighboring blocks B_L, B_R contain different subsets of the incompatible SNP pairs. The dots represent the incompatible SNP pairs contained in the overlapping maximal intervals I_L and I_R. The dots inside the shaded triangle are contained in the neighboring block pair B_L and B_R. 50

3.3 Three block pairs form a node. Block pair 1, 2, and 3 are the left, middle, and right block pair of the node respectively. The breakpoint range of the node is the intersection of the end range of block pair 1, the breakpoint range of block pair 2, and the start range of block pair 3. The vertical stripes correspond to the start range, breakpoint range, and end range of a block. The marked haplotypes in the stripes are the haplotypes which have breakpoints in the corresponding region. 52

3.4 Comparison of Minimum Mosaic and Hapbound/SHRUB results on ADH data. (a): the Minimum Mosaic result; (b): the result inferred from the ARG in (c); (c): the ARG computed using SHRUB(Song and Hein (2005)). The bars in (a) and (b) represent the breakpoints. The dots in (c) represents the recombination events. 54

3.5 Comparison of the running times of MinMosaic and Hapbound over varying number of SNPs (in log scale). The datasets used are from Chr19 of 51-strain dataset and 74-strain dataset. The number of SNPs included varies from 1000 to 4000. 56

4.1 EH summary at a site: This figure highlights the multi-level structure of EH-summary. The incoming data is buffered in equi-sized blocks B_1, B_2,...,B_j,..., each of size $O(\frac{k}{\varepsilon^d})$. The coreset C_j^1 is computed for each block B_j and sent to level $l = 1$. At each level $l > 0$, whenever two coresets C_j^l, C_{j+1}^l come in, they are merged and another coreset $C_{(j+1)/2}^{l+1}$ on $C_j^l \bigcup C_{j+1}^l$ is computed and sent to level $l + 1$. There are at most $\log \frac{N}{k/\varepsilon^d}$ levels. 67

4.2 Error accumulation of Height-aware and Path-aware algorithms. This figure compares the different strategies of assigning additive approximation factors at each site of the tree for height-aware and path-aware algorithms. Height-aware algorithm assigns the additive error uniformly to $\frac{\varepsilon}{2h}$, where h is the height of the tree. Path-aware algorithm assigns the additive error uniformly inside each sub-path, but differently for different sub-paths. 75

4.3 Performance of the algorithms as a function of the total stream size: The overall communications among the sensor network nodes perform k-median clustering are measured using real and synthetic data. The NYSE data consists up to 28K records and the synthetic data consists up to 9 million data values. The approximation error threshold is set to 0.05. For real data, all three algorithms are tested on a 5-node system. For synthetic data, the algorithms are tested on a 10-node system. Figs. 4.3(a) and 4.3(b) demonstrate the overall data communication of the algorithms as a function of input data size. Figs. 4.3(c) and 4.3(d) demonstrate the max per node data communication of the algorithms as a function of input data size. The experiments demonstrate a significant reduction in the overall and max per node communication. 80

4.4 Performance of the algorithms as a function of the number of sites. The data communication of the algorithms is measured as a function of the number of sites on NYSE and synthetic data. 81

4.5 Performance of the algorithms as a function of approximation error: The overall communication of the algorithm decreases as the error increases. Figs. 4.5(a) and 4.5(b) highlight the overall data communication as the error increases. Figs. 4.5(c) and 4.5(d) demonstrate the max per node data communication as the error increases. Approximate clustering is performed on NYSE data with 30K data records and synthetic data with 5 million observations. As the error tolerance increases, it can be observed that both the height-aware and path-aware algorithms perform better than the topology-oblivious algorithm and can further reduce the communication by additional $10 - 30\%$. 82

5.1 Multi-level summary S: This figure highlights the multi-level structure of the ϵ-summary $S = \{s_0, s_1, \ldots, s_L\}$. The incoming data is divided into equi-sized blocks of size b and blocks are grouped into disjoint bags, $B_0, B_1, \ldots, B_l, \ldots, B_L$ with B_l for level l. B_0 contains the most recent block, B_1 contains the older two blocks, and B_L consists of the oldest 2^L blocks. At each level, s_l is maintained as the ϵ_l-summary for B_l. The total number of levels L is no more than $\log \frac{N}{b}$ 87

5.2 Sorted Data: The sorted and reverse sorted input data are used to measure the best possible performance of the summary construction time using the algorithm and GK01. Fig. 5.2(a) shows the computational time as a function of the stream size on a log-scale for a fixed epsilon of 0.001. It is observed that the sorted and reverse sorted computation time curves for GK01 are almost overlapping due to the log-scale presentation and small difference between them (average 1.16% difference). Same reason for the sorted and reverse sorted curves for the algorithm, and the average difference between them is 2.1%. It is also observed that the performance of the algorithm is almost linear and the computational performance is almost two orders of magnitude faster than GK01. Fig. 5.2(b) shows the computational time as a function of the error. It is observed that the higher performance of the algorithm which is $60 - 300\times$ faster than GK01. Moreover, GK01 has a significant performance overhead as the error becomes smaller. 97

5.3 Random Data: The random input data is used to measure the performance of the summary construction time using the algorithm and GK01. Fig. 5.3(a) shows the computational time as a function of the stream size on a log-scale for a fixed epsilon of 0.001. It is observed that the performance of the algorithm is almost linear. Furthermore, the log-scale plot indicates that the algorithm is almost two orders of magnitude faster than GK01. Fig. 5.3(b) shows the computational time as a function of the error. It is observed that the algorithm is almost constant whereas GK01 has a significant performance overhead as the error becomes smaller. . . 98

5.4 Sorted Data: The sorted and reverse sorted input data are used to measure the best possible performance of the summary construction time using our biased quantile algorithm and uniform quantile algorithms ZW07, GK01 using the same error 0.001. This log-scale plot indicates that our algorithm achieves up to 75x higher performance compared to GK01 and comparable performance to ZW07. Fig. 5.4(b) indicates the performance of the our biased quantile algorithm and the uniform quantile algorithms on a 10M stream size. 113

5.5 Random Data: The performance of the summary construction time using our biased quantile algorithm and GK01 over random data. Fig. 5.5(a) shows the computational time as a function of the stream size on a log-scale for a fixed epsilon of 0.001. It is observed that our algorithm is able to compute 1.4-1.6M quantiles per second. In practice, our algorithm is over 30x faster than prior biased quantile algorithms. 115

Chapter 1

Introduction

The recent proliferation of high throughput technologies has catalyzed the transformation of traditional science to data-driven science. The unprecedented growth of scientific datasets leads to the phenomenon of data rich but information poor, demanding for revolutionary knowledge discovery techniques to assist and accelerate scientific discovery.

In this thesis, I present explorative and descriptive modeling with efficient data mining algorithm design for knowledge discovery on two types of emerging large-scale sequence-based scientific datasets: 1) static sequence data generated from SNP diversity arrays for genomic studies, and 2) dynamic sequence data collected in a streaming and sensor network for environmental studies. Both types of datasets are large-scale, containing hundreds of millions of observations. Mining useful patterns, trends, and even anomalies from these datasets can provide valuable insights to the scientific discoveries. The massive, noisy, or distributive, online nature of many datasets poses challenging issues for mining algorithm design such as scalability, robustness, and efficiency.

1.1 Mining Genomic Data

The high-throughput power of modern sequencing facilities generate massive amounts of whole-genome datasets. The Human Genome Project alone sequenced the 3 billion DNA base pairs in the human genome. As part of the Human Genome Project, a

high-quality draft of the mouse genome was produced and analyzed in 2002 by the Mouse Genome Sequencing Consortium. The amount of whole-genome data is growing at an unprecedented rate, with over 1000 whole-genome datasets currently completed or under construction. This new wealth of information is quickly outstripping our ability for analysis in genetic studies, such as recombination detection and association mapping.

1.1.1 Genome-Wide SNP Arrays

Genetic variation plays an important role in determining people's disease susceptibilities and responses to drugs and vaccines. Genome-wide Single-Nucleotide Polymorphism (SNP) arrays provide a comprehensive view of genome variation and serve as powerful resources for genetic and biomedical studies.

Markers for Genetic Variation – SNP

The genetic information of a living organism is encoded in DNA sequences. A DNA sequence consists of the sequence of nucleotide bases (adenine (A), cytosine(C), guanine(G) or thymine(T)) in a DNA strand. For any two humans, 99.9% of their DNA sequences are identical. Of the remaining sites that are polymorphic between two people, 80% are single-nucleotide polymorphism (SNP) sites (Fig. 1.1), where there are at least two alleles occur in the observed population with a frequency above 1%. Genome-wide SNP arrays represent one of the most comprehensive tools for measuring genetic variation. A map of more than 3.1 million SNPs over the human genome has been generated through The International Hapmap Project. For the mouse genome, NIEHS and Perlegen Sciences have generated a genome-wide map of 8.27 million SNPs over 15 commonly used strains of inbred laboratory mice.

The sequence of SNPs on a chromosome is referred to as a *haplotype* sequence. Most of the SNPs are biallelic, with only two alleles at any site across the population. The allele with higher frequency is called the majority allele, the other is called the minority

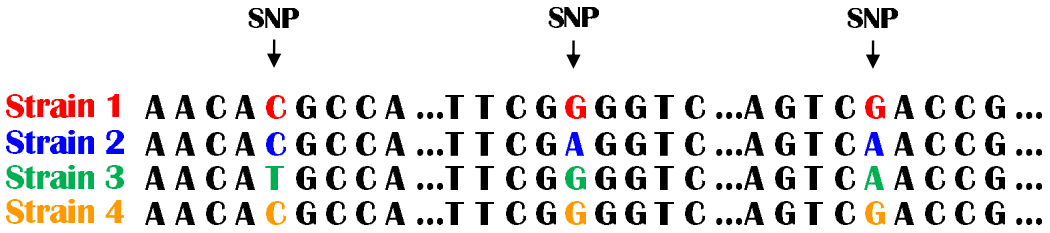


Figure 1.1: Illustration of 3 SNP sites in a panel of DNA sequences from four strains

allele. For diploid animals such as human and mouse, each chromosome has two copies, and each copy corresponds to a haplotype sequence. The combination of the alleles at a SNP site is referred to as a *genotype*. The current technology for obtaining the genotype sequence is called genotyping, which determines for each locus, whether the genotype is homozygous for the majority allele (both haplotypes have majority allele), homozygous for the minority allele (both haplotypes have minority allele), or heterozygous (the two haplotypes have different alleles).

Sources of Genetic Variation – Recombination and Mutation

The two major molecular events shaping genetic variation current populations are recombination and mutation.

- **Recombination**

  During meiosis in sexual organisms, two homologous chromosomes cross over and exchange genetic material (Fig. 1.2). Recombination leads to offsprings with different combinations of genetic variants from their parents.

- **Mutation**

  Mutations are changes to the nucleotide sequence of the genetic material of an organism. The most common type of mutation is point mutation, which is the replacement of a single nucleotide base with another nucleotide base (Fig. 1.3). Other types of mutations include insertions, deletions, duplications, etc.

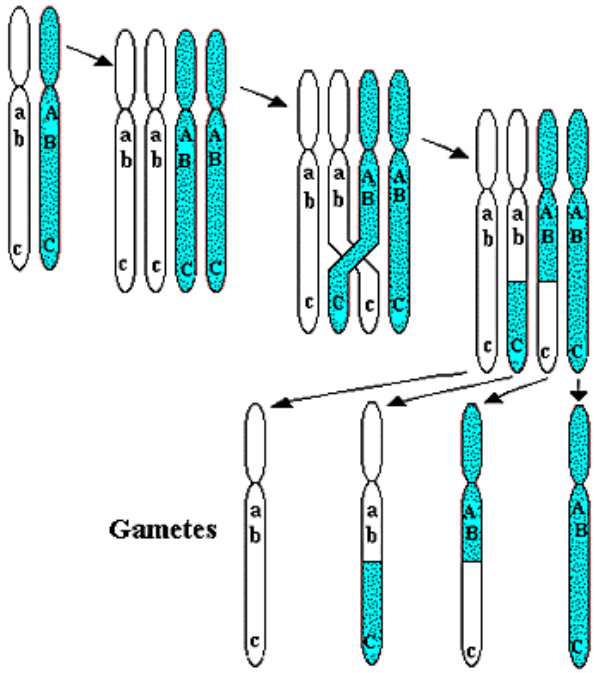


Figure 1.2: Recombination event during meiosis

Normal: A A C A C G C C A ...T T C G C G G T C ...A G T C G A C C G ...
Mutated: A A C A C G C C A ...T T C G A G G T C ...A G T C G A C C G ...

Figure 1.3: Point mutation (base "C" changed to "A")

1.1.2 Inferring Genetic Variation Patterns

Patterns of genetic variation in a population are the product of mutation and recombination events that have occurred over many generations from the ancestors of the population. Understanding genetic variation patterns plays an important role in solving many genetic problems such as reconstructing genealogies and gene association studies.

Inferring Segmentation Structure of a Single Recombinant Strain

Current animal resources for genetics research are often derived by mating a small set of founders. The meiotic recombination events during the mating in each generation result in a fragmental structure in the derived recombinant strains. During the process of generating these model organisms, mutations rarely happen and are considered as

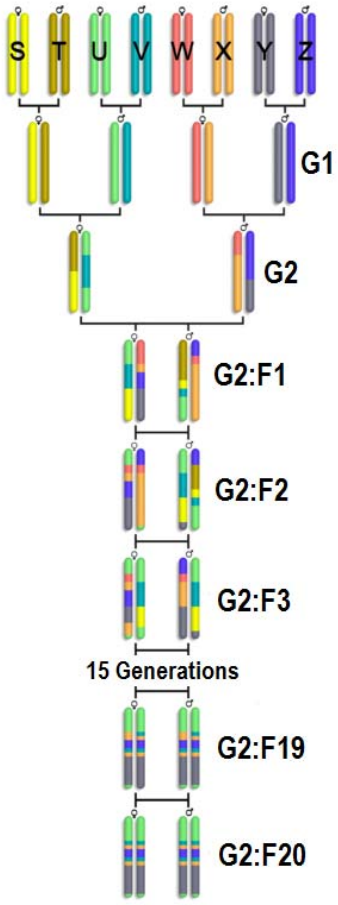


Figure 1.4: The Collaborative Cross breeding funnel (courtesy of Prof. Leonard Mcmillan). S, T, U, V, W, X, Y, and Z denote the 8 founders. The breeding funnel consists of 2 generations of crosses (G1 and G2), followed by 20 generations of inbreeding (G2:F1 - G2:F20)

noise.

An example genetic model system is the Collaborative Cross (CC), a large panel of 1000 recombinant inbred (RI) mouse strains generated from a funnel breeding scheme initiated with a set of 8 founders. (Fig. 2.2). The founder set was selected to maximize diversity among laboratory mouse strains. Each founder strain is inbred, and thus isogenic (homozygous at every allele). The genomes of each founder strain are shown in a different color (Fig. 2.2), and the resulting inbred CC strain is a mosaic of these genomes resulting from 2 crosses (G1 and G2), followed by 20 generations of inbreeding. The CC strains are infinitely reproducible, inbred lines which capture and randomize

nearly 90% of the known variation in laboratory mice, providing unparalleled power for disease association studies. As shown in Fig. 2.2, the CC strains (G2:F20) and the pre-CC strains (G2:F1-G2:F19) are composed of segments from the 8 founder sequences. The segmentation structure identifies the ancestral origin of each region on a recombinant strain, which is important input for subsequent association studies.

In this thesis, I investigate the problem of inferring the segmentation structure for the recombinant strains given a small set of founder sequences. The challenges for solving this problem are three fold:

- **Genotype input**

  Compared with DNA sequence, genotypes are less expensive to obtain experimentally. Algorithms with genotype input are thus more desirable. However, for genotype sequence, the exact allele combinations at heterozygous sites cannot be directly derived. To infer the two plausible haplotypes given the genotypes, phasing algorithms are usually applied, which are known to be computationally intensive.

- **Biological constraints**

  Biologically relevant constraints are critical in defining the biologically feasible solutions for the segmentation problem. For example, in the Collaborative Cross, a specific breeding scheme (the order of the 8 founder strains coming into the funnel) defines the set of possible founder pairs for the genotype at each locus. Moreover, since each autosome undergoes one recombination event on average during each meiosis, the number of segments on the two associated haplotypes of the genotype is expected to be comparable. Without considering these biological constraints, the algorithm may generate spurious solutions.

- **Noise in the data**

Noise is common in biological datasets. There are both biological and technical sources of noise in genotyping, which include point mutations, gene conversions, and genotyping errors. In addition, there may be missing values in the data. Noise and missing values need to be properly modeled to make the algorithm robust for running on real biological datasets.

Previous studies have focused on similar but different models.

- Combinatorial analysis of founder set reconstruction problem

  In (Ukkonen (2002)) and (Wu and Gusfield (2007)), combinatorial approaches are employed to solve the founder set reconstruction problem with a given set of sample haplotype sequences that are evolved from a small set of founders. Different from these models, I study the "inverse" problem where the set of founder sequences are already known, and compute the segmentation structure for genotype sequences of the recombinant strains given the founder sequences. The problem is important for analyzing the ancestral polymorphism presented in experimental model resources for subsequent gene association studies. A real motivating study is analyzing the segmentation structure for pre-CC strains in Collaborative Cross.

- Probabilistic inference of SNP origin

  Mott (Mott et al. (2000)) et al. proposed an HMM-based algorithm for inferring the probability of each founder pair as the origin for each genotype at a SNP site, assuming the founder sequences are known beforehand. Different from the HMM-based algorithm, I study the problem of explicitly deriving all the possible segmentation structures with certain biological meaningful optimization criterion.

- Other related problems

  Other related work on analyzing the genetic variation structure of genome sequences include identifying haplotype blocks (Dally et al. (2001); Gabriel et al.

(2002); Schwartz et al. (2003)), computing phylogenies (Gusfield (2002); Gusfield et al. (2004)).

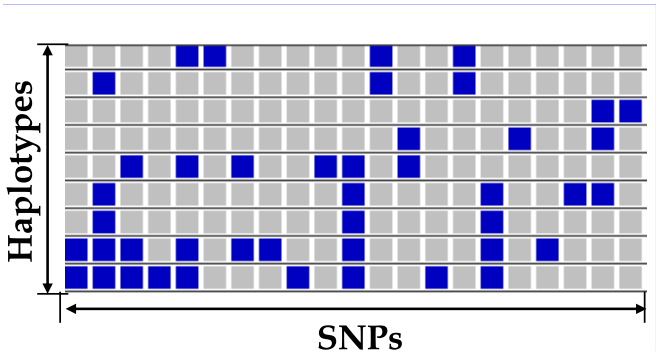
Inferring Mosaic Structure of a Recombinant Strain Panel

Genetic recombination is an important process in shaping the arrangement of polymorphisms within populations. "Recombination breakpoints" in a given set of genomes of a population disrupt the linkage disequilibrium (LD) existing in the genomes and divide the genomes into haplotype blocks, resulting in a mosaic structure (Fig. 1.5). Without prior knowledge of the founder sequences, this mosaic structure can only be inferred through the estimation of recombination breakpoints. Recombination breakpoints represent the locations where the crossovers have occurred, either during the generation of the haplotype itself, or in previous generations (carried over from ancestors).

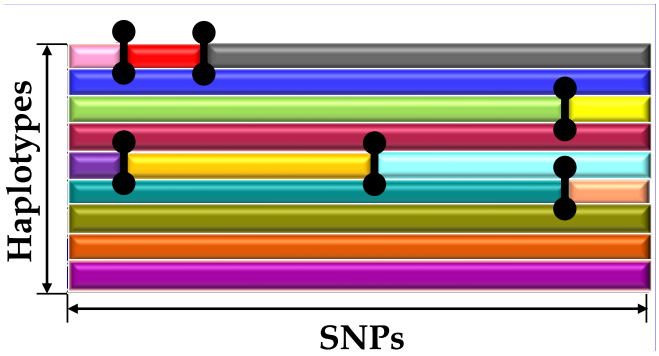
Under the infinite site model (which assumes at most 1 mutation at each site), the recombination breakpoint can be inferred using the Four Gamete Test (FGT). The FGT states that the number of different allele combinations between any two sites can at most be 3 if no visible recombinations have happened in between. Two SNPs are called incompatible if all 4 allele combinations exist. In this thesis, I investigate the problem of inferring the genome-wide mosaic structure consisting of the set of the recombination breakpoints which explain the SNP incompatibilities using FGT.

The challenges of inferring the genome-wide mosaic structure comes from the sheer scale of the data. For a SNP panel of tens of strains over hundreds of thousands of SNPs (a typical size for a mouse chromosome), the number of possible sets of recombination breakpoints would grow exponentially in the size of the panel. Solving large-scale combinatorial problem requires efficient algorithm design.

Many algorithms have been developed for several related problems, such as estimation of recombination rate (Hudson and Kaplan (1985); Myers and Griffiths (2003); Song et al. (2005)), inferring haplotype block structure (Gabriel et al. (2002); Patil et al.



(a) *A SNP panel*



(b) *The mosaic structure on the SNP panel in (a) resulting from historical recombinations*

Figure 1.5: Illustration of a mosaic structure on a SNP panel. Gray and blue cells in (a) represent majority and minority alleles, respectively. The black vertical bars in (b) represent the recombination breakpoints that result in a mosaic structure.

(2001)), recombination detection (Posada (2002); G.F. (1998); Hein (1990, 1993); N.C. and Holmes (1997); Holmes et al. (1999); Lole et al. (1999); Martin and Rybicki (2000); Drouin et al. (1999); Jakobsen et al. (1997); Maynard and Smith (1998); Stephens (1985); Worobey (2001)), and others. Different from these related problems, the mosaic model captures the possible locations of the breakpoints on each haplotype in the population which can explain the SNP incompatibility resulting from recombinations. Inferring such a finer scale mosaic structure is important for many genetics problems such as gene association studies.

1.2 Mining Environmental and Ecological Data

With recent advances in sensor technology, large-scale wireless sensor networks are deployed for observing the natural environments, monitoring habitat and wild populations, offering environmental and ecology scientists access to enormous volumes of data collected from physically-dispersed locations in a continuous fashion. Some of the experimental systems deployed include Berkeley's habitat modeling at Great Duck Island (Szewczyk et al. (2004)), FLOODNET project[1] which provides a flood warning in the UK, and SECOAS project[2] which monitors coastal erosion around small islands intended as wind-farms. The large-scale, distributive and online nature of sensor network and stream data presents new computational challenges for data mining tasks such as mining the distribution of the data, mining frequent patterns or temporally drifting, evolving, periodic patterns, and anomaly detection. Adding to the challenges are stringent system constraints including limited size of memory, battery, and processing power of the sensor nodes. In this thesis, I investigate several related problems to discover the data distribution online for sensor network and stream systems. These problems include clustering analysis and order-statistics computation, which is critical for real-time monitoring, anomaly detection, and domain specific analysis.

1.2.1 Streaming and Sensor Data

The recent advances in sensor technologies enables environmental and ecological data collection at an unprecedent scale and rate. A wireless sensor network is composed of spatially distributed sensors which are battery-powered mini computers that can monitor the environmental measurements such as temperature, sound, light, pressure, motion or pollutants at different locations (Fig. 1.6). A typical sensor node includes

[1] http://envisense.org/floodnet.htm

[2] http://envisense.org/secoas.htm

an antenna and a radio frequency (RF) transceiver to allow communication between sensors, a CPU, a memory unit, a power source such as battery, and a sensor unit which collects environmental measurements.

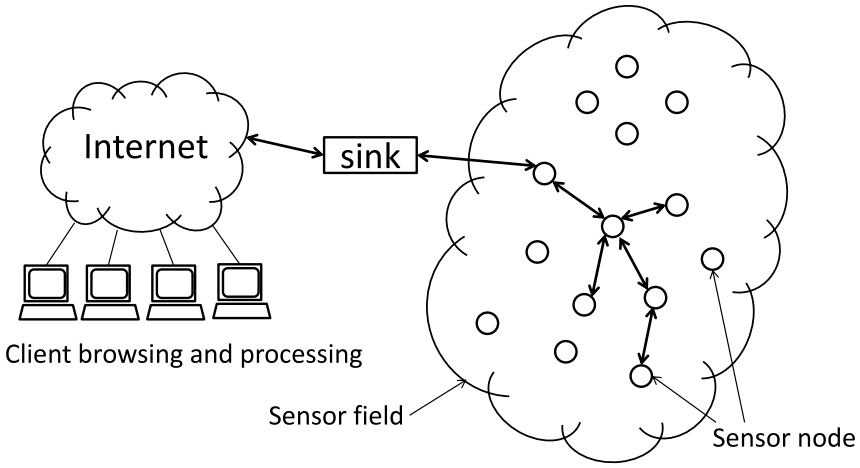


Figure 1.6: Illustration of a wireless sensor network. The edges with arrows between sensor nodes represent the routing structure.

Sensors communicate with each other using multi-hop connections. The data is transmitted towards a special kind of node called a base station (or sink). A base station links the sensor network to the Internet for further browsing and processing at the client side. Base stations usually have enhanced capabilities over the sensor nodes to provide more complex data analysis.

Streaming sensor data and its characteristics

Observations and measurements are collected at each sensor node continuously as ***data streams*** – the ordered sequence of sensor readings. Different from static datasets, data streams present several unique characteristics:

- *Data streams are real-time, and possibly high-speed.*

  The sensors collect and transmit the readings in real-time. Depending on the applications, the readings may arrive at high speed.

- *Data streams are potentially unbounded in size.*

  Once deployed, the sensors continuously collect the measurements which result in massive amount of data over time.

- *Data streams cannot be explicitly stored.*

  Due to the limited memory at each sensor, the massive-scale of raw readings in data streams cannot be completely stored for further retrieval and processing.

Additionally, for sensor network systems,

- *Multiple data streams are collected in a distributed fashion.*

  The distributed data streams are organized according to the routing topology of the sensor network. The data are eventually collected at the sink.

1.2.2 Challenges with Mining Streaming and Sensor Data

The unique characteristics of streaming and sensor network data lead to a number of computational and data mining challenges:

- The sheer volume of the data streams and the limited memory make it impossible to store the entire data stream or even scan the data multiple times. The data mining algorithms are thus required to be single pass, or a few passes.

- The continuous, real-time nature of the data streams requires data mining algorithms to be continuous, incremental, and on-the-fly.

In addition, the system constraints of sensor network pose extra computational challenges for data mining algorithm design:

- Sensor nodes are inherently resource-constrainted. They have limited storage capacity, battery, and processing power. The sensors are sometimes deployed in

hazardous environment which are inaccessible after deployment. The major drain on the sensor's battery is data transmission, which determines the lifetime of the sensor node. Therefore, the data mining algorithms are required to be resource-aware.

- The wireless sensor networks are essentially distributed data stream systems. Due to the limited power, it is not affordable to transmit all local streams of raw sensor readings towards the sinks for centralized processing. As a result, the data mining algorithms need to be distributed across the sensor nodes.

1.2.3 Capturing Data Distribution - Clustering and Order Statistics Computation

In this thesis, I present several algorithms to discover the data distribution online and continuously for streaming and sensor network systems including clustering analysis and order-statistics computation.

Clustering Distributed Data Streams

Clustering, a useful tool in data analysis, is the problem of finding a partition of a dataset so that, under a given distance metric, similar items are grouped together. Clustering the data collected in distributed data streams systems such as a sensor network provides a good estimation of the underlying data distribution.

Clustering over distributed streams is a challenging task. Difficulties lie in various issues:

- Communication

  Distributed stream system continuously produces large volume of data, which imposes prohibitive communication load if all the data are transferred to the sink for centralized computation. In-network aggregation (Madden et al. (2002)) is

one of the techniques (Olston et al. (2003); Madden et al. (2002); Silberstein et al. (2006); Willett et al. (2004)) that push processing operators down into the network to reduce data transmission. It computes a local summary at each site and merges and summarizes further at each internal site towards the root. However, this approach cannot be immediately adopted to solve the clustering problem, since clustering is usually a holistic computation (Madden et al. (2002)) which cannot be readily decomposed into computations on data partitions.

- Clustering Quality

  Accuracy is usually traded for reduced communication through sketching or synopsis construction. However, it is necessary to provide approximate distributed clustering with a guaranteed bounded error.

- Topology

  The topology of the underlying routing network is an important factor that influences the performance of the distributed clustering algorithm. The topology sensitivity of the algorithm determines the adaptability of the algorithm given different knowledge about the routing topology.

Order Statistics Computation on High-Speed Data Streams

In addition to clustering analysis, order statistics is also one of the fundamental tools to capture the distribution of the dataset, by associating the rank and value of the data. As a popular order statistics, quantiles have found wide application in database and scientific computing. Different from quantile computation on static datasets, streaming quantile computation is required to be single-pass, space efficient and continuous.

Many algorithms have been proposed for computing approximate quantiles over the entire stream history (Manku et al. (1998); Greenwald and Khanna (2001)) or over a sliding window (Lin et al. (2004); Arasu and Manku (2004)); with uniform error (Manku

et al. (1998); Greenwald and Khanna (2001)) or with biased error (Cormode et al. (2005, 2006)). The best reported storage bound for approximate quantile computation is $O(1/\epsilon \log(\epsilon N))$ (Greenwald and Khanna (2001)), where N is the size of the stream, and ϵ is the approximation bound. However, most of these algorithms focus on reducing the space requirement at the expense of the computational cost, which is important for processing high-speed data streams with satisfactory real-time performance. In addition to single-pass and continuous computation, the requirements for efficient quantile computation algorithms over high-speed data streams are:

- The algorithm should have a lowe per element computation cost as well as low storage bound.
- In order to guarantee the precision of the result, the algorithm should ensure random or deterministic error bound for the quantile computation.
- The algorithm should be able to handle stream size which is not known a priori.

1.3 Thesis Statement

Efficient algorithms can be designed for mining massive sequence-based scientific datasets from emerging biological and environmental applications such as genomic datasets and streaming and sensor datasets using block-based decomposition methods.

1.4 New Results

This section highlights the key results presented in this thesis. The results in solving the two mining problems for genome data and two mining problems for streaming and sensor network data are summarized below.

1.4.1 Mining Genome Data

Inferring Segmentation Structure of a Single Recombinant Strain

- **Model** I propose the Minimum Segmentation Model to infer the origins of haplotype segments in a recombinant strain (genotype sequence) given a set of known founders (haplotype sequences). Each segment on a recombinant strain is attributable to one of the founders. The minimum segmentation can be used for inferring the relationship among recombinant sequences to identify the genetic basis of traits, which is important for disease association studies. The basic model is also extended to handle noise and support additional biologically-motivated constraints. The biological constraints include the funnel breeding scheme and the requirement of comparable number of segments on both haplotypes. These constraints guarantee the biological validity of the solutions as well as significantly reduce the search space. Furthermore, noise (point mutations, gene conversions, and genotyping errors) and missing values, as common to all biological datasets, are incorporated to improve the robustness of the model.

- **Algorithm** I propose efficient dynamic programming algorithm to solve the Minimum Segmentation problem in polynomial time based on block-wise decomposition of the sequences. The algorithm has a time complexity of $O(LN + P^4)$ and a space complexity of $O(PN^2)$, where L is the number of SNPs, N is the number of founders, and P is the number of blocks.

- **Performance** The proposed algorithms permits genome wide analysis on real mouse genome datasets (Collaborative Cross), and generates feasible biological solutions on CC (preCC and G2F1 strains). The proposed algorithm can also handle noise and missing values properly on these strains.

Inferring Mosaic Structure of a Recombinant Strain Panel

- **Model** I propose the Minimum Mosaic Model to capture the minimum number of recombination breakpoints required for generating a set of genome sequences (in haplotypes). This mosaic structure provides a good estimation of the rate and possible locations of the recombination events, which is useful for inferring haplotype block structures and genealogical history.

- **Algorithm** I proposed an efficient graph-based algorithm for computing the minimum mosaic structure for a given set of haplotype sequences. The strains in the SNP arrays are divided into blocks. For any two neighboring haplotype blocks, the local breakpoints are inferred according to the Four-Gamete Test (FGT). Possible local breakpoint sets (as graph nodes) are connected to form a combinatorial search space for minimum solution.

- **Performance** The proposed algorithm permits genome-wide analysis. The experiments on CGD mouse genome datasets with 51 mouse strains over total 7.8M SNPs demonstrates the good performance of the algorithm (less than half an hour for each chromosome).

1.5 Mining Environmental and Ecological Data

Clustering Distributed Data Streams

- **Model** I define the problem of approximate k-Median Clustering over data arriving at a distributed data stream system such as a sensor network.

- **Algorithm** I propose a suite of resource-aware algorithms for continuously computing $(1+\epsilon)$-approximate k-median clustering over distributed data streams under three different topology settings: topology-oblivious, height-aware, and path-aware. I incorporated the in-network aggregation techniques to avoid the raw

data transmission between sites, which is the main drain on the sensor's battery. Efficient summary structures are designed to reduce data transmission as well as guarantee bounded-error solution. The summary structure is constructed by dividing the in coming streams into fixed-size blocks. The algorithms reduce the maximum per node transmission to $polylogN$ (opposed to N for transmitting the raw data).

- **Performance** The algorithms demonstrate the scalability of the algorithms with respect to the data volume, approximation factor, and the number of sites. All three algorithms can greatly reduce the total as well as per node transmission, especially with larger-scale data.

Order Statistics Computation on High-Speed Data Streams

- **Model** I solve the problem of approximate quantile and biased-quantile for high-speed data streams.

- **Algorithm** I propose a fast algorithm for computing approximate quantiles in high speed data streams with deterministic error bounds. The algorithm uses simple block-wise merge and sample operations. Overall, the proposed algorithm achieves a per-element update computational cost of $O(log(1/\epsilon \log(\epsilon N)))$ for approximation factor ϵ and stream size N (N is not know beforehand). In addition, I proposed an efficient algorithm for computing approximate biased quantiles in large data streams. The algorithm is based on a novel piece-wise uniform sampling technique which computes decomposable biased quantile summaries on fixed size blocks of the incoming data stream. The algorithm is computationally efficient, does not assume prior knowledge of the stream sizes or the range of data values in the streams, and is applicable to distributed data stream system. In practice, the algorithm is able to efficiently maintain summaries over large data streams with

over tens of millions of observations.

- **Performance** Experiments demonstrate that the algorithms are able to efficiently maintain summaries over large data streams with over tens of millions of observations.

1.6 Organization

The rest of the thesis is organized as follows:

- **Chapter 2** presents the Minimum Segmentation model and algorithms for solving the problem of inferring segmentation structure of a single recombinant strain.
- **Chapter 3** presents the Minimum Mosaic model and algorithms for solving the problem of inferring mosaic structure of a recombinant strain panel.
- **Chapter 4** presents the algorithms for approximate K-Median clustering for sensor network data.
- **Chapter 5** presents the algorithms for approximate quantiles and biased-quantiles computation high-speed data streams.
- **Chapter 6** concludes with the major results of the thesis and discusses problem areas for future research.

Chapter 2

Inferring Segmentation Structure of Recombinant Genotype Sequences

2.1 Introduction

Recombination plays an important role in shaping the genetic variations present in current-day populations. Understanding the genetic variations and the genetic basis of traits is crucial for disease association studies. We assume an evolution model (previously proposed and studied in (Ukkonen (2002); Wu and Gusfield (2007))) where a population is evolved from a small number of founder sequences. A real-world biological scenario is the Collaborative Cross (CC). The CC (Churchill et al. (2004); Threadgill et al. (2002)) is a large panel of 1000 recombinant inbred (RI) mouse strains that were generated from a funnel breeding scheme initiated with a set of 8 founder strains followed by 20 generations of inbreeding. These 8 genetically diverse founder strains capture nearly 90% of the known variations present in the laboratory mouse. The resulting RI strains have a population structure that randomizes the known genetic variation, which provide unparallel power for disease association studies.

Given a set of founder haplotype sequences and a randomization of these haplotypes during interbreeding, a sequence in a derived line from a population of lines like the CC is composed of segments from the founders. It is of great interest to identify and

label these segments according to their contributing founder. Although the segmentation for a haplotype sequence may be straightforward to compute, in many cases the sequence to be segmented is a genotype sequence for which the two haplotypes are not completely distinct and they may have different segmentations. For example, the mice generated during the intermediate generations in the CC funnel are genotyped to obtain the genotype sequences each of which contains two different haplotypes in each line.

I study the segmentation problem of genotype sequences with the optimization for the minimum number of segments contained in the two associated haplotypes. Furthermore, I extend this basic model to include additional biologically-motivated constraints as well as noise. Since each autosome undergoes, on average, one recombination per meiosis, it is expected that the number of founder switches per haplotype at a given generation of breeding are comparable. Moreover, noise may exist in the founder sequences as well as the genotype sequence to be segmented. Sources of the noise are both technical and biological. They include point mutations, gene conversions, genotyping errors, etc. In addition to noise, missing genotyping values are also very common in these datasets.

2.2 Related Work

Similar but different models were studied in (Ukkonen (2002); Wu and Gusfield (2007); Mott et al. (2000)). Ukkonen (Ukkonen (2002)) first proposed the founder set reconstruction problem under the assumption that the sample set is evolved from a small set of founders. A dynamic programming algorithm was proposed which computes a minimum number of founders with a given set of sample haplotype sequences, where a segmentation of all the sequences in the sample set can be derived which contains the minimum number of founder switches. Wu and Gusfield (Wu and Gusfield (2007)) proposed improved polynomial time algorithms for haplotype as well as genotype sample sequences for the special case where there are only two founders. Different from these

models, the genotype sequence segmentation problem studied in this thesis assumes that the set of founder sequences are already known, and the main focus is on inferring the segmentation structure for genotype sequences, with the consideration of biologically-relevant constraints and noise present in the data. The genotype sequence segmentation problem is very important for studying the ancestral polymorphism structure in experimental recombinant strains such as PreCC strains (strains from intermediate generations in the CC funnel which are not fully inbred).

Besides the combinatorial analysis of founder set reconstruction problem presented in (Ukkonen (2002)) and (Wu and Gusfield (2007)), another line of related work focuses on probabilistic inference of SNP origins. Mott (Mott et al. (2000)) et al. proposed an HMM-based algorithm for inferring the probability of each founder pair as the origin for each locus, assuming the founder sequences are known beforehand. The founder origin probability distribution is computed for outbred animal stocks as input for subsequent quantitative trait loci (QTL) mapping. Different from locus-based founder origin estimation in (Mott et al. (2000)), the genotype sequence segmentation problem explicitly derives all the possible segmentation structures with certain biological meaningful optimization criterion.

Other related work of analyzing the genetic variation structure of the genome sequences include identifying haplotype blocks (Dally et al. (2001); Gabriel et al. (2002); Schwartz et al. (2003)), computing the phylogenies (Gusfield (2002); Gusfield et al. (2004)), etc.

2.3 The Minimum Segmentation Problem

Assume that we have a set of founding haplotypes $FS = \{F_1, \ldots, F_n, \ldots, F_N\}$. Each haplotype sequence is of length L: $F_n = f_1^n f_l^n \ldots f_L^n$, where $f_l^n \in \{0, 1\}$. Given an input sequence from a population which is derived exclusively from the founder set FS,

the problem is to find a possible segmentation of the sequence, where each segment is inherited from the corresponding region of one of the founders. I first explain the simple case where the input sequence is a haplotype, and then investigate the more interesting case where the input is a genotype sequence.

Given a haplotype sequence, $H = h_1 \dots h_L$, ($h_l \in \{0, 1\}$), a segment of H is denoted as $\overline{H}_k = h_{s_k} h_{s_k+1} h_{s_k+L_k-1}$, where s_k is the starting position of $\overline{H}_k$, and L_k is the length of $\overline{H}_k$. A segmentation of H divides the entire sequence into an ordered list of disjoint segments $Seg(H) = \{\overline{H}_1, \dots, \overline{H}_k, \dots, \overline{H}_K\}$, where each segment $\overline{H}_k$ is identical to the corresponding region of one of the founders and K is the number of segments in $Seg(H)$. In other words, for each segment $\overline{H}_k = h_{s_k} h_{s_k+1} h_{s_k+L_k-1}$, there exists a founder $F_n = f_1^n f_l^n \dots f_L^n$ such that $h_{s_k+l_i} = f_{s_k+l_i}^n$, for $l_i = 0, 1, \dots, L_k - 1$. Furthermore, a minimum segmentation is defined as the segmentation which contains the minimum number of segments. The minimum segmentation is denoted as $MinSeg(H) = \{\overline{H}_1, \dots, \overline{H}_{K_{min}}\}$, where $K_{min} = |MinSeg(H)|$ is the number of segments in $MinSeg(H)$.

If the input is a genotype sequence, it represents two copies of different haplotype sequences, H_a and H_b. Assume that the genotype sequence is $G = g_1 \dots g_L$, where $g_l \in \{0, 1, 2\}$. A site l is *homozygous* if $g_l = 0$ ($h_l^a = h_l^b = 0$) or $g_l = 1$ ($h_l^a = h_l^b = 1$); a site l is *heterozygous* if H_a and H_b take different alleles, in which case, $g_l = 2$. The process of determining whether $h_l^a = 0, h_l^b = 1$ or $h_l^a = 1, h_l^b = 0$ for a heterozygous site l is called *phasing*. The procedure of determining the two haplotype sequences from the genotype sequence by phasing all the heterozygous sites is called *Haplotype Inference*. For the genotype input case, a segmentation $Seg(G)$ consists of segmentations for both haplotype sequences: $Seg_a(H_a)$ and $Seg_b(H_b)$. The number of segments in $Seg(G)$ is the sum of the numbers of segments in $Seg_a(H_a)$ and $Seg_b(H_b)$: $|Seg(G)| = |Seg_a(H_a)| + |Seg_b(H_b)|$. The minimum segmentation is the segmentation which contains the minimum total number of segments: $|MinSeg(G)| = min\{|Seg(G)|\}$. Let $MinSeg(G) = \{Seg_a^*(H_a), Seg_b^*(H_b)\}$.

I develop efficient algorithms for the minimum segmentation problem especially for the genotype input case. In addition to the basic models, there are other issues which need to be considered, such as genotyping errors, point mutations, missing values, the balance of the number of segments in both haplotypes, etc. I will explain later how these biological constraints and noise are modeled in the solutions.

Solutions for Haplotype Input: Computing the minimum segmentation for the haplotype input sequence is relatively easy and has been discussed in previous studies (Wu and Gusfield (2007); Ukkonen (2002)). A simple greedy algorithm can be applied to compute a minimum segmentation solution by scanning from left to right. Assume that the current site is i (initially it is site 1), and we have a minimum segmentation solution for the part of the input sequence from site 1 to site i. Starting from site i, we try to find the segment shared by the input sequence and one of the founders which extends furthest to the right. This greedy algorithm generates one of the minimum segmentation solutions.

A graph-based dynamic programming algorithm can be used to compute all minimum segmentation solutions given the input haplotype sequence and the founder set. At a high level, all maximal shared intervals are first computed between the input sequence and each founder sequence. The maximal shared interval between the input sequence and founder n is a region where the input sequence is exactly the same as founder n. Each shared interval is considered as a node and two intervals are connected with an edge if they overlap. In this way, a minimum segmentation solution corresponds to a shortest path from a node starting at the first site to a node ending at the last site. The complete set of the shortest paths can be computed, which are all possible minimum segmentation solutions.

2.4 Solutions for Genotype Input

The greedy algorithm and the graph-based algorithm for segmenting haplotype input sequences cannot be easily applied on genotype input. The major issue is that the exact sequences of the two haplotypes are not known due to the multiple possible allele pairs at heterozygous sites. Second, the minimum segmentation solution for the genotype may not consist of the minimum segmentation solutions for each haplotype sequence.

In the following discussion, I describe two dynamic programming algorithms for solving the minimum segmentation problem for genotype input sequences. The first algorithm considers each site separately, the second algorithm considers a region of sites simultaneously, and is thus more efficient.

Site-based Dynamic Programming Algorithm: For each site l, the possible founders for the two haplotype sequences H_a and H_b are considered. If site l is a homozygous site, assuming $g_l = 0$ (without loss of generality), we have $h_l^a = h_l^b = 0$. Let $of^{a,l}$ be the original founder where h_l^a was inherited from at site l. Then $of^{a,l}$ must be one of the founders which also take 0 at site l: $of^{a,l} \in \{F_n | f_l^n = 0\}$. Similarly, we have the founder where h_l^b was inherited from as: $of^{b,l} \in \{F_n | f_l^n = 0\}$. Let $fp^l = \langle of^{a,l}, of^{b,l} \rangle$ denote the possible founder pair at site l, we have the set of all possible founder pairs as $FP^l = \{fp^l | fp^l \in \{F_n | f_l^n = 0\} \times \{F_n | f_l^n = 0\}\}$. If site l is a heterozygous site where $g_l = 2$, there are two possibilities: $h_l^a = 1 \wedge h_l^b = 0$ or $h_l^a = 0 \wedge h_l^b = 1$. Therefore, the possible founder pairs for heterozygous site l is $FP^l = \{fp^l | fp^l \in \{F_n | f_l^n = 0\} \times \{F_n | f_l^n = 1\} \cup \{F_n | f_l^n = 1\} \times \{F_n | f_l^n = 0\}\}$. The founder pair set FP^l is computed for each site l.

Assigning a founder pair from FP^l to each site l generates a segmentation of the input genotype sequence. The number of segments of both haplotypes (of the genotype) are the total number of *founder switches* between founder pairs of every consecutive sites plus 2. Consider two neighboring sites l and $l+1$. If the corresponding founder pairs are $fp_{q_l}^l = \langle of_{q_l}^{a,l}, of_{q_l}^{b,l} \rangle$ $(1 \leq q_l \leq |FP^l|)$ and $fp_{q_{l+1}}^{l+1} = \langle of_{q_{l+1}}^{a,l}, of_{q_{l+1}}^{b,l} \rangle$

$(1 \leq q_{l+1} \leq |FP^{l+1}|)$, the number of founder switches between these two founder pairs $FounderSwitch(fp^{l}_{q_l}, fp^{l+1}_{q_{l+1}})$ can be computed as:

$$FounderSwitch(fp^{l}_{q_l}, fp^{l+1}_{q_{l+1}}) = \begin{cases} 0: & \text{if} \quad of^{a,l}_{q_l} = of^{a,l+1}_{q_{l+1}} \wedge of^{b,l}_{q_l} = of^{b,l+1}_{q_{l+1}} \\ 1: & \text{if} \quad of^{a,l}_{q_l} = of^{a,l+1}_{q_{l+1}} \wedge of^{b,l}_{q_l} \neq of^{b,l+1}_{q_{l+1}} \quad \text{or} \\ & \quad\quad of^{a,l}_{q_l} \neq of^{a,l+1}_{q_{l+1}} \wedge of^{b,l}_{q_l} = of^{b,l+1}_{q_{l+1}} \\ 2: & \text{if} \quad of^{a,l}_{q_l} \neq of^{a,l+1}_{q_{l+1}} \wedge of^{b,l}_{q_l} \neq of^{b,l+1}_{q_{l+1}} \end{cases} \tag{2.1}$$

Let $K_{min}(g_1 \dots g_{l-1}|fp^{l}_{q_l})$ be the minimum number of segments in any segmentation solution over the subsequence $g_1 \dots g_l$ which at site l takes the founder pair $fp^{l}_{q_l}$. The minimum number of segments over the entire genotype sequence $K_{min}(g_1 \dots g_L)$ can be computed as:

$$K_{min}(g_1 \dots g_L) = min_{fp^{L}_{q_L} \in FP^{L}}\{K_{min}(g_1 \dots g_{L-1}|fp^{L}_{q_L})\} \tag{2.2}$$

The main recurrence of the dynamic programming algorithm is as follows:

$$\begin{aligned} K_{min}(g_1 \dots g_{l-1}|fp^{l}_{q_l}) = \quad & min_{fp^{l-1}_{q_{l-1}} \in FP^{l-1}}\{K_{min}(g_1 \dots g_{l-2}|fp^{l-1}_{q_{l-1}}) + \\ & FounderSwitch(fp^{l-1}_{q_{l-1}}, fp^{l}_{q_l})\} \end{aligned} \tag{2.3}$$

And initially,

$$K_{min}(\Phi|fp^{1}_{q_1}) = 2, \forall fp^{1}_{q_1} \in FP^{1} \tag{2.4}$$

The solutions for this dynamic programming problem can be easily computed by populating a table T of L rows where row l has at most $|FP^{l}|$ entries. The entry $T(l, q_l)$, $1 \leq q_l \leq |FP^{l}|$ is filled with $K_{min}(g_1 \dots g_{l-1}|fp^{l}_{q_l})$ during the computation. Row

1 is initialized according to Eq.(4), and row $i+1$ is computed after row i. During the computation of $T(l, q_l)$ according to Eq.(3), backtracking pointers from entry $T(l, q_l)$ to any $T(l-1, q_{l-1})$ are preserved where the minimum values are obtained. In this way, all the minimum segmentation solutions can be obtained.

There are at most N^2 founder pairs for each site l, *i.e.*, $|FP^l| \leq N^2, \forall l$. Therefore, the populated table is of size $O(LN^2)$. It takes constant time to compute $FounderSwitch$ $(fp^l_{q_l}, fp^{l+1}_{q_{l+1}})$, then filling out a single entry in the table takes $O(N^2)$ time. Therefore, the computational complexity for the entire algorithm is $O(LN^4)$. The space complexity is $O(LN^2)$. For very long sequences and a small number of founders, i.e., $L \gg N^4$, the algorithm has linear time and space complexity in terms of the length of the input sequence. If multiple backtrack pointers are kept for each entry while populating the table, all the minimum segmentation solutions can be obtained.

Region-based Dynamic Programming Algorithm: For very long sequences, a more efficient algorithm is proposed which considers a subregion of the entire sequence instead of one site at a time.

First consider the homozygous regions, which are the regions of homozygous sites between any two consecutive heterozygous sites. Within a homozygous region, both copies of the haplotype sequences are the same and the exact allele at each site can be inferred. Fig. 2.1 illustrates an example of a set of four founders $(F_1 - F_4)$ and a genotype input sequence G to be segmented. The length of each founder and the genotype sequence is 15, with 14 homozygous sites and 1 heterozygous site (site 10). The homozygous regions are $R_1 = [1, 9]$ and $R_2 = [11, 15]$. For each homozygous region, all the maximal shared intervals between each founder and the haplotype sequences are computed. A maximal shared interval Δ_i is an interval over which a haplotype and a founder shares the same allele at each site and the region cannot be extended further on either side. Each maximum shared interval is represented as a triple, for example, $\Delta_i : (I_i, H_a, F_n)$ is a maximal shared interval between haplotype H_a and founder F_n over

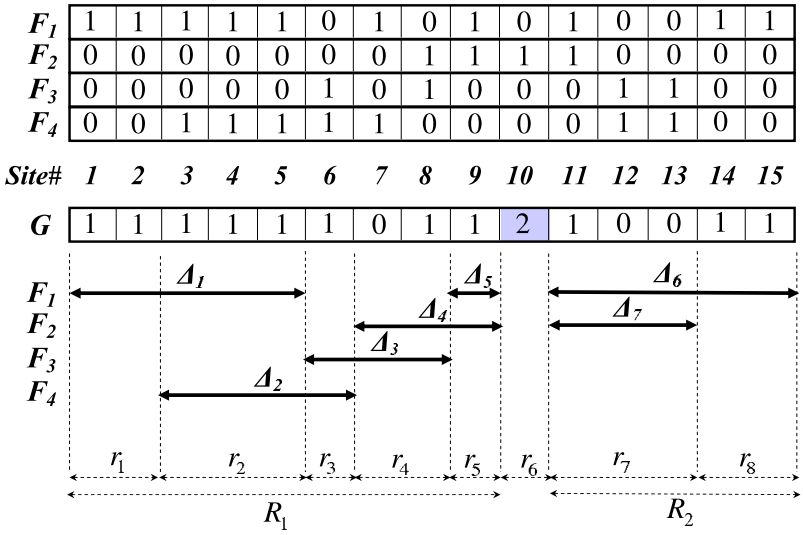


Figure 2.1: An example subregions. F_1-F_4 are four founder sequences. G is the genotype sequence to be segmented. There are 15 sites in all sequences, where site 10 is the only heterozygous site. $R_1 : [1, 9]$ and $R_2 : [11, 15]$ are the homozygous regions. Δ_1-Δ_5 are the maximal shared intervals in R_1. Δ_6 and Δ_7 are the maximal shared intervals in R_2. $r_1 - r_8$ are the subregions for the entire sequence, out of which r_6 is the heterozygous subregions, and the remaining are the homozygous subregions.

interval I_i. Since both haplotypes are the same, a maximal shared interval for haplotype H_a is also a maximal shared interval for haplotype H_b, therefore, the maximal shared interval for the homozygous regions can also be represented as $\Delta_i : (I_i, *, F_n)$. In Fig. 2.1, $\Delta_1 - \Delta_5$ are the maximal shared intervals within region R_1 for both haplotype sequences. Each homozygous region R_j is then divided into a set of subregions using the two end points of all maximal shared intervals inside R_j. For example, in Fig. 2.1, R_1 is divided into subregions r_1, r_2, r_3, r_4, and r_5. If each heterozygous site is considered as a 1-site subregion (e.g. r_6 in Fig. 2.1), together with all the subregions for the homozygous regions, $\{r_p\}$ represents a complete set of subregions which cover the entire sequence (e.g., $r_1 - r_8$ in Fig. 2.1).

For each homozygous subregion r_p, let $fp^{r_p} = \langle of^{a,r_p}, of^{b,r_p} \rangle$ be a possible founder pair for subregion r_p. The set of possible founder pairs is $FP^{r_p} = \{\langle of^{a,r_p}, of^{b,r_p} \rangle | \exists \Delta_{i_1} = (I_{i_1}, *, of^{a,r_p}), \Delta_{i_2} = (I_{i_2}, *, of^{b,r_p})$, where $I_{i_1} \supseteq r_p, I_{i_2} \supseteq r_p\}$. For example, the founder pair for the subregion r_2 in Fig. 2.1 could be $\langle F_1, F_1 \rangle$, or $\langle F_1, F_2 \rangle$, or $\langle F_2, F_1 \rangle$,

or $\langle F_2, F_2 \rangle$. For each heterozygous subregion which is composed of a heterozygous site l, since h_a^l and h_b^l take different alleles, any possible founder pair should consist of a founder taking allele 1 and a founder taking allele 0. For example, in Fig. 2.1, the possible founder pairs for r_6 are $\langle F_1, F_2 \rangle$, $\langle F_2, F_1 \rangle$, $\langle F_2, F_3 \rangle$, $\langle F_2, F_3 \rangle$, $\langle F_2, F_4 \rangle$, and $\langle F_4, F_2 \rangle$.

Instead of considering each site, each subregion is considered as a unit in the dynamic programming solution. Assign $fp_{q_p}^{r_p} = \langle of_{q_p}^{a,r_p}, of_{q_p}^{b,r_p} \rangle$ to be the founder pair for subregion r_p, where $1 \leq q_p \leq |FP^{r_p}|$, and $fp_{q_{p+1}}^{r_{p+1}} = \langle of_{q_{p+1}}^{a,r_p}, of_{q_{p+1}}^{b,r_p} \rangle$ to be the founder pair for subregion r_{p+1}, where $1 \leq q_{p+1} \leq |FP^{r_{p+1}}|$. Similarly, the number of founder switches between $fp_{q_p}^{r_p}, fp_{q_{p+1}}^{r_{p+1}}$ is counted as:

$$FounderSwitch(fp_{q_p}^{r_p}, fp_{q_{p+1}}^{r_{p+1}}) = \begin{cases} 0: & \text{if} \quad of_{q_p}^{a,r_p} = of_{q_{p+1}}^{a,r_{p+1}} \wedge of_{q_p}^{b,r_p} = of_{q_{p+1}}^{b,r_{p+1}} \\ 1: & \text{if} \quad of_{q_p}^{a,r_p} = of_{q_{p+1}}^{a,r_{p+1}} \wedge of_{q_p}^{b,r_p} \neq of_{q_{p+1}}^{b,r_{p+1}} \\ & \quad\quad of_{q_p}^{a,r_p} \neq of_{q_{p+1}}^{a,r_{p+1}} \wedge of_{q_p}^{b,r_p} = of_{q_{p+1}}^{b,r_{p+1}} \\ 2: & \text{if} \quad of_{q_p}^{a,r_p} \neq of_{q_{p+1}}^{a,r_{p+1}} \wedge of_{q_p}^{b,r_p} \neq of_{q_{p+1}}^{b,r_{p+1}} \end{cases} \tag{2.5}$$

Let $K_{min}(r_1 \ldots r_{p-1} | fp_{q_p}^{r_p})$ be the minimum number of segments in any segmentation solution over the subsequence covered by $r_1 \ldots r_p$ which takes the founder pair $fp_{q_p}^{r_p}$ at subregion r_p. The minimum number of segments over the entire genotype sequence $K_{min}(r_1 \ldots r_P)$ where r_P is the last subregion can be computed as:

$$K_{min}(r_1 \ldots r_P) = min_{fp_{q_P}^{r_P} \in FP^{r_P}} \{K_{min}(r_1 \ldots r_{P-1} | fp_{q_P}^{r_P})\} \tag{2.6}$$

The main recurrence of the dynamic-programming algorithm is as follows:

$$\begin{aligned} K_{min}(r_1 \ldots r_{p-1} | fp_{q_p}^{r_p}) = \quad & min_{fp_{q_{p-1}}^{r_{p-1}} \in FP^{p-1}} \{K_{min}(r_1 \ldots r_{p-2} | fp_{q_{p-1}}^{r_{p-1}}) + \\ & FounderSwitch(fp_{q_{p-1}}^{r_{p-1}}, fp_{q_p}^{r_p})\} \end{aligned} \tag{2.7}$$

And initially,

$$K_{min}(\Phi | fp_{q_1}^{r_1}) = 2, \forall fp_{q_1}^{r_1} \in FP^{r_1} \tag{2.8}$$

This dynamic programming problem can also be solved by populating a table T which contains P rows where row p has at most $|FP^{r_p}|$ entries. Entry $T(p, q_p), 1 \leq q_p \leq FP^{r_p}$ is filled with $K_{min}(r_1 \dots r_{p-1}|fp_{q_p}^{r_p})$ during the computation. There are at most N^2 founder pairs for each subregion r_p, *i.e.*, $|FP^{r_p}| \leq N^2$. The populated table is of size $O(PN^2)$. The computation of all the maximal shared intervals is $O(LN)$, and filling out each entry in the table costs $O(N^2)$. Thus the computational complexity of region-based dynamic programming algorithm is $O(LN + PN^4)$. Compared with the site-based algorithm which has a time complexity of $O(LN^4)$, if P is much smaller than L, the running time can be greatly reduced, especially for large L.

2.4.1 Enforcing the Constraints and Modeling Noise

Comparable Number of Founder Switches on Both Haplotypes: During meiosis autosomes undergo one recombination per chromosome on average. Thus, during the development of an recombinant inbred-line (RIL), one expects that the number of founder switches per haplotype at each generation to be comparable.

During each mating in the evolving history, each of the two haplotypes may be generated by a new recombination. Therefore, it is expected that for the given genotype to be segmented, the number of segments for the two haplotype sequences are comparable.

For a segmentation $Seg(G)$ of genotype sequence G, which is composed of a segmentation $Seg_a(H_a)$ on haplotype H_a and a segmentation $Seg_b(H_b)$ on haplotype H_b, an extra constraint is enforced on the minimum segmentation as follows: the difference of the numbers of the segments in the two haplotypes is no more than a threshold α: $||Seg_a(H_a)| - |Seg_b(H_b)|| < \alpha$, where α is a nonnegative integer. The definition of the minimum segmentation for G is then modified as: $|MinSeg(G)| = min\{|Seg(G)|\}$, where $MinSeg(G) = \{Seg_a^*(H_a), Seg_b^*(H_b)\}$ and $||Seg_a^*(H_a)| - |Seg_b^*(H_b)|| < \alpha$.

To compute the minimum segmentation solution with constraints, I propose an effi-

cient heuristic that prunes out solutions that do not satisfy the constraint before they are fully computed during the table population process. This greatly reduces the computation, especially when there are a lot of minimum segmentation solutions that do not satisfy the constraint. I will explain how it works with the region-based algorithm. Assume that we have a founder set $\{F^1, \ldots, F^N\}$, and a genotype sequence G. For any minimum segmentation solution $MinSeg(G) = \{Seg_a^*(H_a), Seg_b^*(H_b)\}$, the following lemma holds:

Lemma 2.4.1. *For any homozygous region R on G, and any minimum segmentation solution $\{Seg_a^*(H_a), Seg_b^*(H_b)\}$, let the set of segments in $Seg_a^*(H_a)$ which completely fall inside R be $Seg_a^*(H_a) \cap R$, and the set of segments in $Seg_b^*(H_b)$ which completely fall inside R be $Seg_b^*(H_b) \cap R$, then we have*

$$||Seg_a^*(H_a) \cap R| - |Seg_b^*(H_b) \cap R|| \leq 2 \tag{2.9}$$

Proof. Consider a homozygous region R between two heterozygous sites l and l'. Let S_a be the segment in $Seg_a^*(H_a)$ containing site l, S_b be the segment in $Seg_b^*(H_b)$ containing site l, S'_a be the segment in $Seg_a^*(H_a)$ containing site l', and S'_b be the segment in $Seg_b^*(H_b)$ containing site l', as shown in Fig. 2. Let the region covered by the segments in $Seg_a^*(H_a) \cap R$ be R_a, the region covered by the segments in $Seg_b^*(H_b) \cap R$ be R_b, and the common region be $R_c = R_a \cap R_b$ (see Fig. 2).

First we take a look at cases when R_c is not empty (Fig. 2(a),(b),(c)). Let $K_{min}(R_c)$ be the minimum number of segments which can cover R_c on one haplotype (both haplotypes are the same over R_c), we have

$$K_{min}(R_c) \leq |Seg_a^*(H_a) \cap R| \leq K_{min}(R_c) + 2, \tag{2.10}$$

and

$$K_{min}(R_c) \leq |Seg_b^*(H_b) \cap R| \leq K_{min}(R_c) + 2 \tag{2.11}$$

The left inequalities in (10) and (11) are obvious. The right inequalities, $|Seg_a^*(H_a) \cap R| \leq K_{min}(R_c) + 2$ and $|Seg_b^*(H_b) \cap R| \leq K_{min}(R_c) + 2$, can be proved as follows. Fig. 2(a),(b) and (c) show the three different cases when R_c is not empty. Let us first take a look at the case in Fig. 2(a), where $R_b = R_c \subset R_a$. The remaining part in R_a which is not in R_c can be covered using two segments: S^1_{remain} from the same founder as S_b, and S^2_{remain} from the same founder as S'_b. The above observation implies that there exists a set with no more than $K_{min}(R_c)+2$ segments covering R_a. Therefore, $|Seg_a^*(H_a) \cap R| \leq K_{min}(R_c)+2$. Similarly, we have $|Seg_b^*(H_b) \cap R| = K_{min}(R_c) < K_{min}(R_c)+2$. A similar case is in Fig. 2(b) where $R_c \subset R_a$ and $R_c \subset R_b$. We have $|Seg_a^*(H_a) \cap R| \leq K_{min}(R_c)+1$, and $|Seg_b^*(H_b) \cap R| \leq K_{min}(R_c) + 1$. In Fig. 2(c), we have $|Seg_a^*(H_a) \cap R| = K_{min}(R_c)$, and $|Seg_a^*(H_a) \cap R| = K_{min}(R_c)$. □

Lemma 2.4.2. *For a genotype sequence G containing Z heterozygous sites, any of its minimum segmentation $\{Seg_a^*(H_a), Seg_b^*(H_b)\}$ satisfies:*

$$||Seg_a^*(H_a)| - |Seg_b^*(H_b)|| \leq 3Z + 1 \tag{2.12}$$

Proof. According to Lemma 1, we know that the number of segments in $Seg_a^*(H_a)$ which completely fall inside homozygous regions and the number of segments in $Seg_b^*(H_b)$ which completely fall inside homozygous regions can at most differ by $2(Z+1)$. The remaining segments are those which cover the heterozygous sites. The number of such segments in $Seg_a^*(H_a)$ and $Seg_b^*(H_b)$ can differ by at most $Z-1$. Together, the difference between the number of segments in $Seg_a^*(H_a)$ and $Seg_b^*(H_b)$ can be at most $3Z+1$.

□

Lemma 2 can be used in the dynamic programming algorithm during the process of populating the table. Assume currently the entry $T(p, q_p)$ is to be filled, *i.e.*, $K_{min}(r_1 \ldots r_{p-1}|fp_{q_p}^{r_p})$ is computed according to Eq.(2.7), which is the minimum segmentation for the subsequence from r_1 to r_p with $fp_{q_p}^{r_p}$ as the founder pair for r_p. In

addition to computing the minimum segmentation, the difference between the number of segments over two haplotypes in the minimum segmentation is also calculated, $\delta(p, q_p) = ||Seg_a^*(H_a[r_1, r_p])| - |Seg_b^*(H_b[r_1, r_p])||$. Let the number of heterozygous sites in the remaining part of the sequence be $Z([r_{p+1}, r_P])$. Then if $\delta(p, q_p) - (3Z([r_{p+1}, r_P]) + 1) > \alpha$, according to Lemma 2, we know that the corresponding solution will not be able to generate a minimum segmentation solution for the entire sequence where the difference between the number of segments on both haplotypes is less than α. If the minimum segmentation solution considered is from $T(p-1, q_{p-1})$ to $T(p, q_p)$, then if $\delta(p, q_p) - (3Z([r_{p+1}, r_P]) + 1) > \alpha$, the backtrack pointer from $T(p, q_p)$ to $T(p-1, q_{p-1})$ will not be added.

Modeling Point Mutations, Genotyping Errors and Gene Conversions: There are both biological and technical resources of noise in genotyping, which include point mutations and gene conversions, and genotyping errors. These potential noise sources in the data are considered in the segmentation algorithms.

A point mutation or genotyping error can be treated as a single site mismatch that falls within a shared interval between a copy of haplotype sequence and a founder sequence. A gene conversion can be treated as a short sequence of mismatches that fall within a shared interval between a copy of haplotype sequence and a founder sequence. In the following, I explain how these noise sources are modeled in the region-based dynamic programming algorithm. First consider the point mutation, gene conversion, and genotyping error that happen inside a homogeneous range. During the computation of maximal shared intervals between sequence G and each founder F_n over a homozygous region R, assume that we have two maximal intervals Δ_1, Δ_2 between G and F_n which are over the intervals $I_1 = [b_1, e_1]$ and $I_2 = [b_2, e_2]$. We know that I_1 and I_2 are not overlapping or touching. Let I_1 be the interval on the left, *i.e.*, $e_1 < b_2 - 1$. If $e_1 = b_2 - 2$, then there is a single mismatch at site $e_1 + 1$ within the combined region $[b_1, e_2]$. Assume that both I_1 and I_2 are of enough length, then this single mismatch may be a point mu-

tation or genotyping error. Therefore, we create another shared interval Δ_3 between G and F_n over the single-site interval $I_3 = [e_1 + 1, e_1 + 1]$. This interval has a probability of $\beta < 1$ to be a shared interval between G and F_n. β is defined as the probability of a single mismatch inside a shared interval being a point mutation or genotyping error. Similarly, if $e_1 < b_2 - 2$ but $b_2 - 1 - e_1 < gc$, which means the gap between interval I_1 and I_2 is shorter than a maximal possible length gc of a typical gene conversion, then this short sequence of mismatches may be a gene conversion, assuming both I_1 and I_2 are of enough length. We create another shared interval Δ_4 between G and F_n over the short interval $I_4 = [e_1 + 1, b_2 - 1]$. This interval has a probability of $\gamma < 1$ to be a shared interval between G and F_n. γ is defined as the probability of a short sequence of mismatches inside a shared interval being a gene conversion. The point mutation and genotyping error that happens at a heterozygous site can be processed in a similar manner, where we check whether the heterozygous site is a single mismatch falling into a shared interval, *i.e.* the shared interval to the left of the heterozygous site and to the right of the heterozygous site are from the same founder. The maximal shared intervals computed without considering noise are of probability 1. By modeling the noise using intervals with probability less than 1, the minimum segmentation solution can be computed with desired noise tolerance $\theta < 1$. When entry $T(p, q_p)$ in the table is to be filled, the accumulated probability is computed which is the multiplication of the probabilities of the intervals in the founder pairs on the minimum segmentations solution so far. Only the solution with the accumulated probability no less than θ are kept.

Modeling Missing Values: Besides noise and incorrect values, there can also be missing values in the data. Assume that the missing value is in founder F_n, at site l, and the value at the same site on the sequence G is not missing. If l is a homozygous site, f_l^n is filled out using g_l. If l is a heterozygous site, f_l^n is considered to be either 0 or 1 when the founder pairs are calculated for this heterozygous site. If the missing value is on G at site l, it is considered to be either 0 or 1, which means this site can be in a

maximal shared interval with any of the founder, no matter whether the founder has a missing value at the same site or not. In this way, the minimum segmentation solution can be generated with the smallest possible number of segments. The missing values in both founders and genotype sequences are filled with values in each solution (it may be filled with different values for different minimum segmentation solutions).

2.5 Experimental Results

The performance of the region-based dynamic programming algorithm is evaluated on the real and simulated data. As presented earlier, the region-based algorithm has less computational complexity than the site-based algorithm. I only demonstrate the results on region-based algorithm.

2.5.1 Datasets

Real Data

The real data consists of G2:F1 strains and Pre-CC strains from the Collaborative Cross project (Fig. 2.2). The Pre-CC strains are from generations G2:F5 to G2:F14. The 8 founders strains (Fig. 2.3) are chosen to maximize the genetic diversity in the resulting CC strains.

Synthetic Data

The set of the founder sequences $\{F_n\}$ and the genotype sequence G (corresponding two haplotype sequences H_a, H_b) are generated so that: (a) The set of founders are generated randomly, except that, at each site, there is at least a founder taking 0 and at least a founder taking 1, (b) The number of the heterozygous sites in G is $h\_rate \times L$ where h_rate is a parameter representing the occurrence rate of the heterozygous sites, L is the total number of sites, and (c) H_a and H_b are generated by randomly patching up n_seg

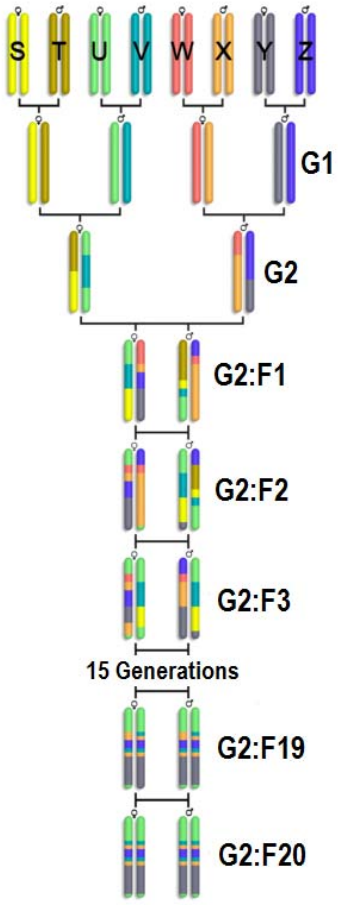


Figure 2.2: The Collaborative Cross breeding funnel. S, T, U, V, W, X, Y, and Z denote the 8 founders. The breeding funnel consists of 2 generations of crosses (G1 and G2), followed by 20 generations of inbreeding (G2:F1 - G2:F20)

random segments from the founders. Note that, the segmentation during the generation provides a lower bound on the number of segments in the minimum segmentation. The computed minimum segmentation may not have the same number of segments on both haplotypes.

The experiments are performed on an Intel Core 2 Duo 1.6GHz machine with 3GB memory.

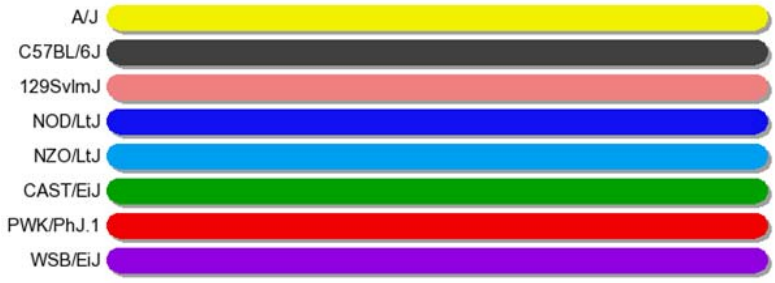


Figure 2.3: The 8 founder strains chosen for the Collaborative Cross breeding funnel shown in Fig. 2.2

2.5.2 Segmentation Results

The segmentation results on a G2:F1 animal (OR65f18) and a Pre-CC animal (13m72) are shown in Fig. 2.4 and Fig. 2.5, respectively.

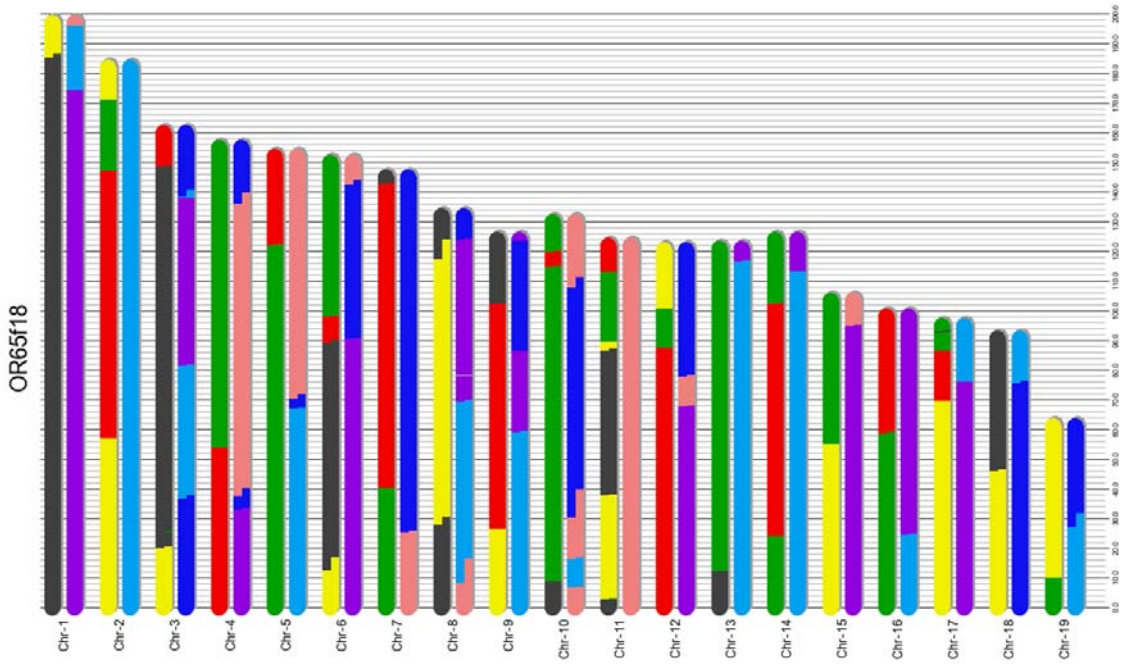


Figure 2.4: The segmentation result of the proposed algorithm on a G2:F1 animal OR65f18 from Collaborative Cross. The colors of different segments represent different founders shown in Fig. 2.3.

There are only limited number of segments on all 19 chromosomes of either G2:F1 mouse OR65f18 or Pre-CC mouse 13m72, which is consistent with the fact that there are only a few recombination events (mostly 1 or 2) during each mating. Furthermore,

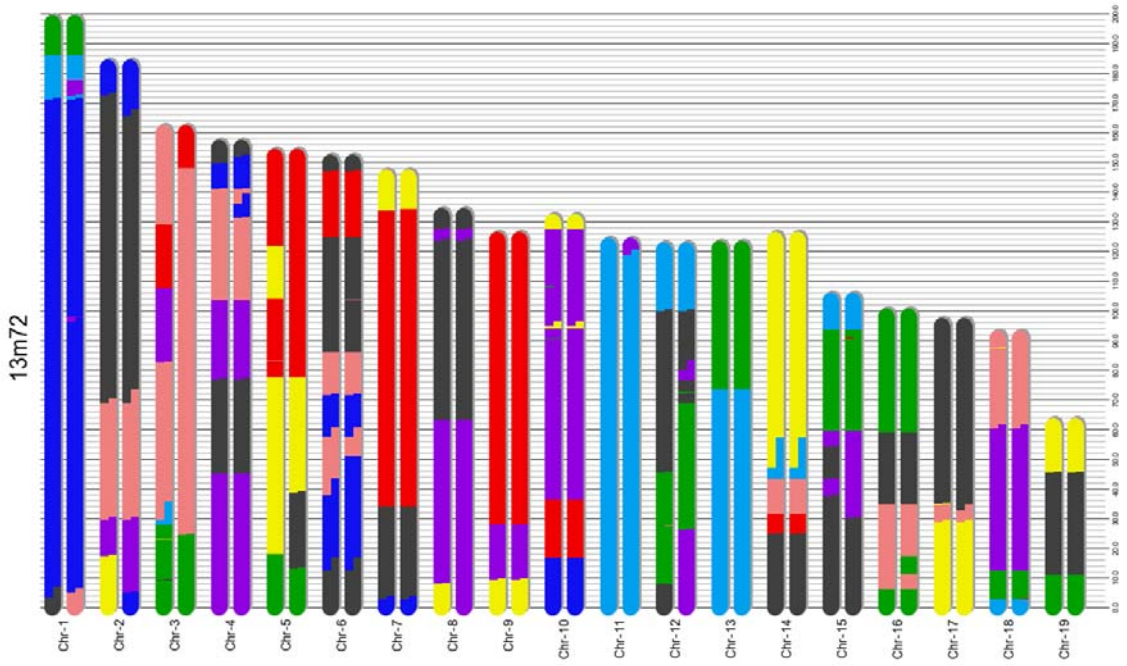


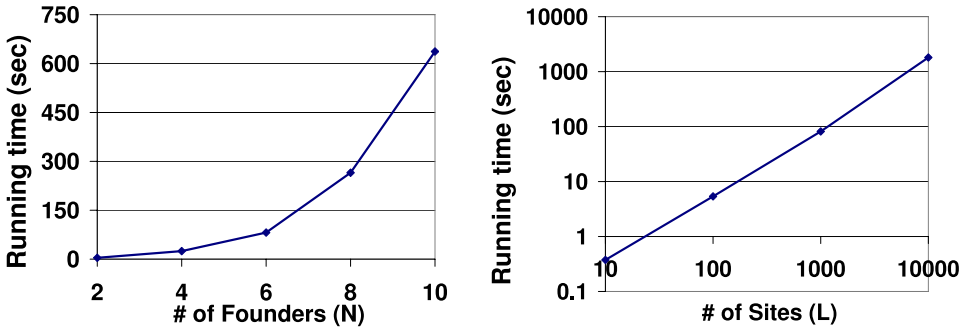
Figure 2.5: The segmentation result of the proposed algorithm on a Pre-CC animal 13m72 from Collaborative Cross. The colors of different segments represent different founders shown in Fig. 2.3.

the chromosomes of Pre-CC animal 13m72 are divided into finer fragmental structure, compared with those of the G2:F1 animal OR65f18. This is due to the fact that 13m72 has gone through more generations of breeding. The purpose of the inbreeding generations is to achieve the homozygosity in the final CC strains to guarantee the ability of reproductivity. By comparing Fig. 2.4 and Fig. 2.5, it can be observed that the Pre-CC animal 13m72 exhibits more homozygosity as expected compared to the G2:F1 animal OR65f18, as most of the segments on both haplotypes along the same subregion of 13m72 are of the same color (from the same founder thus identical).

2.5.3 Running Time

The running time performance is evaluated by varying two parameters: the number of founders (N) and the number of sites (L).

Fig. 2.6(a) highlights the running time by varying the number of founders from 2



(a) Time vs. N (b) Time vs. L

Figure 2.6: Running time with varying parameters.

to 10. Other parameters are fixed to be $L = 1000, n\_seg = 30, h\_rate = 0.01$. The complexity of the algorithm is $O(LN + PN^4)$, which is demonstrated by the superlinear increase in the running time with increasing N. Fig. 2.6(b) shows the running time with varying number of sites. All datasets contain 6 founders. n_seg for 10, 100, 1000, 5000, and 10000-site datasets are chosen to be 3, 8, 30, 150, 300, respectively. h_rate is 0.01 for all the datasets except for the 10-site set where the h_rate is set to be 0.1 to guarantee 1 heterozygous site. It can be observed that the running time is linear to the number of sites L, and super liner to the number of founders N.

2.5.4 Constraint on Segment Number Difference

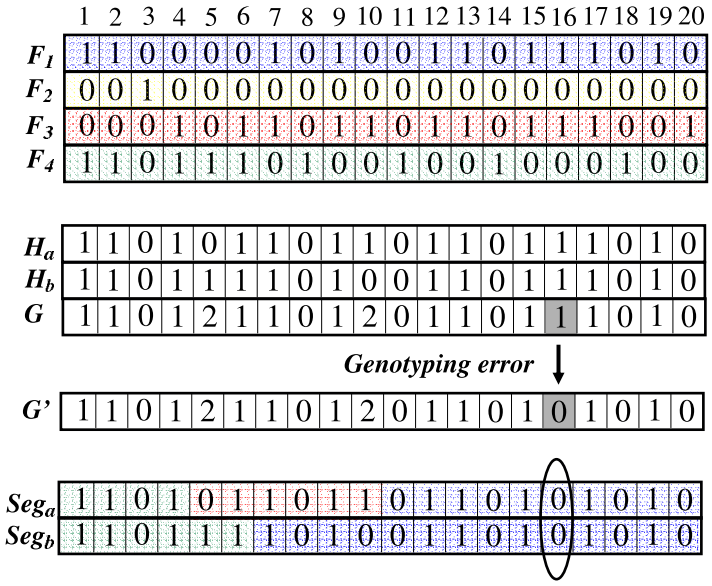
Table 2.1 shows the results on three datasets where constraints on the segment number difference on both haplotypes are applied. As shown in the table, enforcing these constraints greatly reduces the number of minimum segmentation solutions generated. For dataset #1, there are 28 minimum segmentation solutions where both haplotypes take the same number of segments. However, the number of solutions increases to 40 when the number of segments over the two haplotypes can differ by 1. Similarly, for dataset #2, the number of solutions increased 5 times. A similar trend is observed for dataset #3.

dataset	#1		#2			#3	
parameters	$N = 4, L = 20$ $n\_seg = 4, h = 0.1$		$N = 2, L = 50$ $n\_seg = 8, h = 0.05$			$N = 6, L = 20$ $n\_seg = 6, h = 0.05$	
α	0	1	0	1	2	0	1
# solutions	28	40	92	276	460	6	356

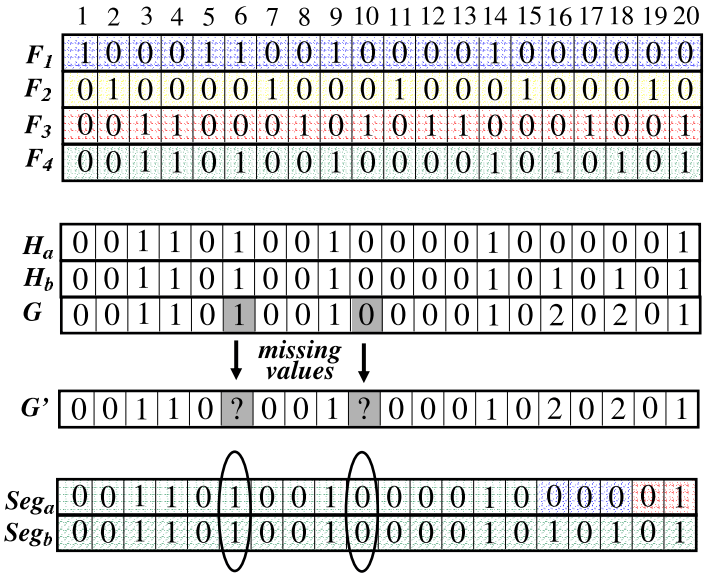
Table 2.1: Effect of Enforcing the Constraint on the Segment Number Difference

2.5.5 Error Tolerance

The algorithms are tested on simulated data with point mutations, genotyping errors, and missing values. Fig. 2.7 shows two example segmentation results. In Fig. 2.7(a), the dataset contains four founders $\{F_1, F_2, F_3, F_4\}$ each of which has 20 sites. The two copies of the haplotypes H_a and H_b, and the corresponding genotype G is shown in the figure as the ground truth. A random site is chosen to take a genotyping error. The resulting genotype G' has a genotyping error (1 is mistaken as 0) at site 16. The segmentation solution on G' is shown at the bottom of Fig. 2.7(a). Although F_1 does not match G at site 16, F_1 is still chosen as the founder in Seg_a and Seg_b, since site 16 is a single mismatch inside a long shared interval with F_1. Fig. 2.7(b) shows the result on a dataset with missing values. Two random sites (site 6 and site 10) are chosen to be the sites with missing values. As shown at the bottom of the figure, the algorithm still generates the correct minimum segmentation with the values at both sites filled in. Fig. 2.7 is better to be viewed in color.



(a) Data with genotyping error



(b) Data with missing values

Figure 2.7: Segmentation results on data with noise.

Chapter 3

Inferring Genome-wide Mosaic Structure

3.1 Introduction

Ancestral genetic recombination events play a critical role in shaping extant genomes. Characterizing the patterns of recombination (*e.g.* the recombination locations and rates), is a crucial step for reconstructing evolutionary histories, performing disease association mapping, and solving other population genetics problems.

During meiosis in diploid organisms, the two homologous chromosomes recombine and undergo a reduction division to form a haploid gamete, which contributes half of the genetic content to its offspring. This mixing of genomes leads to a mosaic chromosome (haplotype) structure composed of segments from each grandparent. The boundaries between the segments of each haplotype are referred to as the recombination breakpoints. Recombination breakpoints represent locations where the crossovers have occurred, either during the generation of the haplotype itself, or in previous generations.

The main goal is to infer the possible mosaic structures of a given set of related haplotypes. This is accomplished by finding a set of recombination breakpoints that divide the haplotypes into compatible blocks according to the Four-Gamete Test (FGT) (Hudson and Kaplan (1985)). The FGT states that, under the infinite-sites assumption (Hudson and Kaplan (1985)), all pairs of polymorphisms should co-occur in only three

out of their four possible configurations. Thus, when four configurations are observed in a pair of markers, it implies that either a recombination or a homoplastic event has occurred between them. I propose an efficient algorithm to solve the "Minimum Mosaic Problem", which finds the mosaic with the minimum number of breakpoints. The algorithm is suitable for genome-wide study.

3.2 Related Work

Algorithms have been developed for several related problems.

- *Estimation of recombination rate*

  Hudson and Kaplan (Hudson and Kaplan (1985)) proposed a lower bound (HK bound) of the minimum recombination events estimated using the FGT (Hudson and Kaplan (1985)). Their algorithm computes a minimum set of non-overlapping intervals where all pairs of SNPs in an interval are compatible according to the FGT. This number of intervals, less one, is the HK bound. Myers and Griffiths (Myers and Griffiths (2003)) proposed a tighter bound, RecMin. However, it is only computationally tractable to find the optimal RecMin in relatively small datasets. Different from RecMin or SHRUB, the problem studied in this thesis focuses on the mosaic structure of a set of sample sequences without explicitly computing the evolutionary history, assuming that the genomic structures of the sample sequences are of more interest. However, the breakpoints on the sample sequences may reflect possible recombination events that happened in previous generations.

- *Inferring haplotype block structure*

  In addition to estimating lower bounds on the number of recombinations, algorithms have also been proposed for partitioning haplotypes into blocks. There are

two common approaches. The first employs statistical linkage disequilibrium (LD) measures (Gabriel et al. (2002)) that assigns blocks to regions with high pairwise LD and with relatively low pairwise LD between blocks. The second class of approaches uses diversity-based methods (Patil et al. (2001)) that assigns blocks to regions of low sequence diversity.

- *Recombination detection*

  A plethora of methods have been developed for detecting the recombinations in the data. A comprehensive review was given in (Posada (2002)), which classifies the methods into four categories: a) distance methods, which search for inversions along the alignment of the distance patterns among sequences (G.F. (1998)); b) phylogenetic methods, which test whether phylogenies from different parts (usually adjacent sliding windows) of the genome result in discordant topologies (Hein (1990, 1993); N.C. and Holmes (1997); Holmes et al. (1999); Lole et al. (1999); Martin and Rybicki (2000)); c) compatibility methods, which is based on site-by-site analysis of the compatibility when the evolutionary history of two sites is congruent with the same tree (Drouin et al. (1999); Jakobsen et al. (1997)); d) substitution distribution methods, which detect recombination by examining the sequences either for a significant clustering of substitutions or for a fit to an expected statistical distribution (Maynard and Smith (1998); Stephens (1985); Worobey (2001)).

3.3 Problem Formulation

Suppose that we have a set of n haplotypes over m SNPs, represented by a binary data matrix $D = [d_{ij}]_{i=1..n,j=1..m}$. Row i in D corresponds to a haplotype h_i, and column j in D corresponds to a SNP s_j. Matrix entry d_{ij} is either 0 or 1, representing the majority allele or minority allele at SNP s_j respectively. Only crossover recombination events are

considered, and gene conversion and homoplasy are ignored (assuming they do not have a significant role).

Over any pair of SNPs s_j and $s_{j'}$, a haplotype takes one of four possible gametes 00, 01, 10, 11 (the combination of alleles at s_j and $s_{j'}$). The set of haplotypes taking 00, 01, 10, or 11 at SNP pair $(s_j, s_{j'})$ is denoted as $HapSet_{00}^{j,j'}$, $HapSet_{01}^{j,j'}$, $HapSet_{10}^{j,j'}$, $HapSet_{11}^{j,j'}$ respectively. If all four sets are nonempty, according to the FGT (Hudson and Kaplan (1985)), a historical recombination event must have occurred between s_j and $s_{j'}$. In this case, we say that the SNP pair $(s_j, s_{j'})$ is incompatible. A recombination breakpoint is represented as a tuple (h_i, s_b), where the breakpoint locates on haplotype h_i between SNPs s_b and s_{b+1}. It is possible that multiple haplotypes may have breakpoints at the same location since they may "inherit" the breakpoint from a common ancestor sequence.

A compatible block of SNPs is defined as a continuous set of SNPs such that any two SNPs inside the block are compatible. Two SNP blocks are incompatible with each other if there exist two incompatible SNPs, one from each block.

A complete set of breakpoints creates a haplotype mosaic structure over the set of genome sequences. A *Mosaic* M over a SNP data matrix D is defined as a set of recombination breakpoints $M = \{(h_i, s_b)\}$, $i \in [1, n], b \in [1, m]$. The set of distinct[1] locations of breakpoints s_b in M divides the entire range of SNPs $[s_1, s_m]$ into blocks that satisfy: 1) each block is a compatible block, 2) any two neighboring blocks are incompatible, and 3) any two neighboring blocks (assume the boundary is between s_b and s_{b+1}) would become compatible if the set of haplotypes that have breakpoints between s_b and s_{b+1} are excluded. I develop an efficient algorithm for computing *Minimum Mosaic* (denoted as M_{min}) – the mosaic structure that contains the least number of breakpoints. This problem is referred to as the *Minimum Mosaic Problem*.

[1]Multiple breakpoints can correspond to the same s_b

3.4 Inferring the Local Mosaic

3.4.1 Maximal Intervals

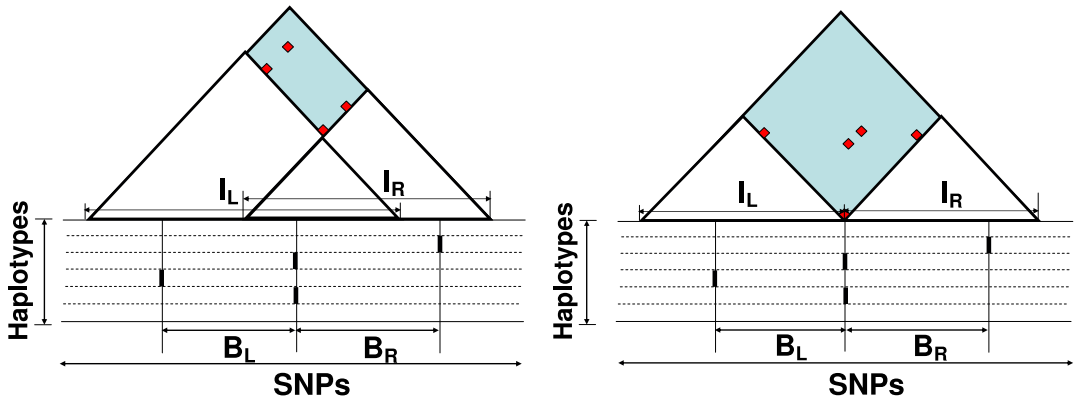
An *interval* $I = [s_b, s_e]$ is a set of consecutive SNPs which are compatible with each other starting from s_b and ending at s_e. An interval I is a *maximal interval* if and only if there is no other interval I', $I' \neq I$, $I' = [s'_b, s'_e]$, which contains I: $s'_b \leq s_b$, and $s'_e \geq s_e$. The complete set of maximal intervals can be computed in $O(mn)$ time (Moore et al. (2008)), assuming that the compatibility test of any two SNPs using FGT takes $O(1)$ time (Moore et al. (2008)).

3.4.2 Finding Local Breakpoints

Maximal intervals are useful for inferring the local mosaic structure. The set of distinct breakpoint locations s_b in a mosaic $M = \{(h_i, s_b)\}$ divide the entire SNP range $[s_1, s_m]$ into compatible blocks, where neighboring blocks are incompatible. The set of breakpoints in M is the union of the set of breakpoints on the boundary of each pair of neighboring blocks. First consider the breakpoints on the boundary of a pair of neighboring blocks. It can be observed that, every pair of neighboring blocks in M fall inside a pair of overlapping or adjacent maximal intervals, as stated in the following Lemma:

Lemma 3.4.1. *For any pair of neighboring blocks (B_L, B_R) deduced by a mosaic, there exists a pair of overlapping or adjacent maximal intervals (I_L, I_R), where B_L completely falls inside I_L ($B_L \subseteq I_L$) but not I_R ($B_L \backslash I_R \neq \phi$), and B_R completely falls inside I_R ($B_R \subseteq I_R$) but not I_L ($B_R \backslash I_L \neq \phi$). (I_L, I_R) is referred to as (B_L, B_R)'s* ***containing interval pair****; and (B_L, B_R) as (I_L, I_R)'s* ***contained block pair*** *(Fig. 1).*

Proof. Firstly, B_L and B_R cannot be contained in the same maximal interval. Assume there exists a maximal interval which contains both of them, then they must be compatible with each other. If they are compatible with each other, we can remove the



(a) I_L *and* I_R *are overlapping* (b) I_L *and* I_R *are adjacent*

Figure 3.1: Neighboring blocks B_L, B_R fall inside overlapping/adjacent maximal intervals I_L, I_R respectively. The dots in the shaded region represent incompatible SNP pairs of I_L and I_R.

breakpoint(s) defining the boundary of B_L and B_R from the mosaic, and the merged block $B_L \bigcup B_R$ is still a compatible block, therefore there exists a smaller mosaic, which contradicts the fact that the original mosaic is not redundant. Since both B_L and B_R are compatible blocks, there must be maximal intervals I_L, I_R which completely contain B_L, B_R respectively, and we know that $I_L \neq I_R$. Since B_L and B_R are neighboring blocks, I_L and I_R are either overlapping (Fig. 1(a)) or touching each other (Fig. 1(b)). Since both B_L and B_R are two compatible blocks which are incompatible with each other, there exist two maximal intervals I_L, I_R which completely contain B_L, B_R respectively. Since B_L and B_R are neighboring blocks, I_L and I_R can either overlap (Fig. 1(a)) or be adjacent to each other (Fig. 1(b)).

Since both B_L and B_R are compatible blocks, there exist two maximal intervals I_L, I_R which completely contain B_L, B_R respectively. We know that B_L and B_R are incompatible. Therefore, they cannot be contained in the same maximal interval: $I_L \neq I_R$, $B_L \backslash I_R \neq \phi$, $B_R \backslash I_L \neq \phi$. Since B_L and B_R are neighboring blocks, I_L and I_R can either overlap (Fig. 1(a)) or be adjacent to each other (Fig. 1(b)).

There might be multiple containing interval pairs for (B_L, B_R), and there might be multiple contained block pair for (I_L, I_R), too. □

For each pair of overlapping or adjacent intervals (I_L, I_R), there exists a set of incompatible SNP pairs $SNPPair(I_L, I_R) = \{(s_l, s_r)\}$, where $l < r$, s_l and s_r are incompatible, and $s_l \in I_L \backslash I_R$, $s_r \in I_R \backslash I_L$. For example, in Fig. 1(a) and 1(b), each dot represents an incompatible SNP pair. Let (B_L, B_R) be a contained block pair of the interval pair (I_L, I_R). The incompatible SNP pairs contained in (B_L, B_R) are denoted as $SNPPair(B_L, B_R) = \{(s_l, s_r)\}$, where $l < r$, s_l and s_r are incompatible, and $s_l, s_r \in B_L \cup B_R$. Apparently, $SNPPair(B_L, B_R)$ is a subset of $SNPPair(I_L, I_R)$. The incompatible SNP pairs in $SNPPair(B_L, B_R)$ determine the minimum number of the breakpoints on the boundary of B_L and B_R, as well as the corresponding set of haplotypes having these breakpoints. Given an interval pair, several candidate block pairs may be derived, each of which corresponds to a different $SNPPair(B_L, B_R)$. Fig. 2(a)-2(d) show four different candidate block pairs derived from the same interval pair. The exact set of incompatible SNP pairs in $SNPPair(B_L, B_R)$ depends on the positions of B_L and B_R, *i.e.*, the leftmost SNP of B_L, and the rightmost SNP of B_R. Formally, the start range R_s, the end range R_e of a block pair (B_L, B_R) are defined as the ranges where $SNPPair(B_L, B_R)$ remains unchanged if the leftmost SNP of B_L changes within R_s and the rightmost SNP of B_R changes within R_e. Moreover, the breakpoint range R_b of (B_L, B_R) is defined as the range where the boundary of B_L and B_R falls into. The breakpoint range is the overlapping region of I_L and I_R (if I_L and I_R overlap), or the boundary of I_L and I_R (if I_L and I_R are adjacent to each other). For example, in Fig. 2(a), $SNPPair(B_L, B_R)$ contains only one incompatible SNP pair (s_q, s_r). The start range $R_s(B_L, B_R)$ is $(s_p, s_q]$, the end range $R_e(B_L, B_R)$ is $[s_r, s_s)$, and the breakpoint range $R_b(B_L, B_R)$ is $I_L \cap I_R$. In Fig. 2(b), $SNPPair(B_L, B_R)$ contains two incompatible SNP pairs (s_q, s_r) and (s_p, s_r). The start range $R_s(B_L, B_R)$ is $[s^*, s_p]$ (s^* denotes the leftmost SNP of interval I_L), and the end range $R_e(B_L, B_R)$ is $[s_r, s_s)$, and the breakpoint range $R_b(B_L, B_R)$ is $I_L \cap I_R$.

Any contained block pair (B_L, B_R) of any overlapping/adjacent maximal interval

pair (I_L, I_R) can be a possible neighboring block pair inside a mosaic M. A subset of these block pairs constitute a mosaic. Specifically, for any neighboring block pair (B_L, B_R) which is inside a Minimum Mosaic M_{min}, we have the following Lemma:

Lemma 3.4.2. *Let (B_L, B_R) be a neighboring block pair in a Minimum Mosaic M_{min}, and $Breakpoints(B_L, B_R)$ be the set of breakpoints on the boundary of B_L and B_R in M_{min}, and $HapSet(Breakpoints(B_L, B_R))$ be the set of haplotypes having breakpoints in $Breakpoints(B_L, B_R)$. Then $Breakpoints(B_L, B_R)$ is the smallest number of breakpoints which satisfies:*

$$\begin{aligned} &\forall (s_l, s_r) \in SNPPair(B_L, B_R) \\ &(\exists g_{l,r} \in \{00, 01, 10, 11\}, HapSet(Breakpoints(B_L, B_R)) \supseteq HapSet^{l,r}_{g_{l,r}}) \end{aligned} \tag{3.1}$$

Proof. Equation 5.3.2 requires $HapSet(Breakpoints(B_L, B_R))$ to be a superset of at least one of $HapSet^{l,r}_{00}$, $HapSet^{l,r}_{01}$,$HapSet^{l,r}_{10}$, $HapSet^{l,r}_{11}$ for any incompatible SNP pair (s_l, s_r) in $SNPPair(B_L, B_R)$. This guarantees that all incompatible SNP pairs in $SNPPair(B_L, B_R)$ would have been compatible if the haplotypes in $HapSet(Breakpoints(B_L, B_R))$ are excluded. According to the Mosaic definition, Equation 5.3.2 holds for any neighboring block pair (B_L, B_R) in any mosaic. Moreover, if (B_L, B_R) is a neighboring block pair in a Minimum Mosaic, $Breakpoints(B_L, B_R)$ is the set of smallest number of breakpoints satisfying Equation 5.3.2. □

It is easy to compute $Breakpoints(B_L, B_R)$ if $SNPPair(B_L, B_R)$ only contains one pair of incompatible SNPs (as shown in Fig. 2(a)). The smallest set of $HapSet^{l,r}_{00}$, $HapSet^{l,r}_{01}$, $HapSet^{l,r}_{10}$ and $HapSet^{l,r}_{11}$ can be chosen to be $Breakpoints(B_L, B_R)$. If $SNPPair(B_L, B_R)$ contains more than one pair of incompatible SNPs (as shown in Fig. 2(b), 2(c), and 2(d)), the smallest set of haplotypes needs to be found which is a superset of at least one of $HapSet^{l,r}_{00}$, $HapSet^{l,r}_{01}$, $HapSet^{l,r}_{10}$ and $HapSet^{l,r}_{11}$, for each pair of incompatible SNPs (s_l, s_r). The computation complexity is $O(4^k)$, where

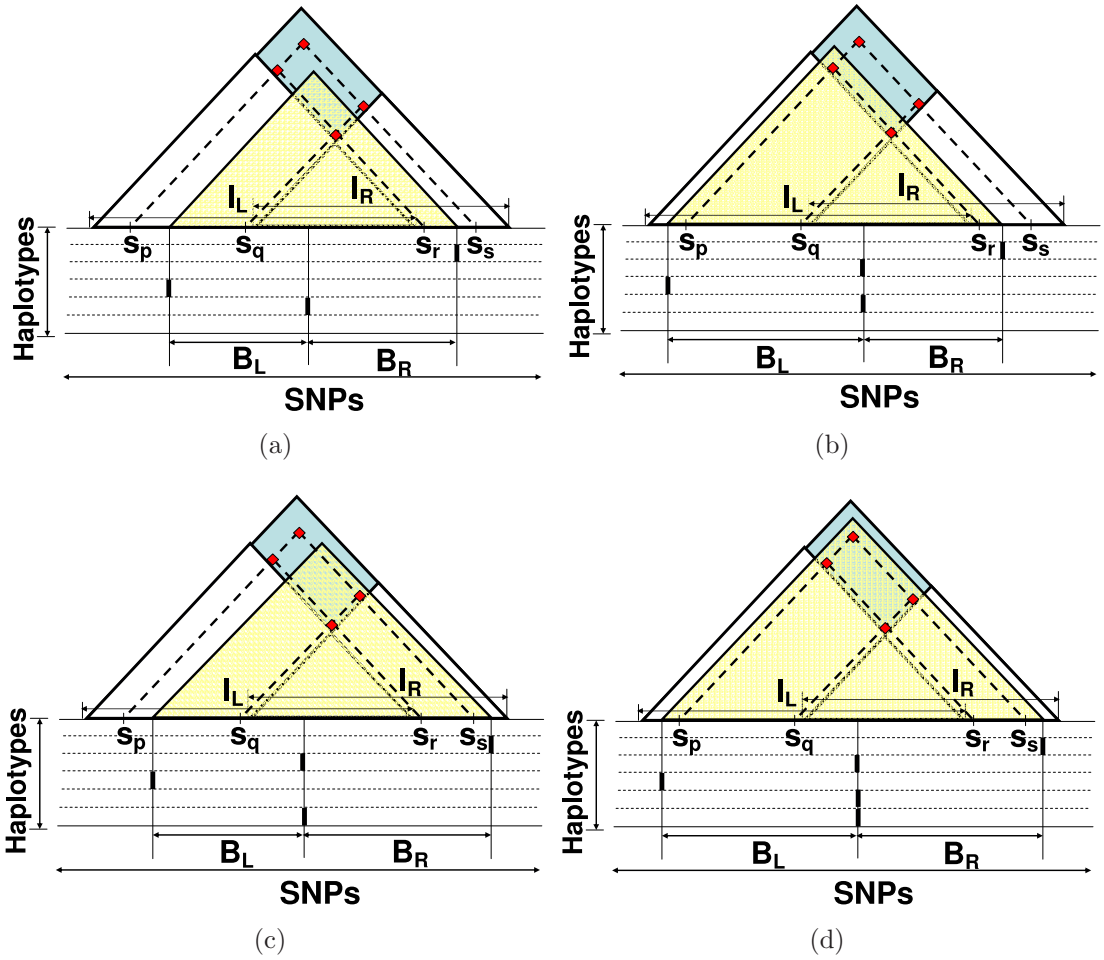


Figure 3.2: Neighboring blocks B_L, B_R contain different subsets of the incompatible SNP pairs. The dots represent the incompatible SNP pairs contained in the overlapping maximal intervals I_L and I_R. The dots inside the shaded triangle are contained in the neighboring block pair B_L and B_R.

$k = |Breakpoints(B_L, B_R)|$. In practice, k is small. Moreover, many incompatible SNP pairs are caused by a small number of SNP patterns, which enables further reduction in computation.

3.5 Finding Minimum Mosaic - A Graph Problem

The set of all possible block pairs $\{(B_L, B_R)\}$ of all overlapping/adjacent maximal interval pairs are the building blocks of a mosaic. These block pairs can be used to

construct a graph as follows. A node nd in this graph represents the combination of three block pairs $BP_1 = (B_{L_1}, B_{R_1})$, $BP_2 = (B_{L_2}, B_{R_2})$, $BP_3 = (B_{L_3}, B_{R_3})$ that satisfy the following constraints: 1) the breakpoint range of BP_1 overlaps with the start range of BP_2: $R_b(BP_1) \cap R_s(BP_2) \neq \phi$; 2) the end range of BP_1, the breakpoint range of BP_2, and the start range of BP_3 overlap: $R_e(BP_1) \cap R_b(BP_2) \cap R_s(BP_3) \neq \phi$; 3) the end range of BP_2 overlaps with the breakpoint range of BP_3: $R_e(BP_2) \cap R_b(BP_3) \neq \phi$. As shown in Fig. 3, BP_1, BP_2, and BP_3 are the left block pair, middle block pair, and right block pair of nd, respectively. The breakpoint range of nd is the intersection of the end range of BP_1, the breakpoint range of BP_2, and the start range of BP_3: $R_b(nd) = R_e(BP_1) \cap R_b(BP_2) \cap R_s(BP_3)$. The set of breakpoints associated with nd is the same as $Breakpoints(BP_2)$, denoted as $Breakpoints(nd)$. The weight of the node is the number of breakpoints in $Breakpoints(nd)$, $weight(nd) = |Breakpoints(nd)|$.

Two special kinds of nodes – starting nodes and ending nodes are also created to model the two ends of a chromosome. All block pairs with start range beginning from the first SNP s_1 are identified which are referred to as starting block pairs. A starting node nd_s is created for every combination of a starting block pair BP_s and another block pair BP satisfying 1) the breakpoint range of BP_s overlaps with the start range of BP: $R_b(BP_s) \cap R_s(BP) \neq \phi$, and 2) the end range of BP_s overlaps with the breakpoint range of BP: $R_e(BP_s) \cap R_b(BP) \neq \phi$. BP_s is the middle block pair of the starting node nd_s, BP is the right block pair of nd_s. There is no left block pair for nd_s. The set of breakpoints associated with nd_s is the same as $Breakpoints(BP_s)$: $Breakpoints(nd_s) = Breakpoints(BP_s)$. The weight of nd_s is $weight(nd_s) = |Breakpoints(nd_s)|$. Similarly, a set of ending nodes $\{nd_e\}$ are computed associated with the set of ending block pairs $\{BP_e\}$.

After all nodes are generated, they are connected with directed edges according to the following rule. For nodes nd_1 and nd_2, if nd_1's middle block pair is the same as nd_2's left block pair and nd_1's right block pair is the same as nd_2's middle block pair, an edge

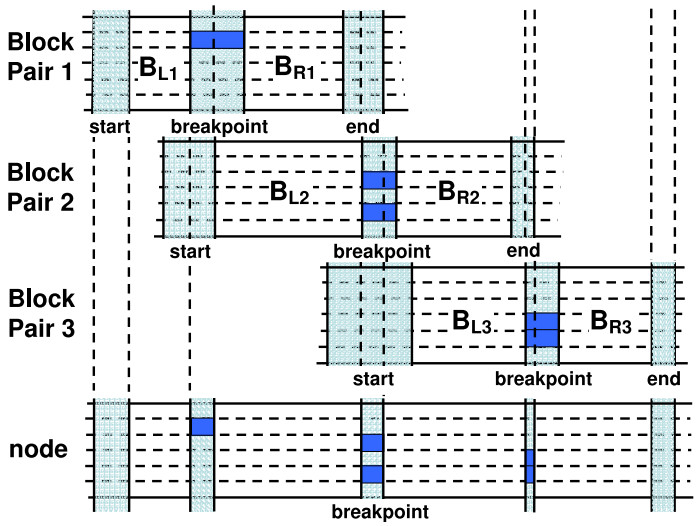


Figure 3.3: Three block pairs form a node. Block pair 1, 2, and 3 are the left, middle, and right block pair of the node respectively. The breakpoint range of the node is the intersection of the end range of block pair 1, the breakpoint range of block pair 2, and the start range of block pair 3. The vertical stripes correspond to the start range, breakpoint range, and end range of a block. The marked haplotypes in the stripes are the haplotypes which have breakpoints in the corresponding region.

is added from nd_1 to nd_2. The nodes and edges form a directed graph. A Minimum Mosaic corresponds to a shortest path from any starting node to any ending node in this graph. The weight of the path is the sum of all node weights on the path. The set of breakpoints $\{Breakpoints(nd)\}$ associated with all nodes on the shortest path is the Minimum Mosaic solution. Any shortest path algorithm can be applied to compute the solution. The details of the complete algorithm is described in Algorithm

Theorem 3.5.1. *M_{min} computed by Algorithm 1 is a Minimum Mosaic.*

Proof. According to Lemma 4.1, Steps 1-7 generate all possible neighboring block pairs which can appear in a mosaic. According to Lemma 4.2, Step 5 computes the number of breakpoints which can be between a block pair *bp*, if *bp* appears in a Minimum Mosaic. The shortest path algorithm guarantees the minimum total number of breakpoints of sequences of neighboring block pairs from a starting block pair to an ending block pair. □

Algorithm 3.1 CompMinMosaic
Input $D_{n\times m}$: The SNP data matrix of n haplotypes and m SNPs
Output M_{min}: a Minimum Mosaic of D

Compute the set of maximal intervals I_{Set}
Initialize the set of neighboring block pairs $B_{Set} \leftarrow \phi$
for any pair of overlapping/adjacent maximal intervals $I_L, I_R \in I_{Set}$ **do**
  Compute the set of contained block pairs $ContainedBP(I_L, I_R)$;
  For each block pair $bp \in ContainedBP(I_L, I_R)$, compute $Breakpoints(bp)$ according to Lemma 4.2.
  $B_{Set} \leftarrow B_{Set} \cup ContainedBP(I_L, I_R)$
end for
Create nodes using the block pairs in B_{Set}, add edges between nodes, generate graph G
Compute the shortest path in G, set M_{min} to be the union of the set of breakpoints associated with the nodes on the shortest path.

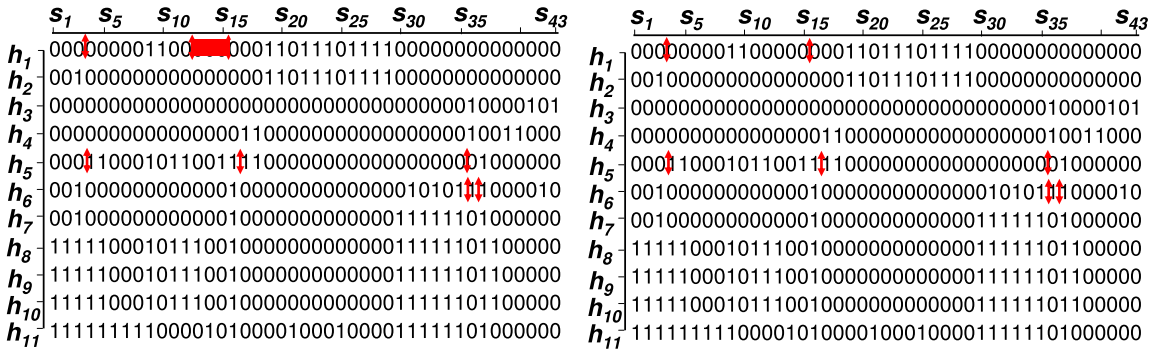
3.6 Experimental Studies

The algorithm is implemented in C++ and all experiments were performed on a machine with an Intel Core2 Duo processor of 1.60GHz and 3GB RAM.

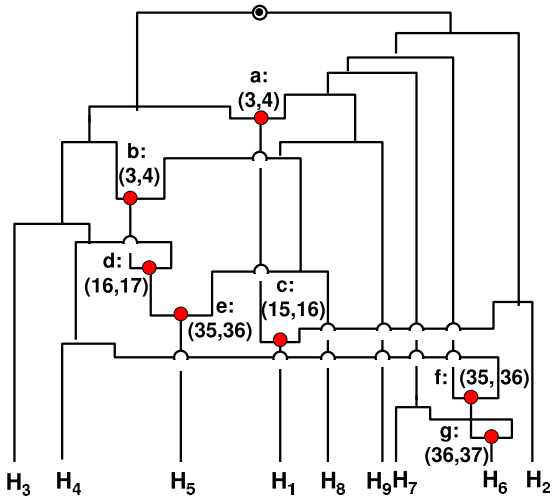
3.6.1 Kreitman's ADH Data

The alcohol dehydrogenase (ADH) data of Kreitman (Kreitman (1983)) consists of 11 haplotypes with over 43 polymorphic sites of the ADH locus of fruit fly, *Drosophila melanogaster*. The haplotypes were sampled from 5 geographically distinct populations: Washington, Florida, Africa, France, and Japan (Song and Hein (2005)). The minimum mosaic solution detected 7 breakpoints shown in Fig. 4(a). The exact locations of 6 out of 7 breakpoints can be estimated: $H_1 : (S_3, S_4)$, $H_5 : (S_3, S_4)$, $H_5 : (S_{16}, S_{17})$, $H_5 : (S_{35}, S_{36})$, $H_6 : (S_{35}, S_{36})$, $H_6 : (S_{36}, S_{37})$. For the remaining breakpoint on H_1, its location can be either (S_{12}, S_{13}), or (S_{13}, S_{14}), or (S_{14}, S_{15}), or (S_{15}, S_{16}) with equal probability.

Note that 7 is the lower and upper bounds of the minimum number of recombinations, estimated by HapBound and SHRUB, respectively (Song et al. (2005)). Therefore, 7 is



(a) (b)



(c)

Figure 3.4: Comparison of Minimum Mosaic and Hapbound/SHRUB results on ADH data. (a): the Minimum Mosaic result; (b): the result inferred from the ARG in (c); (c): the ARG computed using SHRUB(Song and Hein (2005)). The bars in (a) and (b) represent the breakpoints. The dots in (c) represents the recombination events.

the exact number of minimum number of recombination events for the ADH data. The corresponding ARG generated by SHRUB is shown in Fig. 4(c). The breakpoints in the ARG are illustrated in a SNP matrix in Fig. 4(b). By comparing Fig. 4(a) and 4(b), it can be observed that almost the same set of breakpoints are inferred by the proposed algorithm and SHRUB.

3.6.2 Running Time and Scalability Analysis

The performance of the proposed algorithm is tested on two genome-wide SNP datasets from mouse. Both sets represent a combination of experimental and imputed genotypes (Szatkiewicz et al. (2008)) in two overlapping sets of laboratory inbred strains available from the Center of Genome Dynamics at the Jackson's Laboratory [2] The 51-strain dataset contains 51 inbred mouse strains with 7,870,134 SNPs[3]. The 74-strain dataset contains 74 inbred mouse strains with 7,989,200 SNPs[4].

Fig. 5 shows the running time comparison of Hapbound and the proposed algorithm using the first w SNPs from Chromosome 19 of both datasets where w varies from 1000 to 4000. The proposed algorithm is 250x - 7000x faster than Hapbound on 74-strain dataset, and 350x - 4000x faster on 51-strain dataset.

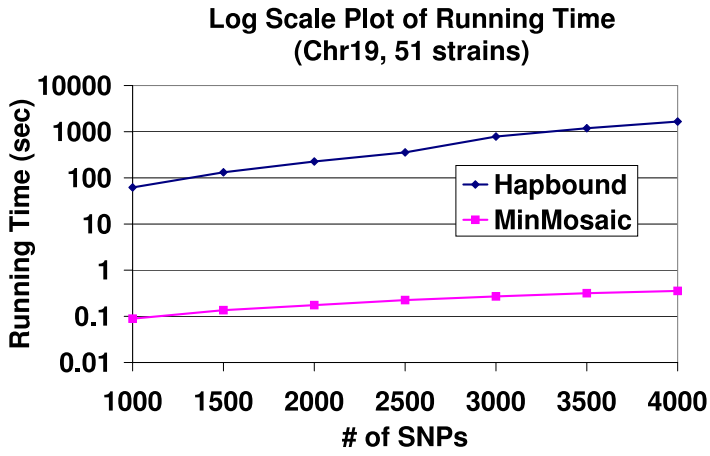
The proposed algorithm is efficient enough to finish on all chromosomes (Chr 1-19 and Chr X). Results from the 51-strain dataset are shown in Table 1. Genome-wide, the number of breakpoints in the Minimum Mosaic varies between 15253 (Chr X) and 266006 (Chr 1), and the number of derived blocks in the Minimum Mosaic varies between 9888 (Chr X) and 68261 (Chr 1). The average number of breakpoints per neighboring block pair is 2.2.

---

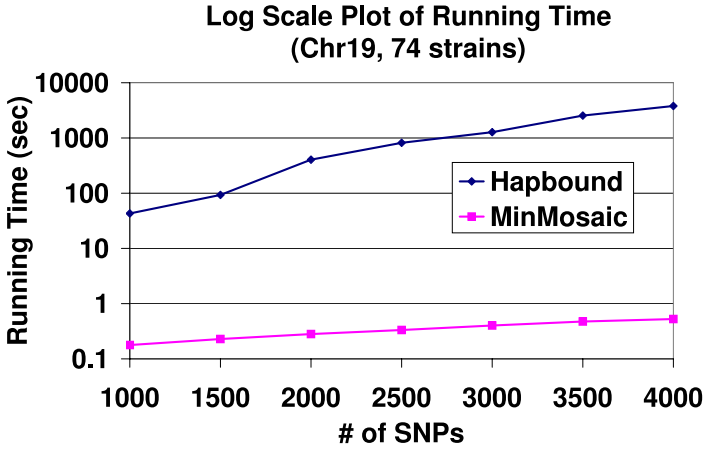
[2] `http://cgd.jax.org/ImputedSNPData/imputedSNPs.htm`

[3] The 51-strain dataset includes Chr 1-19 and Chr X, with 51 mouse strains: X129S1.SvImJ, X129S4.SvJae, X129X1.SvJ A.J, AKR.J, BALB.cByJ, BTBR.T....tf.J, BUB.BnJ, C3H.HeJ, C57BL.6J, C57BLKS.J, C57BR.cdJ, C57L.J, C58.J, CAST.EiJ, CBA.J, CE.J, CZECHII.EiJ, DBA.1J, DBA.2J, DDK.Pas, FVB.NJ, I.LnJ, JF1.Ms, KK.HLJ, LG.J, LP.J, MA.MyJ, MAI.Pas, MOLF.EiJ, MSM.Ms, NOD.LtJ, NON.LtJ, NZB.BlNJ, NZO.HlLtJ, NZW.LacJ, O20, PERA.EiJ, PL.J, PWD.Ph, PWK.PhJ, Qsi5, RIIIS.J, SEA.GnJ, SEG.Pas, SJL.J, SM.J, SPRET.EiJ, ST.bJ, SWR.J, and WSB.EiJ.

[4] The 74-strain dataset includes all strains in the 51-strain dataset and 23 additional strains: BALB.cJ, BPH.2J, BPL.1J, BPN.3J, C57BL.10J, CALB.RK, DDY.JCLSIDSEYFRKJ, EL.SUZ_2, HTG.GOSFSN, ILS, IS.CAMRK, ISS, LEWES.EI, MOLG.DN, MRL.MpJ, NOR.LTJ, P.J, PERC.EI, RF.J, SKIVE.EI, SOD1.EI, TALLYHO.JNGJ, and ZALENDE.EiJ.



(a) 51-strain Dataset



(b) 74-strain Dataset

Figure 3.5: Comparison of the running times of MinMosaic and Hapbound over varying number of SNPs (in log scale). The datasets used are from Chr19 of 51-strain dataset and 74-strain dataset. The number of SNPs included varies from 1000 to 4000.

Chr	# of SNP	# of breakpoints	# of blocks	Runtime (min)
1	694809	266006	68261	6.87
2	524667	210797	47793	11.27
3	509892	113715	52487	8.90
4	476425	100702	43776	7.84
5	496888	110157	49938	33.98
6	509547	97740	49562	6.42
7	405733	94973	46884	38.83
8	444910	87659	45796	37.10
9	361571	86755	40189	3.89
10	399126	64806	35764	3.21
11	259028	65092	27575	23.52
12	396114	89243	42159	1.30
13	399930	75323	39914	3.03
14	345783	67304	34089	2.54
15	337461	78181	35776	4.08
16	305078	57257	28449	1.14
17	266421	73542	31517	0.75
18	291266	69546	31271	8.61
19	222031	49276	22839	1.46
X	223456	15253	9888	0.96

Table 3.1: The result on genome-wide 51-strain mouse dataset

Chapter 4

Clustering Distributed Data Streams

4.1 Introduction

Distributed data stream applications perform continuous, on-the-fly computations over geographically dispersed data streams, such as environmental and ecological sensor network systems monitoring flood and volcano eruption, or monitoring habitat and wild populations. Various tasks of continuous query and monitoring in distributed setting have been developed, including database queries (Cherniack et al. (2003); Ahmad and etintemel (2004); Olston et al. (2003); Cormode et al. (2005)), monitoring simple statistics (Babcock and Olston (2003); Keralapura et al. (2006); Sharfman et al. (2006)), event detection (Aggarwal (2005); Aggarwal et al. (2003); Zhu and Shasha (2003)), etc. The problem being addressed in this work is clustering over distributed data streams, particularly, the continuous k-median clustering in a distributed stream environment.

Consider a set of distributed sites, which push the data towards the base site periodically. Each such period is referred to as an *update epoch*. Assume that all the sites communicate according to a routing tree rooted at the base site. The goal is to perform clustering on the data collected from all the sites. The clustering result is continuously updated at the root for each update epoch. Assume that the epoch is long enough so that all the data of one epoch is able to arrive at the root before next epoch ends.

Clustering over distributed streams is a challenging task. Difficulties lie in various issues: 1) *Communication*. Distributed stream system continuously produces large volume of data, which imposes prohibitive communication load if all data are transferred to the root for centralized computation. In-network aggregation (Madden et al. (2002)) is one of the techniques (Olston et al. (2003); Madden et al. (2002); Silberstein et al. (2006); Willett et al. (2004)) that push processing operators down into the network to reduce data transmission. It computes a local summary at each site and merges and summarizes further at each internal site towards the root. However, this approach cannot be immediately adopted to solve the problem of k-median clustering. The k-median clustering is known as a holistic computation (Madden et al. (2002)), which cannot be readily decomposed into computations on data partitions. More specifically, the k-median clustering of the entire dataset cannot be accurately computed from the k-median centers of individual partitions. In order to get the exact answer, all data need to be transmitted to a central site before the k-median clustering is performed. 2) *Latency*. Another approach to solve the k-median problem is to treat it as an iterative optimization problem. For each iteration, statistics (Forman and Zhang (2001)) need to be computed and transmitted back and forth between the root and each site. The induced latency is unbearable for stream-based applications. 3) *Clustering quality*. Clustering quality is another concern when trading accuracy for reduced communication. In this case, it is necessary that the error of the k-median solution is bounded. The bounded approximate in-network aggregation also raises new issues such as error propagation and topology sensitivity.

The approximate in-network aggregation schemes are employed for $(1+\varepsilon)$-approximate k-median clustering over distributed data streams. A suite of algorithms for both topology-insensitive and network-aware cases are proposed, with the following major contributions:

- For topology-insensitive case, a multi-level structure of ε-kernel of k-median (ε-

coreset (Har-Peled and Mazumdar (2004))) is constructed as the local summary. The size of the multi-level summary is proved to be a *polylog* function of the data volume and it is shown to guarantee a $(1+\varepsilon)$-approximate clustering. Efficient algorithms are proposed for constructing and merging local summaries.

- For network-aware cases, the height-aware and path-aware algorithms are proposed which utilize the topology information to achieve aggressive reduction of data communication.

- Experiments are performed on both synthetic and real data sets to demonstrate the performance of the algorithms in terms of communication reduction, clustering quality and scalability.

4.2 Related Work

Little work has been reported for distributed stream clustering. In this section, a brief review is given on distributed clustering, approximate in-network aggregation, and stream clustering. In addition, an introduction is given on coreset, which is the ε-kernel for k-median clustering.

4.2.1 Distributed Clustering

Distributed clustering needs to address the problem of balancing between communication and precision. Forman and Zhang (Forman and Zhang (2001)) proposed a technique to exactly compute several iterative center-based data clustering algorithms including k-means for distributed database applications. It sends sufficient statistics instead of the raw data to a central site. However, this approach involves each site in every iteration of k-means and transfers data back and forth, which is infeasible for stream applications. Januzaj et al. (Januzaj et al. (2003)) proposed a distributed density-based

clustering algorithm. The algorithm clusters the data locally at each site and computes the aggregation (representatives for local clusters) for the local site. All aggregations are sent to the global server site where the global clustering is carried out. Their algorithm considers the flat two-tier topology instead of tree topology, and it does not provide a bound of the clustering quality.

4.2.2 Approximate In-network Aggregation

Bounded-error approximation is a desired property to have when trading accuracy for communication requirement. Cosdine et al. (Considine et al. (2004)) proposed an approximate in-network aggregation scheme for sensor databases. They provided an algorithm for approximate duplicate-sensitive aggregates across distributed datasets, such as SUM. Their algorithm employs small duplicate-insensitive sketches for SUM which is generalized from similar technique for approximating COUNT. Greenwald et al. (Greenwald and Khanna (2004)) proposed an algorithm for power-preserving computation of order statistics such as quantile. They proposed a scheme of computing ε-approximate quantile over the sensor network routing tree, where each sensor transmits only $O(\log^2 n/\varepsilon)$ data points opposed to the worst case of $\Omega(n)$.

4.2.3 Clustering Single Stream

A lot of work has been reported for clustering over a single stream (Aggarwal et al. (2003, 2004); O'Callaghan et al. (2002); Har-Peled and Mazumdar (2004); Guha et al. (2000)). They developed continuous online clustering algorithms with small space-requirement. Guha et al. (Guha et al. (2000)) proposed a constant factor approximation algorithms for k-Median clustering on a single data stream. Their algorithm requires $O(n^{\varepsilon})$ space with an approximation factor of $2^{O(\frac{1}{\varepsilon})}$. Aggarwal et al. (Aggarwal et al. (2003)) proposed an algorithm which considers both online statistical data collection, and offline analysis, compared with the one-pass clustering algorithms.

4.2.4 Coreset and Streaming k-median

A coreset is a small subset of points that approximates the original set with respect to some tasks (such as k-median, k-means, etc) (Har-Peled and Kushal (2005)). Several coreset construction algorithms have been proposed for k-median and k-means clustering (Har-Peled and Kushal (2005); Har-Peled and Mazumdar (2004); Frahling and Sohler (2005)). In (Frahling and Sohler (2005)), a coreset construction algorithm for streaming $(1+\varepsilon)$-approximate k-median and k-means is proposed. The coreset is computed using a QuadTree, and its space requirement is *polylog* in size of the data stream and the range of the data values. Har-Peled and Mazumdar (Har-Peled and Mazumdar (2004)) proposed another coreset construction algorithm for $(1+\varepsilon)$-approximate k-median and k-means, where the coreset takes *polylog* space. These two coreset-based streaming k-median algorithms achieve a better space-bound than Guha's algorithm (Guha et al. (2000)) mentioned earlier. All of these algorithms consider the clustering over a single stream, not the stream clustering problem in a distributed setting.

4.3 Preliminaries and Background

4.3.1 Problem Definition

Let $d(p,c)$ denote the Euclidean distance between any two points p and c in R^d. The goal of k-median clustering is to find a set C of k points representing the k cluster centers in R^d, which minimize the k-median cost of dataset P. Here the k-median cost is defined as $Cost(P,C) = \sum_{p\in P} d(p,C)$, where $d(p,C) = \min_{c\in C} d(p,c)$. If P is a weighted dataset with $p_i \in P$ of weight w_i, then the weighted k-median of P is defined as the cluster center set C which minimizes $Cost(P,C) = \sum_{p_i\in P} w_i d(p_i,C)$.

Definition 4.3.1. $(1+\varepsilon)$-APPROXIMATE K-MEDIAN OF DATASET P. *Let C denote the k-median center set. Then C' is defined as a $(1+\varepsilon)$-approximate k-median center set if*

$Cost(P, C') \leq (1+\varepsilon)Cost(P, C)$.

Consider a set of sites $\{S_i\}$ which communicate according to a routing tree[1]. Suppose that during each update epoch e, site S_i receives a stream of data $P_i|_e$. There are two problems to be considered: (1) how to compute the $(1+\varepsilon)$-approximate k-median on the set of data $\bigcup_i (P_i|_{e^*})$, where e^* is the latest update epoch; and (2) how to compute the $(1+\varepsilon)$-approximate k-median on all data received in past epochs. In fact, the solutions to both problems are similar except that the root may perform some additional operations to solve the second problem. Therefore, only the first problem will be discussed and the proposed algorithms can be modified to solve the second problem.

The first problem is defined as below:

Definition 4.3.2. $(1+\varepsilon)$-APPROXIMATE K-MEDIAN OVER DISTRIBUTED STREAM SET $\{S_i\}$ DURING EPOCH e. *The $(1+\varepsilon)$-approximate clustering of distributed stream set $\{S_i\}$ during epoch e is defined as the $(1+\varepsilon)$-approximate k-median over all the data received at all the sites during epoch e, which is $\bigcup_i (P_i|_e)$.*

In the following discussion, P_i or P will be used to represent streams, assuming that the streams considered are during one epoch.

4.3.2 Local Summary Structure

The algorithms employ approximate in-network aggregation schemes. Basically, each site computes a local summary and sends it to its parent. For internal nodes, the local summary is merged with the summaries received from children and another local summary operation is performed on the merged set before the summary is sent to the parent. When all summaries reach the root, a k-median clustering procedure is performed at the root to get the $(1+\varepsilon)$-approximate clustering of the whole data.

[1]Each site is a node in the routing tree. In the following discussion, the terms *site* and *node* are used interchangeably.

An important element of the algorithm is the summary structure. A desirable summary structure SM should satisfy the following properties:

- **Property I: distributive (or decomposable):** The summary of a set P can be computed from the summaries of its partitions. For example, $SM(P) = f(SM(P_1), SM(P_2))$, where $P = P_1 \bigcup P_2$ and f is the function to combine the partition summaries.

- **Property II: compact:** the summary should have a much smaller size than the original data.

- **Property III: error bound:** the summary should deliver a k-median solution with bounded error. A k-median solution on the summary should be a $(1 + \varepsilon)$-approximate k-median of the original data. Here ε is the approximation factor associated with the summary.

- **Property IV: error-accumulation:** the summary of the summary of dataset P should still be a summary of P, only with a looser approximation factor due to error accumulation. Formally, $SM_{\varepsilon_1}(SM_{\varepsilon_2}(P)) = SM_{g(\varepsilon_1,\varepsilon_2)}(P)$, where $g(\varepsilon_1, \varepsilon_2)$ is a function to accumulate the error which satisfies $g(\varepsilon_1, \varepsilon_2) \geq \varepsilon_1$ and $g(\varepsilon_1, \varepsilon_2) \geq \varepsilon_2$.

The summary structure and the algorithms for constructing and merging this type of summaries are presented in Section 4.

4.3.3 Topology Dependence

The amount of accumulated error of in-network aggregation is determined by the number of merges, compressions and propagations of the local summaries occurring at each node in the network. Therefore, the approximation factor of the summary is highly dependent on the network topology. Three algorithms are proposed that utilize the topology knowledge at different levels.

- **Topology-oblivious algorithm** A topology-oblivious algorithm is proposed which does not assume any prior knowledge of the tree topologies. The algorithm computes a local summary structure of size $O(\frac{k}{\varepsilon^d}\log^3 N\sqrt{\log N})$ which maintains the same error bound at each node. In this topology-oblivious algorithm, only the merging operation performs at the internal node.

- **Height-aware algorithm** Assume that the height of the topology tree is known beforehand. An improved algorithm is proposed which reduces communication compared with the topology-oblivious algorithm.

- **Path-aware algorithm** The height of the subtrees in a tree topology may vary significantly. Given the same approximation bound, the communication load at each node uniformly determined by the height of the entire tree may not be optimal. The path-aware algorithm adaptively computes the communication load for each node according to the height of its subtree. This approach minimizes the communication per node while still ensuring the same overall additive error bound. The algorithm is well-suited for reducing communication for unbalanced tree topologies.

4.4 Algorithm

This section provides the detailed explanation of the algorithms for computing $(1+\varepsilon)$-approximate k-median over the distributed stream considering the different cases of topology dependency. The sensors are considered to be organized into routing trees. Specifically, details will be given on the design of local summary structure at each node, the construction and merging algorithms for summaries, and the error propagation schemes for the network-aware cases. In addition, formal proof will be provided on the error bound for the clustering result and the bound for max per node transmission.

4.4.1 Topology-oblivious Algorithm

This algorithm is designed for the scenario where the tree structure is unknown, *i.e.*, the size and the height of the routing tree is unknown. The algorithm computes $(1+\varepsilon)$-approximate k-median clustering with reduced maximum per-node transmission of $O(\frac{k}{\varepsilon^d}\log^3 N\sqrt{\log N})$.

The algorithm follows the in-network aggregation scheme. The key component is the construction and merging of summary structure.

Local Summary at Each Node

A level-wise structure of k-median's ε-coreset is used as the local summary at each site.

The ε-coreset is an ε-kernel defined on a point set P for certain geometric problems (Agarwal et al. (2005)). For k-median problem, the ε-coreset C is defined as follows:

Definition 4.4.1. ε-CORESET OF DATASET P *Let P be a weighted set of n points in R^d. A weighted point set C is an ε-coreset in R^d for the k-median problem, if for every set Ctr of k centers:*

$$(1-\varepsilon)Cost(P,Ctr) \leq Cost(C,Ctr) \leq (1+\varepsilon)Cost(P,Ctr)$$

Here C is usually a much smaller set than P, and each point $p \in P$ is uniquely represented by a point $c \in C$, where c's weight is defined as the number of points in P it represents.

The ε-coreset is a good candidate for the local summary since it satisfies all requirements of the summary: 1) coreset is distributive. If C_1 and C_2 are the ε-coresets for two disjoint sets of points P_1 and P_2 respectively, then $C_1 \bigcup C_2$ is an ε-coreset for $P_1 \bigcup P_2$. 2) coreset only has size $O(\frac{k}{\varepsilon^d}logN)$, where N is the size of P, and d is the dimensionality of the data point. 3) A $(1+\varepsilon)$-approximate k-median solution of P can be obtained by computing the exact k-median solution on its ε-coreset. 4) The ε_1-coreset of the

ε_2-coreset of P is an $(\varepsilon_1 + \varepsilon_2)$-coreset of P, which means

$$Coreset_{\varepsilon_1}(Coreset_{\varepsilon_1}(P)) = Coreset_{\varepsilon_1+\varepsilon_2}(P)$$

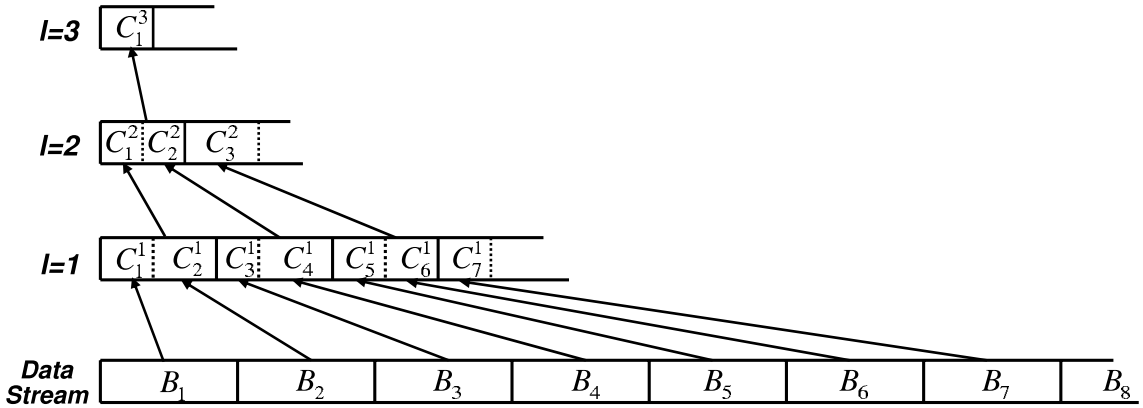


Figure 4.1: EH summary at a site: This figure highlights the multi-level structure of EH-summary. The incoming data is buffered in equi-sized blocks B_1, B_2,. . . ,B_j, . . ., each of size $O(\frac{k}{\varepsilon^d})$. The coreset C_j^1 is computed for each block B_j and sent to level $l = 1$. At each level $l > 0$, whenever two coresets C_j^l, C_{j+1}^l come in, they are merged and another coreset $C_{(j+1)/2}^{l+1}$ on $C_j^l \bigcup C_{j+1}^l$ is computed and sent to level $l + 1$. There are at most $\log \frac{N}{k/\varepsilon^d}$ levels.

EH-Summary

At each site, an EH-summary is computed which is a multi-level structure of coresets over the stream. EH-summary is constructed in the similar fashion to that of building an online exponential histogram. Assume that the stream arriving at a site is P. Whenever a block of B data points comes, a coreset computation is performed and the points in the coreset are appended to the stream one level above. Fig.4.1 illustrates the process of the algorithm. B_j represents the j^{th} block of the original stream data P. Whenever block B_j is filled up and the coreset C_j^1 of B_j is computed, C_j^1 is added to the stream at level $l = 1$. For all levels $l \geq 1$, whenever two coresets C_j^l, C_{j+1}^l come in, they are merged and another coreset $C_{(j+1)/2}^{l+1}$ is computed on top of the merged set and sent to level $l + 1$. This process propagates until at level L, where there is only one coreset. Here block size is set to $B = O(\frac{k}{\varepsilon^d})$. Let P_l denote the stream at level l, P_{Set} denote

the set of streams at all levels. The algorithm is shown as below:

Algorithm 4.1 CompEHSummary(P, ε)
Input P: the original data stream; ε: the required error bound

1: $P_0 \leftarrow P$, $P\_Set \leftarrow \{P_0\}$
2: Arrange P_0 into blocks $B_1, \ldots, B_j, \ldots$ of equal size B
3: $P_1 \leftarrow \Phi$
4: Compute $\Delta\varepsilon_1$-coreset C_j^1 on each block B_j, add C_j^1 into P_1
5: $P\_Set = P\_Set \bigcup \{P_1\}$
6: l=1
7: **while** P_l contains more than one coreset, $P_l = \{C_j^l\}, j = 1, \ldots, J_l, J_l > 1$ **do**
8: **if** $\neg(P_{l+1} \in P\_Set)$ **then**
9: $P_{l+1} \leftarrow \Phi$, $P\_Set = P\_Set \bigcup \{P_{l+1}\}$
10: **end if**
11: Merge C_j^l and C_{j+1}^l, where $j = 1, 3, \ldots$, and $j < J_l$
12: Compute $\Delta\varepsilon_{l+1}$-coreset $C_{(j+1)/2}^{l+1}$ on each $C_j^l \bigcup C_{j+1}^l$, and add $C_{(j+1)/2}^{l+1}$ into P_{l+1}
13: $l \leftarrow l + 1$
14: **end while**

The algorithm generates a set of streams at different levels $P\_Set = \{P_l\}, l = 0, 1, \ldots, L$, where P_0 is the original data stream P. The computation propagates from the lowest level P_0, towards higher levels $P_l, l > 0$ until at level L, there are less than two coresets. At any level l, $\Delta\varepsilon_{l+1} = \frac{1}{(l+1)\sqrt{l+1}}\frac{\varepsilon}{3}$-coresets are computed and sent to level $l + 1$.

Note that in Fig.4.1, a coreset C_j^l at level l represents a sequence of 2^{l-1} consecutive blocks in the original stream $\{B_{(j-1)2^{l-1}+1}, \ldots, B_{j2^{l-1}}\}$, with an approximation factor of $\sum_1^l \Delta\varepsilon_i$. For example, coresset C_1^3 at level 3 covers the blocks $\{B_1, B_2, B_3, B_4\}$, C_3^2 at level 2 covers the blocks $\{B_5, B_6\}$ in P. The union of the original data blocks in P covered by C_j^l is denoted as $CoverInterval(C_j^l)$. The following lemma can be derived:

Lemma 4.4.1. *Each C_j^l is an ε_l-coreset of $CoverInterval(C_j^l)$ where $\varepsilon_l = \varepsilon - \frac{2}{3\sqrt{l}}\varepsilon$.*

According to the distributive and error accumulation properties of coreset, C_j^l is an

ε_l-coreset of $CoverInterval(C_j^l)$, where

$$\begin{aligned}\varepsilon_l &= \sum_1^l \Delta\varepsilon_i \\ &= \sum_1^l \frac{1}{i\sqrt{i}}\frac{\varepsilon}{3} \\ &\leq \frac{\varepsilon}{3} + \frac{\varepsilon}{3}\int_1^l \frac{dx}{x\sqrt{x}} \\ &= \varepsilon - \frac{2}{3\sqrt{l}}\varepsilon.\end{aligned} \tag{4.1}$$

Consider any coreset C_j^l at level l. If $C_{\lceil j/2\rceil}^{l+1} \in P_{l+1}$, we have $CoverInterval(C_j^l) \subset CoverInterval(C_{\lceil j/2\rceil}^{l+1})$, since $C_{\lceil j/2\rceil}^{l+1}$ is a coreset computed on either $C_j^l \bigcup C_{j+1}^l$ or $C_{j-1}^l \bigcup C_j^l$. Formally, we claim that C_j^l is an *obsolete* coreset if $C_{\lceil j/2\rceil}^{l+1} \in P_{l+1}$, and an *active* coreset otherwise. In Fig.4.1, for example, the only three active coresets are C_1^3, C_3^2, and C_7^1. All the remaining coresets are obsolete coresets. Note that there could be at most one active coreset at each level.

Therefore, the EH-summary is defined as follows:

Definition 4.4.2. EH-SUMMARY. *An EH-summary of a stream P is the set of active coresets at all levels (generated by Algorithm 1). $EHSummary(P) = \{\widetilde{C}^0, \widetilde{C}^{l_1}, \ldots, \widetilde{C}^{l_k}, \ldots, \widetilde{C}^{l_K}\}$, where $\widetilde{C}^{l_k}$ is the active coreset at level l_k, $\widetilde{C}^0$ is the newest block in the original stream which is not full, and l_K is the maximum level L generated by Algorithm 1.*

The EH-summary covers the whole data stream P. Each $\widetilde{C}^{l_k}$ in the EH-summary covers a disjoint subset of consecutive complete blocks in P, specifically, $\widetilde{C}^{l_k}$ is an $(\varepsilon - \frac{2}{3\sqrt{l_k}}\varepsilon)$-coreset of $CoverInterval(\widetilde{C}^{l_k})$ (Lemma 4.1). We have

$$CoverInterval(\widetilde{C}^{l_{k_1}}) \bigcap CoverInterval(\widetilde{C}^{l_{k_2}}) = \Phi,$$

and

$$(\bigcup_{l_k} CoverInterval(\widetilde{C}^{l_k}))\bigcup\widetilde{C}^0 = P.$$

In Fig.4.1, for example, the EH-summary is $\{B_8, C_7^1, C_3^2, C_1^3\}$. C_1^3 at level 3 covers the blocks B_1, B_2, B_3, and B_4. C_3^2 covers B_5 and B_6. C_7^1 covers B_7. Together with B_8, they cover the whole data stream.

Lemma 4.4.2. *The union of coresets* $(\bigcup_{l_k}\widetilde{C}^{l_k})\bigcup\widetilde{C}^0$ *in EH-summary* $EH = \{\widetilde{C}^0, \widetilde{C}^{l_1}, ..., \widetilde{C}^{l_k}, ...\widetilde{C}^{l_K}\}$, $l_K = L$, *is an* $(\varepsilon - \frac{2}{3\sqrt{L}}\varepsilon)$*-coreset of* P.

According to Lemma 4.1, each $\widetilde{C}^{l_k}$ is an $(\varepsilon - \frac{2}{3\sqrt{l_k}}\varepsilon)$-coreset of $CoverInterval(\widetilde{C}^{l_k})$. Since $\varepsilon - \frac{2}{3\sqrt{L}}\varepsilon = \max(\varepsilon - \frac{2}{3\sqrt{l_k}}\varepsilon)$, each $\widetilde{C}^{l_k}$ is also an $(\varepsilon - \frac{2}{3\sqrt{L}}\varepsilon)$-coreset of $CoverInterval(\widetilde{C}^{l_k})$. Therefore, according to distributive property, $(\bigcup_{l_k}\widetilde{C}^{l_k})\bigcup\widetilde{C}^0$ is an $(\varepsilon - \frac{2}{3\sqrt{L}}\varepsilon)$-coreset of $CoverInterval(\widetilde{C}^{l_k})\bigcup\widetilde{C}^0 = P$.

Merging EH-Summary at Intermediate Nodes

For any internal node, the local EH-summary needs to be combined with any EH-summary it receives from its children. Consider two EH-summaries $EH = \{\widetilde{C}^0, \widetilde{C}^{l_1}, ..., \widetilde{C}^{l_k}, ..., \widetilde{C}^{l_K}\}$, $EH_* = \{\widetilde{C}_*^0, \widetilde{C}_*^{l_1}, ..., \widetilde{C}_*^{l_k}, ..., \widetilde{C}_*^{l_{K^*}}\}$. Denote the combined EH-summary as EH_{all}. Intuitively, the combination of EH and EH_* can proceed as follows: 1) At level 0, combine $\widetilde{C}^0$ and $\widetilde{C}_*^0$, denote it as $\widetilde{C}_{all}^0$. If $|\widetilde{C}_{all}^0| > B$, arrange $\widetilde{C}_{all}^0$ into blocks B_c and B_r, where B_c is a complete block of size B, and B_r is the remaining part. Compute an $\Delta\varepsilon_1$-coreset $\Delta\widetilde{C}_{all}^1$ over B_c, and send it to level 1. Set $\widetilde{C}_{all}^0$ to be B_r and add it to EH_{all}. 2) For any level $l \geq 1$, start from level 1. a) If both EH and EH_* contain coresets on level l, merge the two coresets, compute another coreset $\Delta\widetilde{C}^{l+1}$ on top of it, and send to level $l+1$. Additionally, if there is a non-empty $\Delta\widetilde{C}^l$ sent by the level $l-1$, add $\Delta\widetilde{C}^l$ into EH_{all} as the coreset at level l. b) If only one of EH and EH_* contains coreset on level l, assume it is EH with $\widetilde{C}^l$. If there is a non-empty $\Delta\widetilde{C}^l$ sent by the lower level, another coreset $\Delta\widetilde{C}^{l+1}$ is computed on top of $\widetilde{C}^l\bigcup\Delta\widetilde{C}^l$ and sent to next

level. Otherwise, $\widetilde{C}^l$ is added into EH_{all} as the coreset at level l. c) Finally, if neither EH nor EH_* contains coreset on level l, and there is a non-empty $\Delta\widetilde{C}^l$ sent by the lower level, $\Delta\widetilde{C}^l$ is added into EH_{all} as the coreset at level l. The algorithm is shown in Algorithm. 2.

Compute k-Median at the Root

At the root node, we have an EH-summary for all the data during the last epoch e, P_e. Let the EH-summary at the root be $EH_{root} = \{\widetilde{C}^0, \widetilde{C}^{l_1}, \ldots, \widetilde{C}^{l_k}, \ldots, \widetilde{C}^{l_K}\}$, where $(\bigcup_l \widetilde{C}^{l_k}) \bigcup \widetilde{C}^0$ is an ε_L-coreset of P_e, and $\varepsilon_L < \varepsilon$, according to Lemma 4.2. Therefore, a $(1+\varepsilon)$-approximate k-median of P_e can be computed by computing the exact k-median on $(\bigcup_l \widetilde{C}^{l_k}) \bigcup \widetilde{C}^0$, using the algorithm in (Har-Peled and Mazumdar (2004)).

The process above describes how to compute the $(1+\varepsilon)$-approximate k-median clustering for only one update epoch. To continuously maintain the k-median clustering on data received in all previous epochs, an EH-summary needs to be maintained at the root that covers all previous epochs. Let EH_{root}^{past} denote the EH-summary for all past epochs. EH_{root}^{past}can be incrementally updated by merging with the EH-summary EH_{root} for epoch e, using Algorithm 2. $EH_{root}^{past} \leftarrow Merge(EH_{root}^{past}, EH_{root})$. This summary is efficient, with guaranteed approximation and can be incrementally maintained. Performing the exact k-median clustering on this summary will always produce a $(1+\varepsilon)$-approximate k-median clustering.

Overall Analysis

The transmission cost for the topology-oblivious algorithm can be derived as follows. For each site, the EH-summary is transferred instead of the original stream data. Assume that the EH-summary is $EH = \{\widetilde{C}^0, \widetilde{C}^{l_1}, \ldots, \widetilde{C}^{l_k}, \ldots, \widetilde{C}^{l_K}\}$. The size of EH depends on the size of coreset $\widetilde{C}^{l_k}$ at each level l_k, and the number of levels. The size of any $\widetilde{C}^{l_k}$ is no more than $O(\frac{k}{\varepsilon^d_{l_k}} \log N) = O(\frac{kl_k\sqrt{l_k}}{\varepsilon^d} \log N)$, which is bounded by $O(\frac{k}{\varepsilon^d} \log^2 N\sqrt{\log N})$,

since $l_k < \log N$. Totally, there are no more than $\log N$ levels. Therefore, the EH-summary size is $O(\frac{k}{\varepsilon^d}\log^3 N\sqrt{\log N})$. Overall, the communication for all the sites is no more than the number of sites times the max per node transmission bound.

4.4.2 Height-aware algorithm

This section consider the case when the height of the tree h is known. A height-aware algorithm is proposed to further reduce the transmission size. The basic idea of the algorithm is to compute a coreset on top of the EH-summary and use it as the local summary for transmission. Each internal node computes another coreset after merging all the coresets from the children with the local coreset. Both local coreset computation and coreset combination are incorporated into the algorithm shown below:

Theorem 4.4.3. *The coreset C_T computed at the root node using Algorithm 3 is an ε-coreset of the data received over all streams.*

At any node, let $EH = \{\widetilde{C}^0, \widetilde{C}^{l_1}, \ldots, \widetilde{C}^{l_k}, \ldots, \widetilde{C}^{l_K}\}$ be the EH-summary after the first step of Algorithm 3. According to Lemma 4.2, $(\bigcup_{l_k}\widetilde{C}^{l_k})\bigcup\widetilde{C}^0$ is an $(\frac{\varepsilon}{2}-\frac{\varepsilon}{3\sqrt{L}})$-coreset of P, since $\varepsilon_L = \sum_1^L \frac{1}{l\sqrt{l}}\frac{\varepsilon}{6} \leq \frac{\varepsilon}{2}-\frac{\varepsilon}{3\sqrt{L}}$. In Step 2, a $\frac{1}{3\sqrt{L}}\varepsilon$-coreset C_{EH} on EH is computed, which makes C_{EH} an $\frac{\varepsilon}{2}$-coreset of P. Whenever we move up from a node to its parent in the tree, the combination of coresets in Steps 3 and 4 increases the error of coreset by $\frac{\varepsilon}{2h}$. Since the total height is h, the final coreset C_T at root will be a ε-coreset of P ($\frac{\varepsilon}{2}+\frac{\varepsilon}{2h}h = \varepsilon$). In this algorithm, only the height of the tree h is required. Sites do not know their locations in the tree.

Overall Analysis The transmission cost for the height-aware algorithm can be derived as follows. For each site, a single coreset C_T (Algorithm 3, Step 4) is computed and transferred instead of an EH-summary as in topology-oblivious algorithm. C_T is an ε-coreset of C_U, where $|C_U| < N$, thus $|C_T| = O(\frac{k}{\varepsilon^d/2h}\log N)$. Therefore, the max per node transmission for height-aware algorithm is $O(\frac{kh}{\varepsilon^d}\log N)$, which is smaller compared

Algorithm 4.2 CombineEHSummary(EH,EH_*)

$EH_{all} \leftarrow \Phi$, $l \leftarrow 0$
$\widetilde{C}^0_{all} \leftarrow \widetilde{C}^0 \bigcup \widetilde{C}^0_*$
if $|\widetilde{C}^0_{all}| > B$ **then**
  Divide $\widetilde{C}^0_{all}$ into block B_c of size B and set the remaining part to be $\widetilde{C}^0_{all}$. Add $\widetilde{C}^0_{all}$ into EH_{all}
  Compute $\Delta\varepsilon_1$-coreset $\Delta\widetilde{C}^1_{all}$ over Blk
else
  Add $\widetilde{C}^0_{all}$ into EH_{all}
  $\Delta\widetilde{C}^1_{all} = \Phi$
end if
for $l = 1$ to L **do**
  $\Delta\widetilde{C}^{l+1} = \Phi$
  if $\widetilde{C}^l \in EH$ and $\widetilde{C}^l_* \in EH_*$ **then**
    Compute $\Delta\widetilde{C}^{l+1}_{all}$ as the $\Delta\varepsilon_{l+1}$-coreset over $\widetilde{C}^l \bigcup \widetilde{C}^l_*$
    if $\Delta\widetilde{C}^l_{all} \neq \Phi$ **then**
      $\widetilde{C}^l_{all} \leftarrow \Delta\widetilde{C}^l_{all}$, add $\widetilde{C}^l_{all}$ into EH_{all}.
    end if
  else
    if $\Delta\widetilde{C}^l_{all} \neq \Phi$ **then**
      if $\widetilde{C}^l \in EH$ **then**
        Compute ΔC^{l+1}_{all} as the $\Delta\varepsilon_{l+1}$-coreset over $\widetilde{C}^l \bigcup \Delta\widetilde{C}^l_{all}$
      else
        if $\widetilde{C}^l_* \in EH$ **then**
          Compute ΔC^{l+1}_{all} as the $\Delta\varepsilon_{l+1}$-coreset over $\widetilde{C}^l_* \bigcup \Delta\widetilde{C}^l_{all}$
        else
          $\widetilde{C}^l_{all} \leftarrow \Delta\widetilde{C}^l_{all}$, add $\widetilde{C}^l_{all}$ into EH_{all}
        end if
      end if
    else
      if $\widetilde{C}^l \in EH$ **then**
        $\widetilde{C}^l_{all} \leftarrow \widetilde{C}^l$, add $\widetilde{C}^l_{all}$ into EH_{all}
      else
        if $\widetilde{C}^l_* \in EH$ **then**
          $\widetilde{C}^l_{all} \leftarrow \widetilde{C}^l_*$, add $\widetilde{C}^l_{all}$ into EH_{all}
        end if
      end if
    end if
  end if
end for

Algorithm 4.3 CombineCoresetHW(P, h)

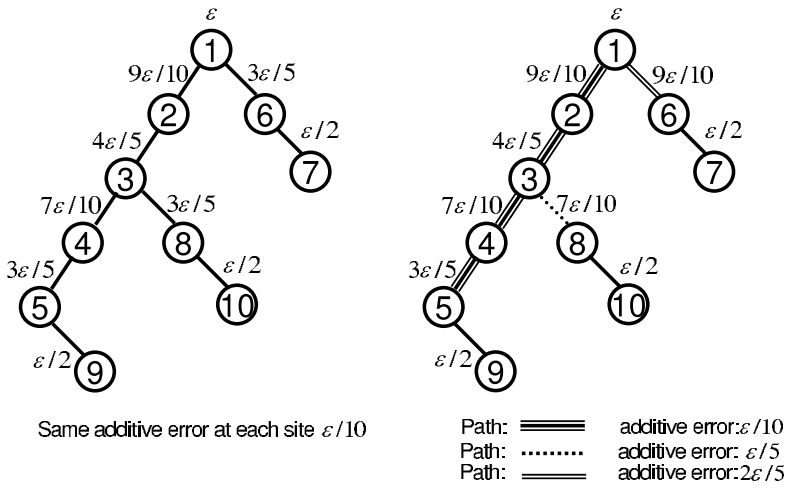
1: Compute the local EH-summary $EH = \{\widetilde{C}^0, \widetilde{C}^{l_1}, \ldots, \widetilde{C}^{l_k}, \ldots, \widetilde{C}^{l_K}\}$ using Algorithm 1, with $\Delta\varepsilon_l = \frac{1}{l\sqrt{l}}\frac{\varepsilon}{6}$
2: Compute a $\frac{1}{3\sqrt{l_K}}\varepsilon$-coreset C_{EH} on $(\bigcup_{l_k} \widetilde{C}^{l_k}) \bigcup \widetilde{C}^0$
3: Take the union of C_{EH} with all the coresets C^j_{EH} received from the child nodes $C_U = C_{EH} \bigcup (\bigcup_j C^j_{EH})$
4: Compute an $\frac{\varepsilon}{2h}$-coreset C_T on C_U
5: Return C_T as the final coreset for transmission

with the topology-oblivious case considering small h. However, the prior knowledge of the height of the tree is required.

4.4.3 Path-aware algorithm

In height-aware algorithm, the additive approximation factor $\frac{\varepsilon}{2h}$ is uniformly assigned to each site. Assume that each site is aware of the height of the subtree rooted at itself. This information can be obtained during the routing process. With the extra information about the topology, a path-aware algorithm is proposed which assigns approximation factor uniformly along each *path*. The path-aware algorithm further reduces data transmission compared with height-aware algorithm.

In the height-aware algorithm, each site computes an $\frac{\varepsilon}{2}$-coreset of the local data, merges all coresets from the children, and computes another $\frac{\varepsilon}{2h}$-coreset on top of it. It works well for balanced tree. However, for unbalanced tree, the transmission of the sites on a shorter path in the tree can be further reduced. For example, in Fig.2(a), the height of the tree is 5, so that the additive approximation factor at each site is $\frac{\varepsilon}{10}$, which is determined by the longest path in the tree (path $9 \to 5 \to 4 \to 3 \to 2 \to 1$). However, for shorter path such as $7 \to 6 \to 1$, the additive approximation factor can be taken as $\frac{2\varepsilon}{5}$ instead of $\frac{\varepsilon}{10}$ without affecting the final approximation factor ε at the root. This approach saves communication because the coreset size is inversely proportional to the approximation factor. Therefore, the additive error on each path can be determined separately. In the following discussion, a top-down algorithm is proposed for assigning



(a)Height-aware (b)Path-aware

Figure 4.2: Error accumulation of Height-aware and Path-aware algorithms. This figure compares the different strategies of assigning additive approximation factors at each site of the tree for height-aware and path-aware algorithms. Height-aware algorithm assigns the additive error uniformly to $\frac{\varepsilon}{2h}$, where h is the height of the tree. Path-aware algorithm assigns the additive error uniformly inside each sub-path, but differently for different sub-paths.

additive approximation factor $\Delta\varepsilon$ in a piecewise fashion along each path, assuming that each node knows the height of the subtree rooted at itself.

The algorithm proceeds as follows. Initially, AssignEps ($root$, $\frac{\varepsilon}{2}$) is invoked and the algorithm runs recursively and traverses the tree to assign the proper additive approximation factor for each site. In Fig.2(b), for example, the algorithm starts with AssignEps(S_1, $\varepsilon/2$), and set the additive error at root to be $\varepsilon/10$ (height of the tree $h = 5$). Two children of S_1 are S_2 and S_6, where $height(S_2) = 4$, and $height(S_6) = 1$. Therefore, the additive approximation factor at S_2 is assigned to be $\varepsilon_{SC_2} = \varepsilon/10$. Similarly, the additive approximation factor at S_6 is $\varepsilon_{SC_6} = 2\varepsilon/5$. Note that ε_{SC_6} is assigned to be $\varepsilon/10$ in height-aware algorithm. S_8 is also assigned $\varepsilon/5$ instead of $\varepsilon/10$.

After the additive error assignment phrase, the remaining part of the path-aware algorithm is the same as that of the height-aware algorithm.

Algorithm 4.4 AssignEps(S_i,ε_i)
Input S_i: current site; ε_i: the maximum possible approximation factor of the coreset sent by S_i's children

1: **if** S_i is leaf **then**
2: return
3: **end if**
4: **if** S_i is root **then**
5: $\varepsilon_{S_i} = \varepsilon_i/h$, where h is the height of the entire routing tree
6: $\varepsilon_i = \varepsilon_i - \varepsilon_{S_i}$
7: **end if**
8: **for** each site in S_i's children set $\{SC_j\}$ **do**
9: **if** SC_j is a leaf **then**
10: Assign the additive approximation factor of SC_j to be $\varepsilon_{SC_j} = \varepsilon_i$
11: **else**
12: Assign the additive approximation factor of SC_j to be $\varepsilon_{SC_j} = \frac{\varepsilon_i}{height(SC_j)}$, where $height(SC_j)$ is the height of the subtree rooted at SC_j.
13: **end if**
14: AssignEps(SC_j,$\varepsilon_i - \varepsilon_{SC_j}$)
15: **end for**

4.5 Experiments and Analysis

The algorithms are simulated using different tree topologies with up to 50 nodes. The routing tree is generated by placing the sites on a 100×100 grid. It is assumed that the sites can communicate with other sites within a distance of 5. Based on this assumption, a site graph is generated and the distances between the sites are used as the weights of the site graph edges. A spanning tree algorithm is then applied to generate the routing tree.

4.5.1 Benchmark Data

The algorithms are tested on both real and synthetic datasets with up to millions of data points. Specifically, the algorithms are tested on the following two data sets:

- **New York stock exchange (NYSE) data**: An archived dataset representing data volumes at the end of each day in the New York stock exchange[2]. The data

[2]http://www.nyse.com/marketinfo/datalib/1022221393023.html

volume information is collected at the stock exchange over a hundred years. The overall dataset has over 30K observations. The trading data volume can vary significantly from day to day.

- **Synthetic data**: The synthetic data is generated using a weighted combination of normalized distributions at a given set of centers. 15 centers are randomly chosen in the user-specified data range. The algorithms are tested on up to 2 million observations.

4.5.2 Results and Analysis

The experiments are conducted in different configurations and analyzed the data transmission as a function of the total stream data size, the approximation bound in k-median computation, and the number of sites. The experimental results are presented and compared with the theoretical bounds of the algorithms.

Stream Data Size

The max per node transmission is asymptotically bounded by a *polylog* function of total stream size N. For topology-Oblivious algorithm, the max per node bound is $Bound_{max} = O(\frac{k}{\varepsilon^d} \log^3 N \sqrt{\log N})$; for height-aware algorithm, the max per node bound is $Bound'_{max} = O(\frac{kh}{\varepsilon^d} \log N)$. Thus, the total data transmission size is asymptotically bounded by the number of sites times $Bound_{max}$. The advantage of the algorithms is more prominent for large data volume.

Fig.3 shows the overall and max per node transmission for synthetic and real data with varying per-epoch data volume of the input streams. For the experiments on both real data and synthetic data, the parameters used are $k = 10$, $\varepsilon = 0.05$. For real data, all three proposed algorithms are tested on a simulated stream system of 5 nodes, with total stream size varying between $20K$ and $28K$. For synthetic data, all three algorithms are

tested on a simulated stream system of 10 nodes, with total stream size varying between $1M$ and $9M$.

Figs. 4.3(a) and 4.3(b) demonstrate the overall communication load of all three algorithms. In 4.3(a) and 4.3(b) it can be observed that the total communication of topology oblivious algorithm is more than height-aware algorithm, and height-aware algorithm is more than path-aware algorithm. All three algorithms are well below the corresponding theoretical bounds for total data transmission. Figs. 4.3(c) and 4.3(d) highlight the maximum per node data transmission of the algorithms on the real and synthetic data. The plots also indicate a *polylog* relationship between data transmission and total stream size. In addition, height-aware algorithm and path-aware algorithm have much smaller max per node transmission than topology-oblivious algorithm. In the experiments, a significant reduction can be observed in both the total and max per node data transmission, and more reduction for larger stream size. We noticed that the max per node transmission of path-aware algorithm is only a small fraction ($\approx 20\%$) of the overall stream size for real data (see Fig. 4.3(c)) and ($\approx 2\%$) for synthetic data (see Fig. 4.3(d)).

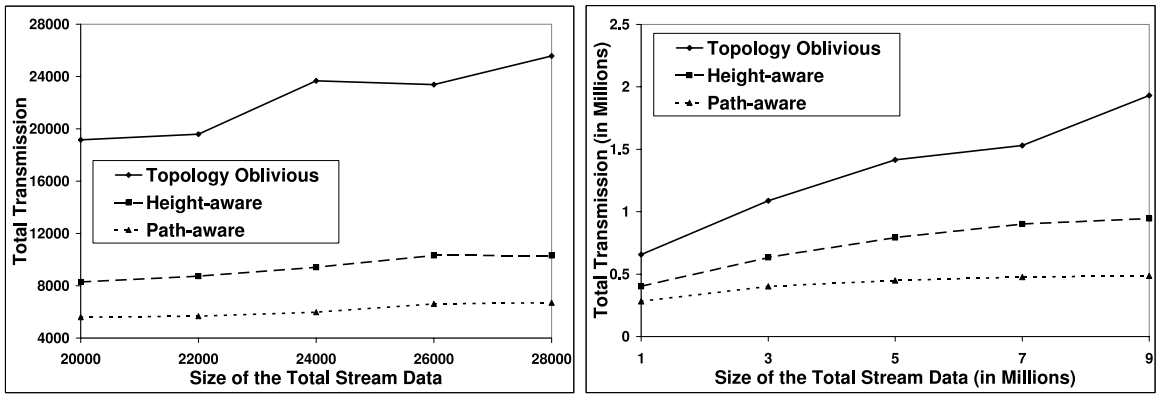
Number of Sites

The total transmission and max per node transmission are evaluated by varying the number of sites. The total data transmission is asymptotically linear to the number of sites. Figs. 4.4(a) and 4.4(b) highlight the total data transmission as a function of the number of sites for both NYSE dataset and synthetic dataset. The total input stream size is fixed to $30K$ for the real data and $5M$ for the synthetic data. It can be observed that the total data transmission by all three algorithms are below the theoretical bound. The graphs indicate that the height-aware algorithm performs better than the topology-oblivious algorithm, and the path-aware algorithm performs better than the height-aware algorithm. Figs. 4.4(c) and 4.4(d) highlight the maximum per

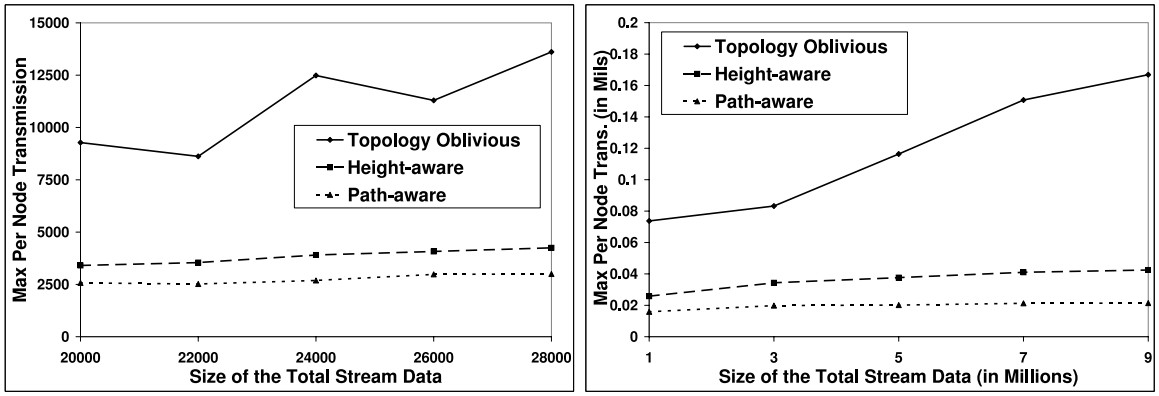
node data transmission as a function of the number of sites in the network. The height-aware and path-aware algorithms slowly increases with larger number of sites due to the increases of the height of the tree. The topology-oblivious algorithm does not exhibit increasing tendency with the change of the number of sites. The ups and downs of the topology-oblivious curve are due to the different distribution of the total stream data in different tree topologies (with increasing sensor numbers). The graphs demonstrate the scalability of the algorithms in terms of the number of sites.

Approximation Error

The data transmission is expected to reduce when a larger approximation error is allowed. Using the same datasets, the experiments demonstrate that the total transmission in all three algorithms declines slowly as the approximation error increases on both synthetic and real data, as shown in Figs. 4.5(a) and 4.5(b). Figs. 4.5(c) and 4.5(d) highlight the maximum per node data transmission as a function of the approximation error, which also decreases slowly with larger approximation error. It can be observed that the path-aware algorithm can reduce the total data transmission and the maximum data transmission per node by additional 30%.

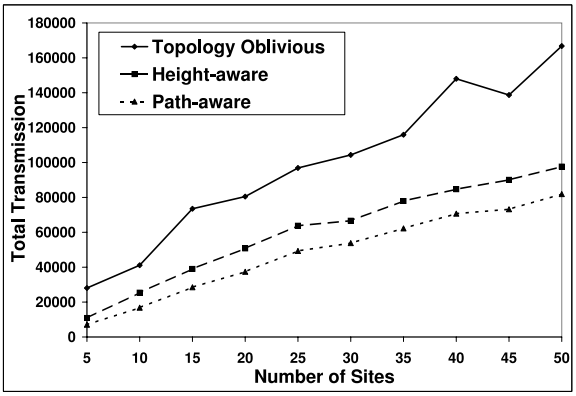


(a) *Overall communication on real data* (b) *Overall communication on synthetic data*

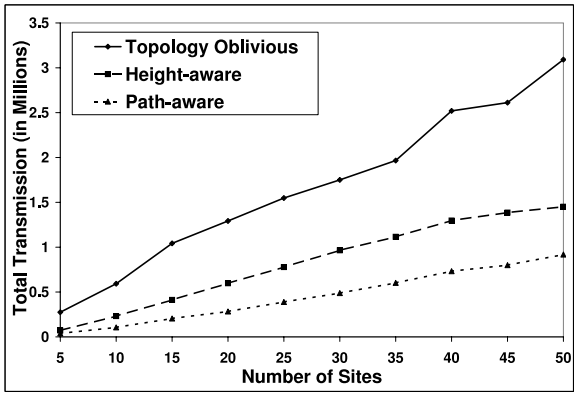


(c) *Maximum per node communication on real data* (d) *Maximum per node communication on synthetic data*

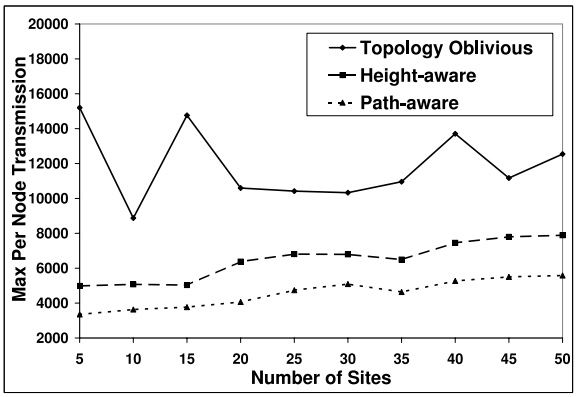
Figure 4.3: Performance of the algorithms as a function of the total stream size: The overall communications among the sensor network nodes perform k-median clustering are measured using real and synthetic data. The NYSE data consists up to 28K records and the synthetic data consists up to 9 million data values. The approximation error threshold is set to 0.05. For real data, all three algorithms are tested on a 5-node system. For synthetic data, the algorithms are tested on a 10-node system. Figs. 4.3(a) and 4.3(b) demonstrate the overall data communication of the algorithms as a function of input data size. Figs. 4.3(c) and 4.3(d) demonstrate the max per node data communication of the algorithms as a function of input data size. The experiments demonstrate a significant reduction in the overall and max per node communication.



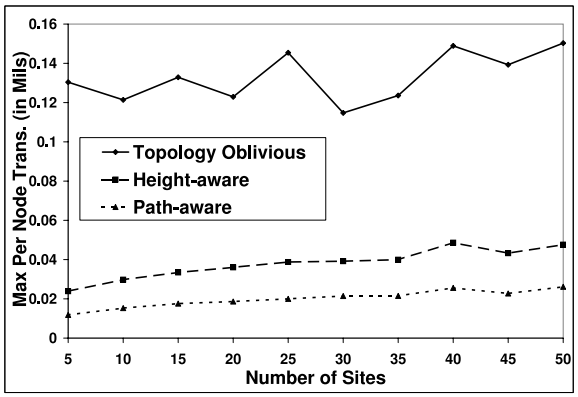
(a) *Normalized overall communication on real data*



(b) *Normalized overall communication per node on synthetic data*

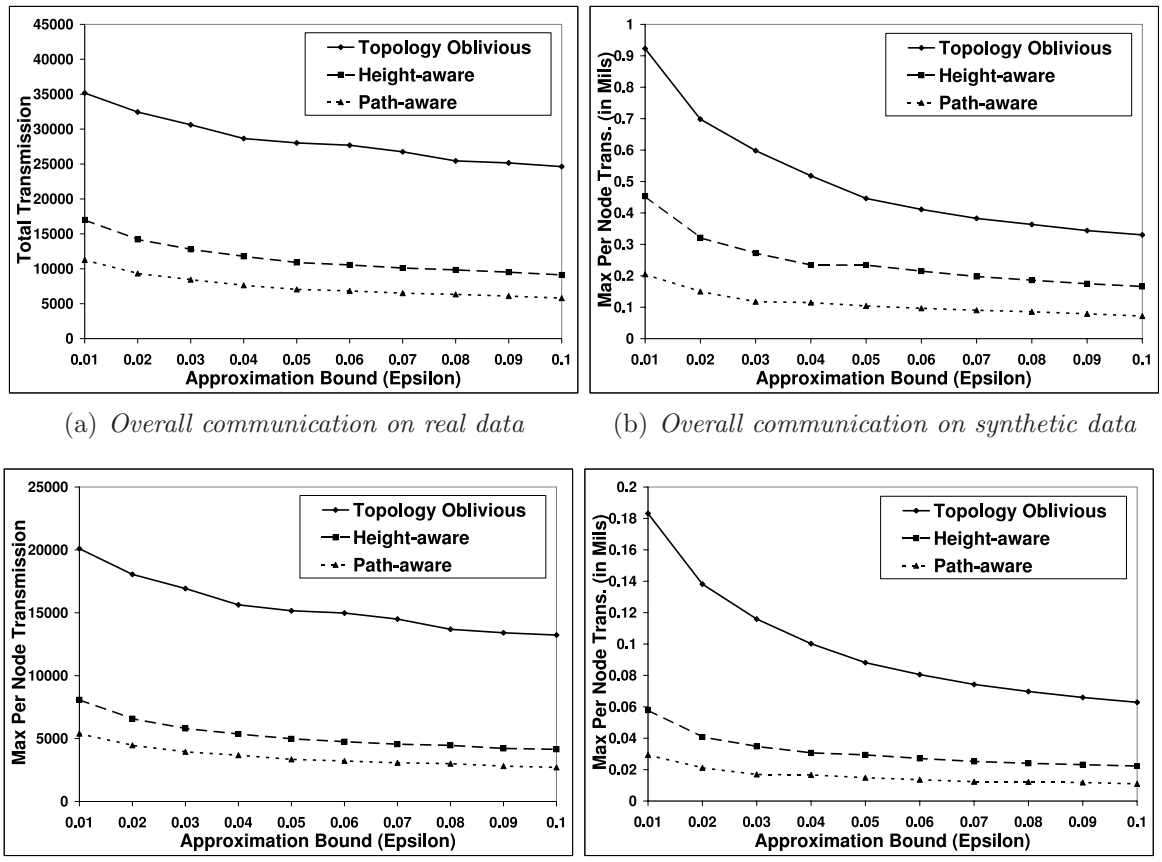


(c) *Normalized maximum per node communication on real data*



(d) *Normalized maximum per node communication on synthetic data*

Figure 4.4: Performance of the algorithms as a function of the number of sites. The data communication of the algorithms is measured as a function of the number of sites on NYSE and synthetic data.



(a) *Overall communication on real data*

(b) *Overall communication on synthetic data*

(c) *Maximum per node communication on real data*

(d) *Maximum per node communication on synthetic data*

Figure 4.5: Performance of the algorithms as a function of approximation error: The overall communication of the algorithm decreases as the error increases. Figs. 4.5(a) and 4.5(b) highlight the overall data communication as the error increases. Figs. 4.5(c) and 4.5(d) demonstrate the max per node data communication as the error increases. Approximate clustering is performed on NYSE data with 30K data records and synthetic data with 5 million observations. As the error tolerance increases, it can be observed that both the height-aware and path-aware algorithms perform better than the topology-oblivious algorithm and can further reduce the communication by additional $10 - 30\%$.

Chapter 5

Fast Algorithms for Approximate Order-Statistics Computation in Data Streams

5.1 Introduction

Order-statistics such as quantiles and biased-quantiles capture the underlying distribution of the datasets. Computing exact quantiles or biased-quantiles on large datasets or unlimited data streams requires either huge memory or disk-based sorting. It is proven that at least $O(N^{\frac{1}{p}})$ storage is needed for exact median (0.5 quantile) computation in p passes for a dataset of size N (Munro and Paterson (1980)). Recently, researchers have studied the problem of computing approximate quantiles with guaranteed error bound to improve both memory and speed performance (Manku et al. (1998, 1999); Greenwald and Khanna (2001, 2004); Arasu and Manku (2004); Lin et al. (2004); Cormode et al. (2005, 2006)).

Streaming quantile or biased-quantile computation faces several chanllenges. Data streams are transient and can arrive at a high speed. Different from static datasets with fixed size, the streams are usually with unbounded size. Streaming computations therefore require single pass algorithms with small space requirement which can handle

arbitrary sized streams. In order to guarantee the precision of the result, the algorithm should ensure random or deterministic error bound for the quantile computation. Many algorithms were developed for computing approximate quantiles over the entire stream history (Manku et al. (1998); Greenwald and Khanna (2001)) or over a sliding window (Lin et al. (2004); Arasu and Manku (2004)); with uniform error (Manku et al. (1998); Greenwald and Khanna (2001)) or with biased error (Cormode et al. (2005, 2006)). However, most of these algorithms focus on reducing the space requirement and can trade off the computational cost, which is one of the important concerns for high-speed data streams. In this thesis, fast algorithms are presented for computing approximate quantiles and biased-quantiles over data streams.

5.2 Related Work

Quantile computation has been studied extensively in the database literature. At a broad level, they can be classified as exact algorithms and approximate algorithms.

Exact Algorithms: Several algorithms are proposed for computing exact quantiles efficiently. There is also considerable work on deriving the lower and upper bounds of number of comparisons needed for finding exact quantiles. Mike Paterson (Paterson (1997)) reviewed the history of the theoretical results on this aspect. The current upper bound is $2.9423N$ comparisons, and the lower bound is $(2 + \alpha)N$, where α is the order of 2^{-40}. Munro and Paterson (Munro and Paterson (1980)) also showed that algorithms which compute the exact ϕ-quantile of a sequence of N data elements in p passes, will need $\Omega(N^{1/p})$ space. For single pass requirement of stream applications, this requires $\Omega(N)$ space. Therefore, approximation algorithms that require sublinear-space are needed for online quantile computations on large data streams.

Approximate Algorithms: Approximate algorithms are either deterministic with guaranteed error or randomized with guaranteed error of certain probability. These al-

gorithms can further be classified as uniform, biased or targetted quantile algorithms. Moreover, based on the underlying model, they can be further classified as quantile computations on entire stream history, sliding windows and distributed stream algorithms.

Jain and Chlamatac (Jain and Chlamtac (1985)), Agrawal and Swami (Agrawal and Swami (1995)) have proposed algorithms to compute uniform quantiles in a single pass. However, both of these two algorithms have no *apriori* guarantees on error. Manku *et al.* (Manku et al. (1998)) proposed a single pass algorithm for computing ϵ-approximate uniform quantile summary. Their algorithm requires prior knowledge of N. The space complexity of their algorithm is $O(\frac{1}{\epsilon}\log^2 \epsilon N)$. Manku *et al.* (Manku et al. (1999)) also presented a randomized uniform quantile approximation algorithm which does not require prior knowledge of N. The space requirement is $\frac{1}{\epsilon}(\log^2(\frac{1}{\epsilon}) + \log^2\log(\frac{1}{\delta}))$ with a failure probability of δ. Greenwald *et al.* (Greenwald and Khanna (2001)) improved Manku's (Manku et al. (1999)) algorithm to achieve a storage bound of $O(\frac{1}{\epsilon}\log \epsilon N)$. Their algorithm can deterministically compute an ϵ-approximate quantile summary without the prior knowledge of N. Lin *et al.* (Lin et al. (2004)) presented algorithms to compute uniform quantiles over sliding windows. Arasu and Manku (Arasu and Manku (2004)) improved the space bound using a novel exponential histogram-based data structure.

In addition to exact and approximate algorithms for quantile computation, recent work has also studied biased quantile computation in data streams and quantile computation in sensor network systems.

Biased Quantile Computation in Streams: Cormode *et al.* (Cormode et al. (2005)) first studied the problem of biased quantiles computation in data streams. They proposed an algorithm with poly-log space complexity based on (Greenwald and Khanna (2001)). However, it is shown in (Zhang et al. (2006)) that the space requirement of their algorithm can grow linearly with the input size with carefully crafted data. Cormode *et al.* (Cormode et al. (2006)) presented a better algorithm with an improved space bound of $O(\frac{logU}{\epsilon}\log \epsilon N)$ and amortized update time complexity of $O(\log\log U)$ where U is the

size of the universe where data element is chosen from and N is the size of the data stream.

Quantile Computation in Sensor Network: Recent work has also focussed on approximate quantile computation algorithms in distributed streams and sensor networks. Greenwald *et al.* (Greenwald and Khanna (2004)) proposed an algorithm for computing ϵ-approximate quantiles distributely for sensor network applications. Their algorithm guarantees that the summary structure at each sensor is of size $O(\log^2 n/\epsilon)$. Shrivastava *et al.* (Shrivastava et al. (2004)) presented an algorithm to compute medians and other quantiles in sensor networks using a space complexity of $\mathrm{O}(\frac{1}{\epsilon}\log(U))$ where U is the size of the universe.

5.3 Approximate Quantile Computation

5.3.1 Algorithms

This section describes the algorithms for fast computation of approximate quantiles on large high-speed data streams. The data structures and algorithms are presented for both fixed-sized (with known size) and arbitrary-sized (with unknown size) streams. Furthermore, the analysis of computational complexity and the memory requirements are given for the algorithms.

Let N denote the total number of values of the data stream and n denote the number of values in the data stream seen so far. Given a user-defined error ϵ and any rank $r \in [1, n]$, an *ϵ-approximate quantile* is an element in the data stream whose rank $r\prime$ is within $[r - \epsilon n,\ r + \epsilon n]$. A summary structure is maintained to continuously answer ϵ-approximate quantile queries.

Fixed Size Streams

First consider the case where N is given in advance. The solution for fixed size streams will be generalized for streams with unknown N in the following subsection. In practice, the former algorithm can be used for applications such as summarizing large databases that do not fit in main memory. The latter algorithm is useful for continuous streams whose size can not be predicted beforehand. The summary structure as well as the construction algorithm for the summary structure are described as below.

Multi-level Quantile Summary A multi-level ϵ-summary S of the stream is main-

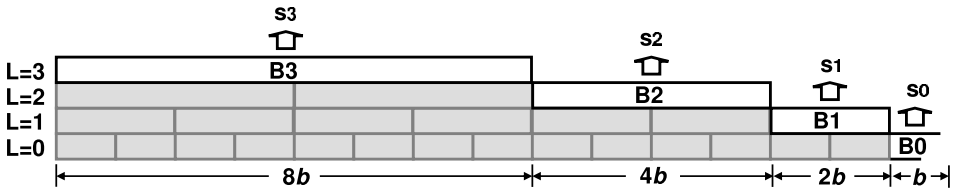


Figure 5.1: Multi-level summary S: This figure highlights the multi-level structure of the ϵ-summary $S = \{s_0, s_1, \ldots, s_L\}$. The incoming data is divided into equi-sized blocks of size b and blocks are grouped into disjoint bags, $B_0, B_1, \ldots, B_l, \ldots, B_L$ with B_l for level l. B_0 contains the most recent block, B_1 contains the older two blocks, and B_L consists of the oldest 2^L blocks. At each level, s_l is maintained as the ϵ_l-summary for B_l. The total number of levels L is no more than $\log \frac{N}{b}$

.

tained as data elements are coming in. An *ϵ-summary* is a sketch of the stream which can provide ϵ-approximate answer for quantile query of any rank $r \leq n$. A multi-level summary structure $S = \{s_0, ..., s_l, ..., s_L\}$ is maintained, where s_l is the summary at level l and L is the total number of levels (see Fig.1). Basically, the incoming stream is divided into blocks of size b ($b = \lfloor \frac{\log \epsilon N}{\epsilon} \rfloor$). Each level l covers a disjoint bag B_i of consecutive blocks in the stream, and all levels together $\bigcup B_i$ cover the whole stream. Specifically, B_0 always contains the most recent block (whether it is complete or not), B_1 contains the older two blocks, and B_L consists of the oldest 2^L blocks. Each s_l is an ϵ_l-summary of B_i, where $\epsilon_l \leq \epsilon$.

The multi-level summary construction and maintainance is performed as follows.

Initially, all levels are empty. Whenever a data element in the stream arrives, the update procedure is performed as follows.

1. Insert the element into s_0.

2. If s_0 is not full ($|s_0| < b$), stop and the update procedure is done for the current element. If s_0 becomes full ($|s_0| = b$), the size of s_0 is reduced by computing a sketch s_c of size $\lceil \frac{b}{2} \rceil + 1$ on s_0. This sketch computation operation is referred to as `COMPRESS`, which will be described in detail in later discussion. Consider s_0 as an ϵ_0-summary of B_0 where $\epsilon_0 = 0$, the `COMPRESS` operation guarantees that s_c is an $(\epsilon_0 + \frac{1}{b})$-summary. After `COMPRESS` operation, s_c is sent to level 1.

3. If s_1 is empty, s_1 is set to be s_c and the update procedure is done. Otherwise, s_1 is merged with s_c which is sent by level 0 and empty s_0. These operations are referred to as `MERGE` on s_1, s_c and `EMPTY` on s_0. Generally, the `MERGE(`s_{l+1}`, `s_c`)` operation merges s_{l+1} with the sketch s_c by performing a merge sort. The `EMPTY(`s_l`)` operation empties s_l after `MERGE` operation is finished. Finally, `COMPRESS` is performed on the result of `MERGE`, and the resulting sketch s_c is sent to level 2.

4. If s_2 is empty, s_2 is set to be s_c and the update procedure is done. Otherwise, the operations s_2 =`MERGE`(s_2, s_c), s_c =`COMPRESS`(s_2), `EMPTY`(s_1) are performed in the given order, and new s_c is sent to level 3.

5. Step 4 is repeatedly performed for $s_i, i = 3, \ldots, L$ until level L is reached where s_L is empty.

The pseudo code of the entire update procedure whenever an element e comes is shown in Algorithm 1. The detail of operations `COMPRESS, MERGE` will be given in the following sections.

Assume that s is an ϵ-summary of stream B. For each element e in s, $rmax(e)$ and $rmin(e)$ are maintained which represent the maximum and minimum possible ranks

Algorithm 5.1 Update(e,S,ϵ)
Input e: current data element to be inserted, S: current summary structure $S = \{s_0, \ldots, s_l, \ldots, s_L\}$, ϵ: required approximation factor of S

```
1: insert e into s_0
2: if |s_0| = b then
3:    sort s_0
4:    s_c ← compress(s_0, 1/b)
5:    empty(s_0)
6: else
7:    exit
8: end if
9: for l = 1 to L do
10:   if |s_l| = 0 then
11:      s_l ← s_c
12:      break
13:   else
14:      s_c = compress(merge(s_l, s_c), 1/b)
15:      empty(s_l)
16:   end if
17: end for
```

of e in B, respectively. Therefore, the ϵ-approximate quantile query of any rank r can be answered by returning the value e which satisfies: $rmax(e) \leq r + \epsilon|B|$ and $rmin(e) \geq r - \epsilon|B|$. Initially, $rmin(e) = rmax(e) = rank(e)$. $rmin(e), rmax(e)$ are updated during the `COMPRESS` and `MERGE` operations.

`COMPRESS`$(s, \frac{1}{b})$: The `COMPRESS` operation takes at most $\lceil \frac{b}{2} \rceil + 1$ values from s, which are: $quantile(s, 1)$, $quantile(s, \lfloor \frac{2|B|}{b} \rfloor)$, $quantile(s, \lfloor 2\frac{2|B|}{b} \rfloor)$, ...,$quantile(s, \lfloor i\frac{2|B|}{b} \rfloor)$,..., $quantile(s, |B|)$, where $quantile(s, r)$ queries summary s for quantile of rank r. According to (Greenwald and Khanna (2004)), the result of `COMPRESS`$(s, \frac{1}{b})$ is an $(\epsilon + \frac{1}{b})$-summary, assuming s is an ϵ-summary.

`MERGE`(s, s'): The `MERGE` operation combines s and s' by performing a merge-sort on s and s'. According to (Greenwald and Khanna (2004)), if s is an ϵ-summary of B and s' is an ϵ'-summary of B', the result of `MERGE`(s, s') is an $\overline{\epsilon}$-summary of $B \bigcup B'$ where $\overline{\epsilon} = max(\epsilon, \epsilon')$.

Lemma 5.3.1. *The number of levels in the summary structure is less than* $\log(\epsilon N)$.

Proof. In the entire summary structure construction, s_0 becomes full at most $\frac{N}{b}$ times, s_l becomes full $\frac{N}{2^l b}$ times and the highest level s_L becomes full at most once. Therefore,

$$L \leq \log(\frac{N}{b}) < \log(\epsilon N) - \log(\log(\epsilon N)) < \log(\epsilon N) \tag{5.1}$$

. □

Lemma 5.3.2. *Each level in the summary maintains an error less than ϵ.*

Proof. During the construction process of S, the error at each level ϵ_l depends on the `COMPRESS` and `MERGE` operations. Intially, $\epsilon_0 = 0$. At each level, `COMPRESS`$(s_l, \frac{1}{b})$ operation generates a new sketch s_c with error $\epsilon_l + \frac{1}{b}$ and added to level $l+1$, and `MERGE` does not increase the error. Therefore, the error of the summary in s_{l+1} is given by

$$\epsilon_{l+1} = \epsilon_l + \frac{1}{b} = \epsilon_0 + \frac{l+1}{b} = \frac{l+1}{b} \tag{5.2}$$

From equations 5.2 and 5.1, it is easy to verify that

$$\epsilon_l = \frac{l}{b} < \frac{\log(\epsilon N)}{\frac{\log(\epsilon N)}{\epsilon}} = \epsilon \tag{5.3}$$

□

To answer a query of any rank r using S, s_0 is first sorted and the summaries at all levels $\{s_l\}$ are merged using the `MERGE` operation, denote it as `MERGE`(S). Then the ϵ-approximate quantile for any rank r is the element e in `MERGE`(S) which satisfies: $rmin(e) \geq r - \epsilon N$ and $rmax(e) \leq r + \epsilon N$.

Theorem 5.3.3. *For multi-level summary S, `MERGE`(S) is an ϵ-approximate summary of the entire stream.*

Proof. `MERGE` operation on all s_l generates a summary for $\bigcup B_l$ with approximation factor $\epsilon_U = \max(\epsilon_1, \epsilon_2, \ldots, \epsilon_L)$. According to Lemma 2, $\epsilon_U < \epsilon$. Since the union of

all the B_l is the entire stream, `MERGE`(S) is an ϵ-approximate summary of the entire stream. □

Performance Analysis The summary structure maintains at most $b+3$ elements in each level (after `MERGE` operation) and there are L levels in the summary structure. Therefore, the storage requirement for constructing the summary is bounded by $(b+3)L = O(\frac{1}{\epsilon}\log^2(\epsilon N))$. The storage requirement for the algorithm is higher than the best storage bound proposed by Greenwald and Khanna (Greenwald and Khanna (2001)), which is $O(\frac{1}{\epsilon}\log(\epsilon N))$. However, The goal behind the algorithm is to achieve faster computational time with reasonable storage. In practice, the memory size requirements for the algorithm can be a small fraction of the RAM on most PCs even for peta-byte-sized datasets (see table 5.2).

Theorem 5.3.4. *The average update cost of the algorithm is* $O(\log(\frac{1}{\epsilon}\log(\epsilon N)))$.

Proof. At level 0, for each block, a sort and a `COMPRESS` operation are performed. The cost of sort per block is $b\log b$, `COMPRESS` per block is $\frac{b}{2}$. Totally, there are $\frac{N}{b}$ blocks, so the total cost at level 0 is: $N\log b + \frac{N}{2}$. At each level $L_i, i > 0$, a `COMPRESS` and a `MERGE` operation are performed. Each `COMPRESS` costs b, since a linear scan is required to batch query all the values needed (refer to `COMPRESS` operation). Each `MERGE` costs b with a merge sort. In fact, the computation cost of `MERGE` also includes the updates of *rmin* and *rmax* (will be discussed in Sec 3.3), which can be done in linear time. Thus the cost of a `MERGE` adds up to $2b$. Therefore, the total expected cost of computing the summary structure is $N\log b + \frac{N}{2} + \sum_{i=1}^{i=L}\frac{N}{2^i b}3b = O(N\log(\frac{1}{\epsilon}\log(\epsilon N))$. The average update time per element is $O(\log(\frac{1}{\epsilon}\log(\epsilon N)))$.

□

In practice, for a fixed ϵ, the average per element computation cost of the algorithm is given by $O(\log\log N)$ and the overall computation is almost linear in performance. The algorithm proposed by Greenwald and Khanna (Greenwald and Khanna (2001)) has

Stream size (N)	Maximum Block Size (Bytes)	Bound of Number of Tuples	Bound of Summary Size (Bytes)
10^6	191.2KB	161K	1.9MB
10^9	420.4KB	717K	8.6MB
10^{12}	669.6KB	1.67M	20MB
10^{15}	908.8KB	3.03M	36.4MB

Table 5.1: This table shows the memory size requirements of the Generalized algorithm (with unknown size) for large data streams with an error of 0.001. Each tuple consists of a data value, and its minimum and maximum rank in the stream, totally 12 bytes. Observe that the block size is less than a MB and fits in the L2 cache of most CPUs. Therefore, the sorting will be in-memory and can be conducted very fast. Also, the maximum memory requirement for the algorithm is a few MB even for handling streams of 1 peta data.

a best case computation time (per element) of $O(\log s)$, and worst computation time (per element) of $O(s)$ where s is $\frac{1}{\epsilon}\log(\epsilon N)$. The comparison of the performance will be demonstrated in the experiment section.

The majority of the computation in the summary construction is dominated by the sort operations on blocks. Although sorting is computationally intensive, it is fast on small blocks which fit in the CPU L2 caches. Table 5.2 shows a comparison of the block size, memory requirement as a function of stream size N with error bound 0.001 using the generalized streaming algorithm in the next section. In practice, the size of the blocks in the algorithm is smaller than the CPU cache size even for peta-byte-sized data streams.

Generalized Streaming Algorithm

The algorithm for fixed size streams can be generalized to compute approximate quantiles in streams without prior knowledge of size N. The basic idea of the algorithm is as follows. The input stream P is partitioned into disjoint sub-streams $P_0, P_1, \ldots, P_m$ with increasing size. Specifically, sub-stream P_i has size $\frac{2^i}{\epsilon}$ and covers the elements whose location is in the interval $[\frac{2^i-1}{\epsilon}, \frac{2^{i+1}-1}{\epsilon})$. By partitioning the incoming stream into sub-streams with known size, a multi-level summary S_i can be constructed on each sub-stream P_i using the algorithm for fixed size streams. The summary construction

Algorithm 5.2 gUpdate(e,$\overline{S}$, ϵ, S_C)
Input e: current data element, $\overline{S}$: current summary structure, $\overline{S} = \{\overline{S}_0, \overline{S}_1, \ldots, \overline{S}_{k-1}\}$ (sub-streams $P_0, \ldots, P_{k-1}$ have completely arrived), ϵ: required approximation factor of $\overline{S}$, S_C: the fixed size multi-level summary corresponding to the current sub-stream P_k, $S_C = \{s_0, s_1, \ldots, s_L\}$

1: **if** e is the last element of P_k **then**
2: Apply *merge* on all the levels of S_C: $s_{all} = merge(S_C) = merge(s_0, s_1, \ldots, s_L)$
3: $\overline{S}_k = compress(s_{all}, \frac{\epsilon}{2})$
4: $\overline{S} = \overline{S} \bigcup \{\overline{S}_k\}$
5: $S_C \leftarrow \phi$
6: **else**
7: update S_C: $S_C = Update(e, S_C, \frac{\epsilon}{2})$
8: **end if**

algorithm is as follows.

1. For the latest sub-stream P_k which has not completed, a multi-level ϵ'-summary S_C is computed using Algorithm 1 by performing $Update(e, S_C, \epsilon')$ whenever an element comes. Here $\epsilon' = \frac{\epsilon}{2}$.

2. Once the last element of sub-stream P_k arrives, an $\frac{\epsilon}{2}$-summary on `MERGE`(S_C) is computed, which is the merged set of all levels in S_C. The resulting summary $\overline{S}_k$ =`COMPRESS`(`MERGE`(S_C), $\frac{\epsilon}{2}$) is an ϵ-summary of P_k and it consists of $\frac{2}{\epsilon}$ elements.

3. The ordered set of the summaries of all complete sub-streams so far $\overline{S} = \{\overline{S}_0, \overline{S}_1, \ldots, \overline{S}_{k-1}\}$ is the current multi-level ϵ-summary of the entire stream except the incomplete sub-stream P_k.

The pseudo code for the update algorithm for stream with unknown size is shown in Algorithm 2. Initially, $\overline{S} = \phi$. Whenever an element comes, *gUpdate* is performed to update the summary structure $\overline{S}$.

To answer a query of any rank r using $\overline{S}$, if S_C is not empty, $\overline{S}_k$ is first computed for the incomplete sub-stream P_k: $\overline{S}_k = compress(merge(S_C), \frac{\epsilon}{2})$, then all the ϵ-summaries $\overline{S}_0, \overline{S}_1, \ldots, \overline{S}_{k-1}$ in $\overline{S}$ are merged together with $\overline{S}_k$ using `MERGE` operation, the final summary is the ϵ-summary for P.

Performance Analysis The analysis storage complexity as well as computational complexity of the algorithm is given as below.

Theorem 5.3.5. *The space requirement of Algorithm 2 is* $O(\frac{1}{\epsilon}\log^2(\epsilon n))$.

Proof. At any point of time, assume that the number of data elements arriving so far is n. According to Algorithm 1, a multi-level ϵ-approximate summary S_C is computed and maintained for the current sub-stream P_k. For each of the previous sub-streams $P_i, i = 1, \ldots, k-1$ which are complete, ϵ-summary $\overline{S}_i$ of size $\frac{2}{\epsilon}$ is maintained. Since $k \leq \lfloor \log(\epsilon n + 1) \rfloor$, totally $O(\frac{1}{\epsilon}\log \epsilon n)$ space is required. According to the space bound for fixed size streams, $\mathrm{O}(\frac{1}{\epsilon}\log^2(\epsilon n))$ space is required for computing the summary S_C for the current sub-stream. Therefore, the space requirement for the entire algorithm at any point of time is $O(\frac{1}{\epsilon}\log^2(\epsilon n))$. □

Theorem 5.3.6. *The average update cost of Algorithm 2 is* $O(\log(\frac{1}{\epsilon}\log \epsilon n))$.

Proof. According to Theorem 2, the computational complexity of each sub-stream $P_i, i = 0, 1, \ldots, \lfloor \log(\epsilon n + 1) \rfloor$ is $\mathrm{O}(n_i \log(\frac{1}{\epsilon'}\log(\epsilon' n_i)))$ where $n_i = |P_i| = \frac{2^i}{\epsilon}, \Sigma n_i = n, \epsilon' = \frac{\epsilon}{2}$. After each sub-stream P_i is complete, an additional `MERGE` and `COMPRESS` operation are performed each of cost $O(\frac{1}{\epsilon'}\log^2(\epsilon' n_i))$ to construct $\overline{S}_i$.

Given the above observations, the total computational cost of the algorithm is

$$\sum_{i=0}^{i=\lfloor \log(\epsilon n+1) \rfloor} (\frac{2^i}{\epsilon}\log(\frac{2(i-1)}{\epsilon}) + \frac{2}{\epsilon}(i-1)^2) \tag{5.4}$$

.

Simplifying equation 5.23, the total computational cost of the algorithm is $O(n\log(\frac{1}{\epsilon}\log(\epsilon n)))$, the average updating time per element is $O(\log(\frac{1}{\epsilon}\log(\epsilon n)))$, which is $O(\log\log n)$ if ϵ is fixed. □

Update $rmin(e)$ and $rmax(e)$

For both fixed size stream and arbitrary size stream, to answer the quantile query, $rmin$ and $rmax$ values of each element e in the summary need to be updated properly.

$rmin(e)$ and $rmax(e)$ are updated during `COMPRESS` and `MERGE` operations as follows (as in (Greenwald and Khanna (2004))):

Rank update in `MERGE`: Let $S' = x_1, x_2, \ldots, x_a$ and $S'' = y_1, y_2, \ldots, y_b$ be two quantile summaries. Let $S = z_1, z_2, \ldots, z_{a+b} = \texttt{MERGE}(S', S'')$. Assume z_i corresponds to some element x_r in Q'. Let y_s be the largest element in S'' that is smaller than x_r (y_s is undefined if no such element), and let y_t be the smallest element in S'' that is larger than x_r (y_t is undefined if no such element). Then

$$rmin_S(z_i) = \begin{cases} rmin_{S'}(x_r) & \text{if } y_s \text{ undefined} \\ rmin_{S'}(x_r) + rmin_{S''(y_s)} & \text{otherwise} \end{cases}$$

$$rmin_S(z_i) = \begin{cases} rmax_{S\prime}(x_r) + rmax_{S\prime\prime(y_s)} & \text{if } y_t \text{ undefined} \\ rmax_{S\prime}(x_r) + rmax_{S\prime\prime(y_t)} - 1 & \text{otherwise} \end{cases}$$

Rank update in `COMPRESS`: Assume $\texttt{COMPRESS}(S') = S$, for any element $e \in S$, define $rmin_S(e) = rmin_{S'}(e)$ and $rmax_S(e) = rmax_{S'}(e)$.

5.3.2 Implementation and Resuts

The algorithms are implemented in C++ on an Intel 1.8 GHz Pentium PC with 2GB main memory. The algorithm is tested against a C++ implementation of the algorithm in (Greenwald and Khanna (2001)) (refer to as GK01 in the remaining discussion) from the authors.

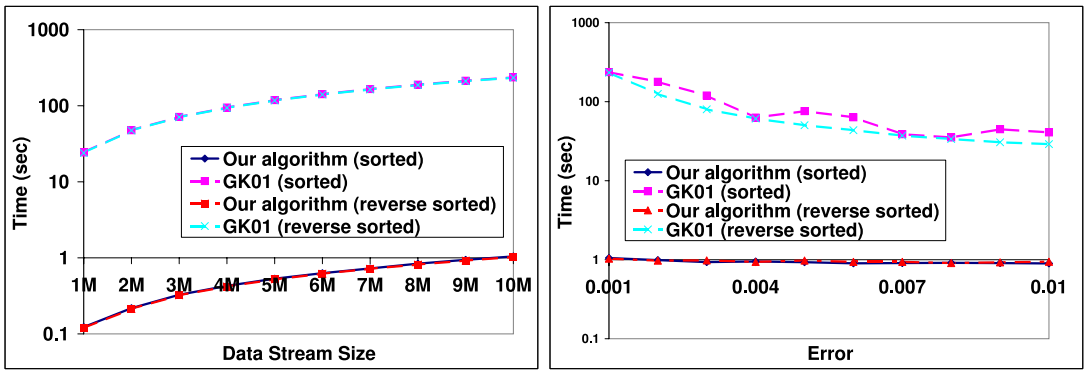
Results

The performance of GK01 and the algorithm are measured on different datasets. Specifically, the computational performance is studied as a function of the size of the incoming stream, the error and input data distribution. In all experiments, the stream size are not known beforehand, and data type is $float$ which takes 4 bytes.

Sorted Input The algorithms are tested using an input stream with either sorted or reverse sorted data. Fig. 5.2(a) shows the performance of GK01 and the algorithm as the input data stream size varies from 10^6 to 10^7 with a guaranteed error bound of 0.001. For these experiments, as the data stream size increases, the block size in the largest sub-stream varies from 191.2K to 270.9K. In practice, the algorithm is able to compute the summary on a stream of size $10^7(40MB)$ using less than 2MB RAM. The algorithm is able to achieve a $200 - 300\times$ speedup over GK01. Note that the sorted and reverse sorted curves for GK01 are almost overlapping due to the log-scale presentation and small difference between them (average 1.16% difference). Same reason for the sorted and reverse sorted curves for the algorithm, and the average difference between them is 2.1%.

The performance of the algorithm and GK01 are also measured by varying the error bound from 10^{-3} to 10^{-2} on sorted and reverse sorted streams. Fig. 5.2(b) shows the performance of the algorithm and GK01 on an input stream of 10^7 data elements. It is observed that the performance of the algorithm is almost constant even when the approximation accuracy of quantiles increases by $10\times$. Note that the performance of GK01 is around $60\times$ slower for large error and around $300\times$ slower for higher precision quantiles compared with the algorithm.

Random Input In order to measure the average case performance, the performance of our algorithm and GK01 are evaluated on random data. Fig. 5.3(a) shows the performance of GK01 and our algorithm as the input data stream size varies from 10^6 to 10^7 with error bound of 0.001. As the data size increases, the time taken by our

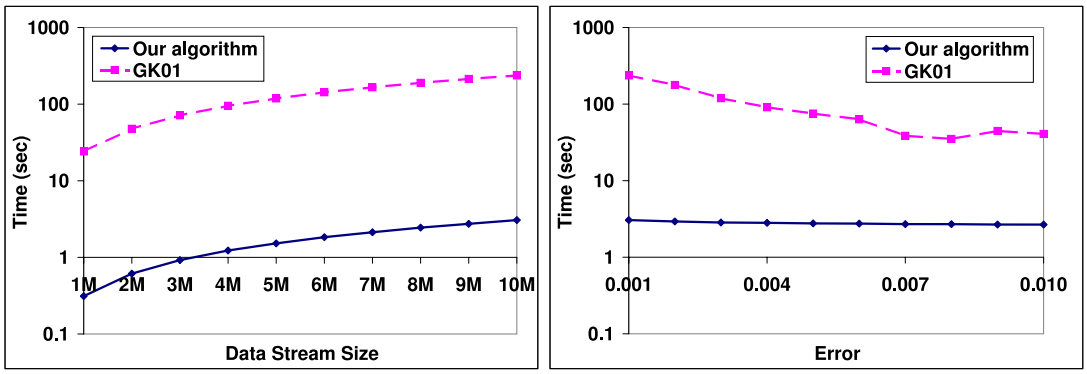


(a) *Summary construction time vs stream size* (b) *Summary construction time vs error*

Figure 5.2: Sorted Data: The sorted and reverse sorted input data are used to measure the best possible performance of the summary construction time using the algorithm and GK01. Fig. 5.2(a) shows the computational time as a function of the stream size on a log-scale for a fixed epsilon of 0.001. It is observed that the sorted and reverse sorted computation time curves for GK01 are almost overlapping due to the log-scale presentation and small difference between them (average 1.16% difference). Same reason for the sorted and reverse sorted curves for the algorithm, and the average difference between them is 2.1%. It is also observed that the performance of the algorithm is almost linear and the computational performance is almost two orders of magnitude faster than GK01. Fig. 5.2(b) shows the computational time as a function of the error. It is observed that the higher performance of the algorithm which is $60-300\times$ faster than GK01. Moreover, GK01 has a significant performance overhead as the error becomes smaller.

algorithm increases almost linearly as the computational requirement of our algorithm is $O(n \log\log n)$. It is observed that our algorithm is able to achieve about $200 - 300\times$ speedup over GK01.

In Fig. 5.3(b), the performance of the algorithms are evaluated on a data stream size of 10^7 by varying the error bound from 10^{-2} to 10^{-3}. It is observed that the performance of our algorithm degrades by less than 10% while computing a summary with $10\times$ higher accuracy. This graph indicates that the performance of our algorithm is sub-linear to the inverse of the error bound. In comparison, the performance of GK01 algorithm degrades by over 500% as the accuracy of the computed summary increases by $10\times$. In practice, the computational time increase for computing a higher accuracy summary



(a) *Summary construction time vs stream* (b) *Summary construction time vs error size*

Figure 5.3: Random Data: The random input data is used to measure the performance of the summary construction time using the algorithm and GK01. Fig. 5.3(a) shows the computational time as a function of the stream size on a log-scale for a fixed epsilon of 0.001. It is observed that the performance of the algorithm is almost linear. Furthermore, the log-scale plot indicates that the algorithm is almost two orders of magnitude faster than GK01. Fig. 5.3(b) shows the computational time as a function of the error. It is observed that the algorithm is almost constant whereas GK01 has a significant performance overhead as the error becomes smaller.

using our algorithm is significantly lower than that using GK01.

Analysis

The worst-case storage requirement for our algorithm is $O(\frac{1}{\epsilon}\log^2(\epsilon N))$. It is comparable to the storage requirement of MRL (Manku et al. (1998)) and higher than GK01. Although the storage requirement is comparatively high, for many practical applications, the storage used by our algorithm is small enough to manage. For example, a stream with 100 million values and error bound 0.001 has a worst-case storage requirement of 5MB and practical on most PCs. Although our algorithm has a higher storage requirement than GK01, our algorithm can construct the summary upto two orders of magnitude faster than GK01. In terms of the computational cost, our algorithm has an expected cost of $O(n\log(\frac{1}{\epsilon}\log(\epsilon N)))$. Therefore, for a fixed error bound, the algorithm has an almost linear increase in computational time in n. Our algorithm also has a

near-logarithmic increase in time as error bound decreases. Therefore, our algorithm is able to handle higher accuracy, large data streams efficiently.

5.4 Approximate Baised-Quantile Computation

5.4.1 Preliminary

The concept of biased quantiles is first proposed in Cormode et al. (2005) to best capture the skewed distribution of the quantiles based on their rank. Biased quantiles can be further classified as low-biased quantiles and high-biased quantiles based on whether the bias is towards the lower ranked quantiles or higher ranked quantiles respectively. Low-biased quantiles can be defined as follows and the higher-biased quantiles can be defined symmetrically Cormode et al. (2005).

Definition 5.4.1. *Let P be a sorted list of n data elements. Let ϕ be a parameter in the range $0 < \phi < 1$. The low-biased quantiles of P are the set of values $P[\lceil \phi^j n \rceil]$ for $j = 1, \ldots, \log_{1/\phi} n$.*

Exact computation of biased quantiles requires linear space. In order to reduce the space requirement to a sublinear function in stream size and to perform computations on streams using a single pass algorithm, Cormode et al. Cormode et al. (2005) defined approximate biased quantiles where the user specifies an error threshold ϵ where $0 \leq \epsilon \leq 1$.

Definition 5.4.2. *The approximate low-biased quantiles of a sorted list P of n elements, is a set of values $\{q_j\}, q_j \in P$, and $j = 1, \ldots, \log_{1/\phi} N$, which satisfy*

$$(1 - \epsilon)\phi^j n \leq r(q_j) \leq (1 + \epsilon)\phi^j n \tag{5.5}$$

where $r(q_j)$ is the rank of q_j in P.

Usually, only the top k biased quantiles are queried.

Considering $\phi^j n$ as the rank in P, the more general problem is for any $r \in \{1, 2, \ldots, n\}$, return an ϵ-approximate biased quantile q which satisfies

$$(1 - \epsilon)r \leq r(q) \leq (1 + \epsilon)r \tag{5.6}$$

Different from the uniform error for approximate uniform quantiles, approximate low-biased quantiles have biased error bound based on their ranks. This notion of biased error guarantees more accuracy for lower-ranked elements.

5.4.2 Algorithms

For stream P, a biased quantile summary is maintained online to answer the approximate biased quantile query at any point of time. This section describes the biased quantile summary structure and property, and the algorithm for computing and maintaining the biased quantile summary for arbitrary sized stream P. The algorithm uses a similar approach to that of Greenwald and Khanna (2004). But the biased quantile problem poses extra difficulty in designing decomposable summary structure and essentially requires siginificantly new approaches and insights.

Biased Quantile Summary

An ϵ-approximate biased quantile summary is a much smaller subset of the original data stream which is able to answer any biased quantile query of rank r over the stream within an error of no more than ϵr. Assume at current time point, n elements have arrived for data stream P. An ϵ-approximate biased quantile summary $Q = \{q_1, \ldots, q_i, \ldots, q_m\}$ for P is an ordered set where $q_i \in P$, $q_1 \leq \ldots \leq q_i \leq \ldots \leq q_m$. For each q_i, two values $rmax_Q(q_i)$ and $rmin_Q(q_i)$ are maintained which represent the maximum and minimum possible ranks of q_i in sorted P. In addition, the smallest and the largest elements in P

are both included in Q, and $rmax_Q(q_1) = rmin_Q(q_1) = 1$, $rmax_Q(q_m) = rmin_Q(q_m) = n$. Finally, Q satisfies the sufficient condition given in the following Lemma which guarantees that Q is able to answer any ϵ-approximate biased quantile query.

Lemma 5.4.1. *Q is able to answer any ϵ-approximate biased quantile query if: $rmax_Q(q_{i+1}) - rmin_Q(q_i) \leq 2\epsilon r(q_i)$, where $1 \leq i < m$*

Proof. Assume summary Q satisfies the sufficient condition. We will prove that for any rank $r \in \{1, 2, \ldots, n\}$, an ϵ-approximate biased quantile q_i can be computed which satisfies $(1-\epsilon)r \leq r(q_i)$ and $r(q_i) \leq (1+\epsilon)r$.

If $n \geq r \geq \frac{n}{1+\epsilon}$, q_m which has rank n in P is the answer, since $n \leq (1+\epsilon)r$ and $n > (1-\epsilon)r$. If $r < \frac{n}{1+\epsilon}$, the ϵ-approximate quantile for r can be obtained by searching in Q for q_j with smallest j such that $rmax(q_j) > (1+\epsilon)r$. The proof is as follows.

According to the choice of q_j, we have

$$rmax_Q(q_j) > (1+\epsilon)r \geq rmax_Q(q_{j-1}) \tag{5.7}$$

According to the sufficient condition, we have

$$rmax_Q(q_j) - rmin_Q(q_{j-1}) \leq 2\epsilon r(q_{j-1}) \tag{5.8}$$

From equation 5.7 and 5.8, we have

$$rmin_Q(q_{j-1}) > (1+\epsilon)r - 2\epsilon r(q_{j-1}) \tag{5.9}$$

Since $r(q_{j-1}) \geq rmin_Q(q_{j-1})$ and using equation 5.9,

$$r(q_{j-1}) > (1+\epsilon)r - 2\epsilon r(q_{j-1})$$

Therefore,

$$r < \frac{1+2\epsilon}{1+\epsilon} r(q_{j-1}) \tag{5.10}$$

Also since $r(q_{j-1}) \leq rmax_Q(q_{j-1}) \leq (1+\epsilon)r$, we have

$$r \geq \frac{r(q_{j-1})}{1+\epsilon} \tag{5.11}$$

Combining equations 5.10 and 5.11, we obtain

$$\frac{r(q_{j-1})}{1+\epsilon} \leq r < \frac{1+2\epsilon}{1+\epsilon} r(q_{j-1}) \tag{5.12}$$

and

$$\frac{\epsilon r(q_{j-1})}{1+\epsilon} \leq \epsilon r < \frac{1+2\epsilon}{1+\epsilon} \epsilon r(q_{j-1}) \tag{5.13}$$

Using equations 5.12 and 5.13, we obtain

$$r - \epsilon r < \frac{1+2\epsilon}{1+\epsilon} r(q_{j-1}) - \frac{\epsilon r(q_{j-1})}{1+\epsilon} = r(q_{j-1}) \tag{5.14}$$

Since $r(q_{j-1}) \leq rmax_Q(q_{j-1}) \leq r + \epsilon r$, we have

$$r - \epsilon r < r(q_{j-1}) \leq r + \epsilon r$$

q_{j-1} is the ϵ-approximate biased quantile for rank r. □

Computing Biased Quantile Summary

In the following discussions, a simple non-uniform sampling technique is presented for computing biased quantile summary for a static dataset. Using this as a building block, our algorithm computes the biased quantile summary online for a stream with known size. The fixed-size stream algorithm is then extended to perform summary computation on arbitrary sized streams and analyze the space and time cost.

Summary Computation for Static Dataset For static dataset S of n elements, a simple non-uniform sampling technique can be used for computing the biased quantile summary.

1. Sort S, for each $s \in S$, set $rmin(s) = rmax(s) = r(s)$, where $r(s)$ is the rank of s in S.

2. For elements of ranks in $[\frac{n}{2}, n]$, sample the elements with sampling rate ϵn, *i.e.*, taking the values of rank $\frac{n}{2}, \frac{n}{2} + \epsilon n, \frac{n}{2} + 2\epsilon n, \ldots, n$.

3. For elements of ranks in $[\frac{n}{4}, \frac{n}{2})$, sample the elements with sampling rate $\frac{\epsilon n}{2}$, *i.e.*, taking the values of rank $\frac{n}{4}, \frac{n}{4} + \frac{\epsilon n}{2}, \frac{n}{4} + 2\frac{\epsilon n}{2}, \ldots$.

4. Repeatedly perform the sampling for elements whose ranks are in range $[\frac{n}{2^i}, \frac{n}{2^{i-1}})$ with sampling rate $\frac{\epsilon n}{2^{i-1}}$, $i = 3, \ldots, \log n$ until at some i, every element in $[\frac{n}{2^i}, \frac{n}{2^{i-1}})$ needs to be sampled.

Lemma 5.4.2. *Let $Q = \{q_1, q_2, \ldots, q_m\}$ $(q_i \leq q_{i+1})$ be the biased quantile summary obtained using the above algorithm, then Q is an ϵ-approximate biased quantile summary for S.*

Proof. We prove that Q satisfies the sufficient condition in Lemma 5.4.1, i.e., for any j, $1 \leq j < m$, $rmax(q_{j+1}) - rmin(q_j) \leq 2\epsilon r(q_j)$.

Consider any $q_j, q_{j+1} \in Q$, there must be an interval $I = [\frac{n}{2^i}, \frac{n}{2^{i-1}})$, where $r(q_j), r(q_{j+1}) \in I$, or $r(q_j)$ is the last element sampled in I and $r(q_{j+1})$ is $\frac{n}{2^{i-1}}$. In any case, we have

$$rmax(q_{j+1}) - rmin(q_j) = r(q_{j+1}) - r(q_j) \leq \frac{\epsilon n}{2^{i-1}} \leq 2\epsilon r(q_j) \tag{5.15}$$

□

Lemma 5.4.3. *The size of Q is $O(\frac{\log n}{\epsilon})$*

Proof. For each interval, at most $\lceil \frac{1}{2\epsilon} \rceil$ elements are sampled. There are no more than $\log n$ intervals. □

Summary Computation for Streams with Known Size As the stream data is continuously arriving, the summary needs to be computed and maintained online. First consider the case that the size of the data stream P is known, say n. At a high-level, the algorithm works as follows. The incoming stream is divided into small blocks of fixed size. For each block, a biased quantile summary is computed using the algorithm in Sec 5.4.2. As blocks come in, an exponential histogram of biased quantile summaries is constructed which covers the entire stream arrived so far. Two operations are used during the online summary computation. The *Merge* and *Prune* operations are introduced on biased quantile summaries as well as their properties in terms of error accumulation. These operations and properties are the main building blocks for the online summary computation algorithms.

Merge Operation The *Merge* operation combines the biased quantile summaries of two streams to generate a new biased quantile summary on the combined stream. Let $Q_1 = \{q_1^1, q_2^1, \ldots, q_{m_1}^1\}$ be an ϵ_1-approximate biased quantile summary of stream P_1, and $Q_2 = \{q_1^2, q_2^2, \ldots, q_{m_2}^2\}$ be an ϵ_2-approximate biased quantile summary of stream P_2. The two quantile summaries Q_1 and Q_2 are merged by merging the elements in Q_1 and Q_2 and updating the $rmax$ and $rmin$ of each element. The update method of $rmax$ and $rmin$ is similar to the method proposed in Greenwald and Khanna (2004) and is given below:

Let $Q = Merge(Q_1, Q_2)$, where $Q = \{q_1, q_2, \ldots, q_{m_1+m_2}\}$. Assume q_i corresponds to q_j^1 in Q_1 (or $q_{j'}^2$ in Q_2 which will be a similar case). Let q_k^2 be the largest element in Q_2 which is smaller than q_j^1, and let q_l^2 be the smallest element in Q_2 which is larger than q_j^1 (if q_j^1 or q_l^2 does not exist then treat them as undefined), then the rank of q_i is updated as follows:

$$rmin_Q(q_i) = \begin{cases} rmin_{Q_1}(q_j^1) & \text{if } q_k^2 \text{ undefined} \\ rmin_{Q_1}(q_j^1) + rmin_{Q_2}(q_k^2) & \text{otherwise} \end{cases}$$

$$rmin_Q(q_i) = \begin{cases} rmax_{Q_1}(q_j^1) + rmax_{Q_2}(q_k^2) & \text{if } q_l^2 \text{ undefined} \\ rmax_{Q_1}(q_j^1) + rmax_{Q_2}(q_l^2) - 1 & \text{otherwise} \end{cases}$$

Greenwald and Khanna Greenwald and Khanna (2004) proved that the *Merge* operation on uniform quantile summaries Q_1 and Q_2 with approximation factors ϵ_1 and ϵ_2 results in a new uniform quantile summary Q with approximation factor $max\{\epsilon_1, \epsilon_2\}$. It can be proved that the same property holds for biased quantile summaries.

Lemma 5.4.4. *Let Q_1 be an ϵ_1-approximate biased quantile summary for P_1, and let Q_2 be an ϵ_2-approximate biased quantile summary for P_2. Then $Merge(Q_1, Q_2)$ produces an ϵ-approximate biased quantile summary Q for $P = P_1 \bigcup P_2$ where $\epsilon = max\{\epsilon_1, \epsilon_2\}$.*

Proof. Based on Lemma 5.4.1, it is sufficient to prove that for any q_i, q_{i+1} in Q, $rmax_Q(q_{i+1}) - rmin_Q(q_i) \leq 2\epsilon(r_Q(q_i))$, where $r_Q(q_i)$ is the rank of q_i in P. First, consider the case when q_i and q_{i+1} are from the same summary, say Q_1. Let q_j^1, q_{j+1}^1 be the corresponding elements of q_i and q_{i+1} in Q_1. Assume q_k^2 be the largest element in Q_2 that is smaller than q_j^1 and let q_l^2 be the smallest element in Q_2 that is larger than q_{j+1}^1. If q_k^2 and q_l^2 are both defined, we have

$$\begin{aligned} & rmax_Q(q_{i+1}) - rmin_Q(q_i) \\ & \leq [rmax_{Q_1}(q_{j+1}^1) + rmax_{Q_2}(q_l^2) - 1] - [rmin_{Q_1}(q_j^1) + rmin_{Q_2}(q_k^2)] \\ & \leq [rmax_{Q_1}(q_{j+1}^1) - rmin_{Q_1}(q_j^1)] + [rmax_{Q_2}(q_l^2) - rmin_{Q_2}(q_k^2) - 1] \\ & \leq 2\epsilon_1 r_{Q_1}(q_j^1) + 2\epsilon_2 r_{Q_2}(q_k^2) - 1 \\ & \leq 2\epsilon r_Q(q_i) \end{aligned}$$

Similar analysis can be applied for the case where q_k^2 or q_l^2 are undefined.

If q_i and q_{i+1} come from different summaries, assume q_i corresponds to q_j^1 in Q_1 and q_{i+1} corresponds to q_l^2 in Q_2. In addition, q_{j+1}^1 is the smallest element in Q_1 that is larger than q_l^2, and q_{l-1}^2 is the largest element in Q_2 that is smaller than q_j^1. If both q_{j+1}^1 and q_{l-1}^2 are defined, we have the following derivation:

$$
\begin{aligned}
& rmax_Q(q_{i+1}) - rmin_Q(q_i) \\
& \leq [rmax_{Q_2}(q_l^2) + rmax_{Q_1}(q_{j+1}^1) - 1] - [rmin_{Q_1}(q_j^1) + rmin_{Q_2}(q_{l-1}^2)] \\
& \leq [rmax_{Q_2}(q_l^2) - rmin_{Q_2}(q_{l-1}^2)] + [rmax_{Q_1}(q_{j+1}^1) - rmin_{Q_1}(q_j^1) - 1] \\
& \leq 2\epsilon_2 r_{Q_2}(q_{l-1}^2) + 2\epsilon_1 r_{Q_1}(q_j^1) - 1 \\
& \leq 2\epsilon r_Q(q_i)
\end{aligned}
$$

Similar analysis can be applied for the case where q_{j+1}^1 and q_{l-1}^2 are undefined. □

Prune Operation The pruning operation takes as input an ϵ'-approximate quantile summary Q' and a parameter B, and returns a new summary Q such that Q is an $(\epsilon' + 3/B)$-approximate quantile summary for P.

First consider the case when $\epsilon' > 0$. We partition all possible ranks $[1, n]$ into $\log(\epsilon' n)$ partitions: $[1, 1/\epsilon')$, $[1/\epsilon', 2/\epsilon')$,..., $[\frac{2^{i-1}}{\epsilon'}, \frac{2^i}{\epsilon'})$,..., $[\frac{2^{\log(\epsilon' n)-1}}{\epsilon'}, n]$. For each interval, we query for B quantiles. For instance, in interval i, ranks are between $[\frac{2^{i-1}}{\epsilon'}, \frac{2^i}{\epsilon'})$, and we query Q' for quantiles of ranks $\frac{2^{i-1}}{\epsilon'} + \frac{j2^{i-1}}{\epsilon' B}$, $j = 0, \ldots, B-1$. If the interval size is less than B, then we simply sample all the ranks in that interval. We also make sure that rank n is queried. For each element $q \in Q$, we set $rmin_Q(q) = rmin_{Q'}(q)$ and $rmax_Q(q) = rmax_{Q'}(q)$.

Assume that q_j, q_{j+1} are quantiles obtained by querying two consecutive ranks r_1, r_2 from Q'. According to the pruning process, we know that $r_2 - r_1 = \frac{2^{i-1}}{\epsilon' B}$ for some i. W.l.o.g. assume that $r_1 = \frac{2^{i-1}}{\epsilon'} + \frac{l2^{i-1}}{\epsilon' B}$ and $r_2 = \frac{2^{i-1}}{\epsilon'} + \frac{(l+1)2^{i-1}}{\epsilon' B}$ where $l \in [0, \ldots, B-1]$.

According to Lemma 5.4.1, we know that

$$rmin_Q(q_j) = rmin_{Q'}(q_j) \geq rmax_{Q'}(q'_{j+1}) - 2\epsilon' r(q_j) \tag{5.16}$$

where q'_{j+1} is the element next to q_j in Q'.

According to the choice of q_j, we have

$$rmax_{Q'}(q'_{j+1}) > (1+\epsilon')r_1 \tag{5.17}$$

From Equation 5.16 and 5.17, we have

$$rmin_Q(q_j) > (1+\epsilon')r_1 - 2\epsilon' r(q_j) \tag{5.18}$$

Since $rmax_Q(q_{j+1}) = rmax_{Q'}(q_{j+1}) \leq (1+\epsilon')r_2$, from Equation 5.18 we have

$$rmax_Q(q_{j+1}) - rmin_Q(q_j) < (1+\epsilon')(r_2 - r_1) + 2\epsilon' r(q_j) \tag{5.19}$$

Since $r(q_j) \geq rmin_Q(q_j)$, from Equation 5.18, we have

$$r_1 < \frac{1+2\epsilon'}{1+\epsilon'} r(q_j) \tag{5.20}$$

Since $r_1 = \frac{2^{i-1}}{\epsilon'} + \frac{l2^{i-1}}{\epsilon' B}$, from Equation 5.20 we have

$$r_2 - r_1 = \frac{2^{i-1}}{\epsilon' B} < \frac{1+2\epsilon'}{1+\epsilon'} \frac{r(q_j)}{B+l} < \frac{1+2\epsilon'}{B} r(q_j) < \frac{3}{B} r(q_j) \tag{5.21}$$

From Equation 5.19 and 5.21 we have

$$\begin{aligned} &rmax_Q(q_{j+1}) - rmin_Q(q_j) \\ &< \frac{3(1+\epsilon')}{B} r(q_j) + 2\epsilon' r(q_j) < 2(\epsilon' + \frac{3}{B}) r(q_j) \end{aligned}$$

Therefore, the summary Q obtained after *Prune* operation is also a biased quantile summary (Lemma 5.4.1), and with an additional approximation factor of $\frac{3}{B}$.

Now let us consider the case when ϵ' is 0. This is a special case where *Prune* operation described above does not apply. Instead the algorithm in Sec 5.4.2 is used to compute the initial biased quantile summary with error.

Online Computation of Biased Quantile Summary The stream P is divided into blocks $\{Blk_0, Blk_1, \ldots, Blk_{\lceil \log \frac{n}{b} \rceil}\}$ of size $b = \frac{3 \log^2(\epsilon n)}{\epsilon}$, where Blk_0 denotes the most recent block (could be incomplete), n is the size of the stream which is known *a priori*, and ϵ is the desired error bound. For each block, the local biased quantile summary is comptued using the algorithm in Sec 5.4.2. As blocks come in, the exponential histogram of biased quantile summaries is computed $Summary\_EH = \{Q_0, \ldots, Q_l, \ldots, Q_L\}$, where Q_0 is Blk_0, Q_1 is the summary which covers Blk_1, Q_2 is the summary which covers $Blk_2 \bigcup Blk_3$, Q_3 is the summary which covers $Blk_4 \bigcup Blk_5 \bigcup Blk_6 \bigcup Blk_7$, and Q_L is the summary which covers the oldest 2^L blocks. The detailed update procedure whenever a new element comes in is described below:

1. Insert the element into Q_0.

2. If Q_0 is not full ($|Q_0| < b$), stop and the update procedure is done for the current element. If Q_0 becomes full ($|Q_0| = b$), the biased quantile summary Q of Q_0 is computed using the algorithm in Sec 5.4.2. Send Q to level 1, empty Q_0.

3. If Q_1 is empty, Q_1 is set to be Q and the update procedure is done. Otherwise, the operations $Q_1 = Merge(Q_1, Q)$, $Q = Prune(Q_1)$, $Empty(Q_1)$ are performed in the given order, and send new Q to level 2

4. Repeatedly perform step 3 for $Q_i, i = 2, \ldots, L$ until a level L is reached where Q_L is empty.

The pseudo code of the entire update procedure whenever an element e comes is shown in Algorithm 1.

Algorithm 5.3 bQUpdate(e,$Summary\_EH$,ϵ)
Input e: current data element to be inserted, $Summary\_EH$: current exponential histogram of biased quantile summaries: $Summary\_EH = \{Q_0, \ldots, Q_l, \ldots, Q_L\}$, ϵ: required approximation factor

```
1: Insert e into Q_0
2: if |Q_0| = b(b = 3 log^2(εn)/ε) then
3:    Compute the biased quantile Q of Q_0 using the algorithm in Sec 5.4.2, with
      approximation factor ε/log(εn)
4:    empty(Q_0)
5: else
6:    exit
7: end if
8: for l = 1 to L do
9:    if |Q_l| = 0 then
10:      Q_l ← Q
11:      break
12:   else
13:      Q = Merge(Q_l, Q)
14:      Q = Prune(Q), with B = 3 log(εn)/ε
15:      empty(s_l)
16:   end if
17: end for
```

Lemma 5.4.5. *The number of levels L in $Summary\_EH$ is less than* $\log(\epsilon n)$

Proof. There are $\frac{n}{b}$ blocks at level 0, therefore, $L \leq \log(\frac{n}{b}) \leq \log \frac{n}{\frac{3\log(\epsilon n)}{\epsilon}} < \log(\epsilon n)$ □

Lemma 5.4.6. *Consider $Summary\_EH = \{Q_0, \ldots, Q_l, \ldots, Q_L\}$, then $Merge(Q_1, \ldots, Q_l, \ldots, Q_L)$ is an ϵ-approximate biased quantile summary for the entire stream.*

Proof. First prove that Q_l is an ϵ-approximate quantile summary for the blocks covered by Q_l. Q_1 is an $\epsilon/\log(\epsilon n)$-approximate summary. The *Prune* operation at each level increases the error by $\frac{3}{B} = \epsilon/\log(\epsilon n)$. Therefore, Q_l is an $l\epsilon/\log(\epsilon n)$-approximate summary. Since $l \leq L < \log(\epsilon n)$, Q_l is an ϵ-approximate summary. Secondly, all the Q_ls cover the entire stream. According to $Merge$ operation, $Merge(Q_1, \ldots, Q_l, \ldots, Q_L)$ is an ϵ-approximate biased quantile summary for the entire stream. □

Summary Computation for Stream with Unknown Size The algorithm described in the last section can be extended to compute approximate quantiles in streams

without prior knowledge of the size. At a high-level, the input stream P is partitioned into disjoint sub-streams $P_0, P_1, \ldots, P_m$ with increasing size. Specifically, sub-stream P_i has size $\frac{2^i}{\epsilon}$ and covers the elements whose location is in the interval $[\frac{2^i-1}{\epsilon}, \frac{2^{i+1}-1}{\epsilon})$. The biased quantile summary is comptued for each sub-stream with known size using Algorithm 1, and then $Merge$ operation is applied to obtain the summary for the entire stream. The summary construction algorithm is as follows.

1. For the latest sub-stream P_k whose summary is currently being constructed, an ϵ'-exponential histogram of summaries $Summary\_EH$ is computed using Algorithm 1 by performing $bQUpdate\ (e, Summary\_EH, \epsilon')$ whenever an element e comes. Here $\epsilon' = \frac{\epsilon}{2}$.

2. Once the last element of sub-stream P_k arrives, an $\frac{\epsilon}{2}$-summary is computed on $Merge(Q_1, \ldots, Q_l, \ldots, Q_L)$ by a $Prune$ operation with $B = \frac{6}{\epsilon}$. The resulting summary $\overline{Q}_k = Prune(Merge(Q_1, \ldots, Q_l, \ldots, Q_L), \frac{\epsilon}{2})$ is an ϵ-summary of P_k.

3. The ordered set of the summaries of all complete sub-streams so far $\overline{Q} = \{\overline{Q}_0, \overline{Q}_1, \ldots, \overline{Q}_{k-1}\}$ is the current multi-level ϵ-summary of the entire stream except the most recent incomplete sub-stream P_k.

The pseudo code for the update algorithm for stream with unknown size is shown in Algorithm 2. Initially, $\overline{Q} = \phi$. Whenever an element comes, $gbQUpdate$ is performed to update the summary structure $\overline{Q}$.

To answer a query of any rank r using $\overline{Q}$, if $Summary\_EH$ is not empty, first compute $\overline{Q}_k$ for the incomplete sub-stream P_k: $\overline{Q}_k = Prune(Merge(Summary\_EH))$, then merge all the ϵ-summaries $\overline{Q}_0, \overline{Q}_1, \ldots, \overline{Q}_{k-1}$ in $\overline{Q}$ together with $\overline{Q}_k$ using $Merge$ operation, the final summary is the ϵ-summary for entire P.

Analysis

The summary structure maintains $\log(\epsilon n)$ sub-stream summaries. Each sub-stream except the last one maintains at most $O(\frac{\log(\epsilon n)}{\epsilon})$ elements, totally it takes $\frac{\log^2(\epsilon n)}{\epsilon}$. The last

Algorithm 5.4 gbQUpdate(e,$\overline{Q}, \epsilon, Summary\_EH$)
Input e: current data element, $\overline{Q}$: current summary structure, $\overline{Q} = \{\overline{Q}_0, \overline{Q}_1, \ldots, \overline{Q}_{k-1}\}$ (sub-streams $P_0, \ldots, P_{k-1}$ have completely arrived), ϵ: required approximation factor of $\overline{Q}$, $Summary\_EH$: the fixed size multi-level summary corresponding to the current sub-stream P_k, $Summary\_EH = \{Q_0, Q_1, \ldots, Q_L\}$

1: **if** e is the last element of P_k **then**
2: Apply $Merge$ on all the Q_l in $Summary\_EH$: $Q_{all} = merge(Summary\_EH) = Merge(Q_0, Q_1, \ldots, Q_L)$
3: $\overline{Q}_k = compress(Q_{all}, \frac{\epsilon}{2})$
4: $\overline{Q} = \overline{Q} \bigcup \{\overline{Q}_k\}$
5: $Summary\_EH \leftarrow \phi$
6: **else**
7: update $Summary_E H$: $Summary_E H = bQUpdate(e, Summary\_EH, \frac{\epsilon}{2})$
8: **end if**

stream maintains an exponential histogram of summaries which has a space requirement of $O(\frac{\log^3(\epsilon n)}{\epsilon})$. Therefore, the algorithm has a space requirement of $O(\frac{\log^3(\epsilon n)}{\epsilon})$.

Theorem 5.4.7. *The amortized update time of the summary at each node is* $O(\log(\frac{1}{\epsilon}\log(\epsilon n))$

Proof. First consider at a single sub-stream, P_i of size n_i. At level 0, for each block, sorting is performed which costs $b \log b$ per block, where $b = 3^{\frac{\log^2(\epsilon' n_i)}{\epsilon'}}$. Also for each block, the algorithm in Sec 5.4.2 is performed which requires a linear scan of cost b. There are n_i/b blocks. Totally at level 0, the cost is bounded by $O(n_i \log \frac{\log(\epsilon n_i)}{\epsilon})$. At level $l > 0$, there are $\frac{n_i}{2^{l-1}b}$ blocks. For every two blocks, a merge and a pruning operation are performed, each of which takes linear time to scan the two block summaries which have size of $O(b)$. Therefore, the cost for all levels except level 0 is

$$\sum_{l=1}^{\log(n_i/b)} \frac{n_i}{2^l b} 2b = \sum_{l=1}^{\log(n_i/b)} \frac{n_i}{2^{l-1}} \tag{5.22}$$

which is $O(n_i)$

Therefore, the total cost for computing the summary for the sub-stream is $O(n_i \log \frac{\log(\epsilon' n_i)}{\epsilon'})$.

After each sub-stream is complete, an additional *Merge* and *Prune* operation are performed each of cost $O(\frac{1}{\epsilon'}\log^2(\epsilon' n_i))$. Given the above observations, the total compu-

tational cost of the algorithm is

$$\sum_{i=0}^{i=\lfloor \log(\epsilon n+1)\rfloor} (\frac{2^i}{\epsilon}\log(\frac{2(i-1)}{\epsilon}) + \frac{2}{\epsilon}(i-1)^2) \tag{5.23}$$

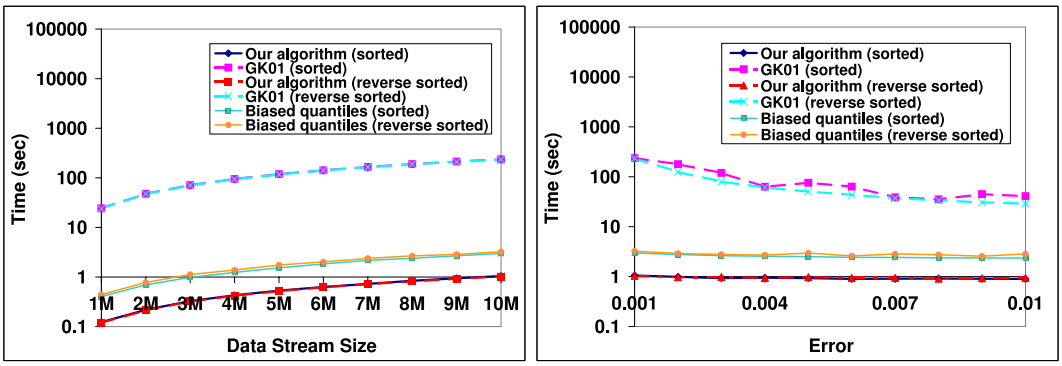
Simplifying equation the above equation, the total computational cost of the algorithm is $O(n\log(\frac{1}{\epsilon}\log(\epsilon n)))$, the average updating time per element is $O(\log(\frac{1}{\epsilon}\log(\epsilon n)))$, which is $O(\log\log n)$ if ϵ is fixed. □

Distributed Streams

The algorithm is applicable to compute approximate biased quantile summaries in distributed networks. In Greenwald and Khanna (2004), Greenwald and Khanna proposed merge and prune operations to maintain decomposable summaries in sensor networks. Although the merge and prune operations are more involved than the ones described in Greenwald and Khanna (2004), their overall algorithm for maintaining approximate uniform quantile summaries in sensor networks can be directly extended to biased quantiles by replacing their operations with the operations. In addition, the algorithm can also take advantage of the transmission reduction optimizations presented in Greenwald and Khanna (2004).

5.4.3 Implementation and Results

The algorithm is implemented in C++ on a Windows XP PC with 1.8GHz Intel Pentium processor and 2GB RAM. The algorithm is compared with two uniform quantile implementations obtained from the authors of GK01 Greenwald and Khanna (2001) and ZW07 Zhang and Wang (2007) respectively. Both of these implementations are general and do not make assumptions on stream size or on the range of data distribution. In particular, ZW07 uses similar optimizations as our approach to achieve high performance in uniform quantiles. Although ZW07 and GK01 only compute uniform quantile, their



(a) *Summary construction time vs stream* (b) *Summary construction time vs error size*

Figure 5.4: Sorted Data: The sorted and reverse sorted input data are used to measure the best possible performance of the summary construction time using our biased quantile algorithm and uniform quantile algorithms ZW07, GK01 using the same error 0.001. This log-scale plot indicates that our algorithm achieves up to 75x higher performance compared to GK01 and comparable performance to ZW07. Fig. 5.4(b) indicates the performance of the our biased quantile algorithm and the uniform quantile algorithms on a 10M stream size.

performance at the same error can provide an upper bound on the performance for related biased quantile algorithms such as Cormode et al. (2005) which uses the similar technique as GK01.

Results

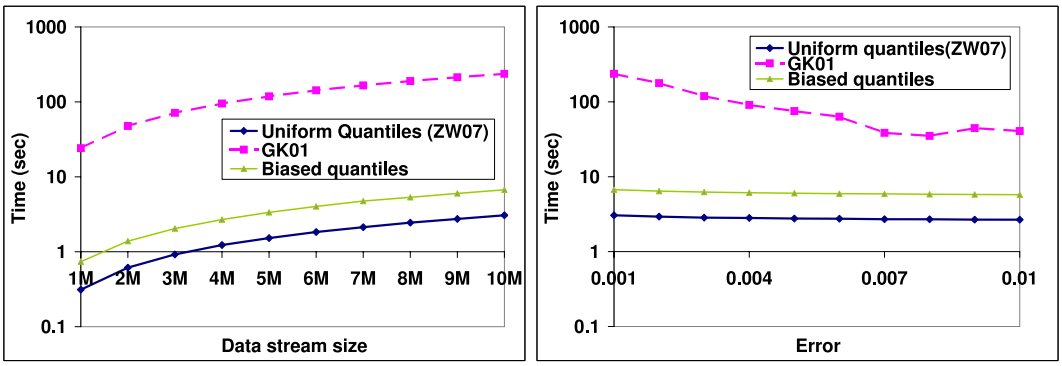
The most time consuming portion of our algorithm is the sorting of the blocks at level 0. As the sort performance is dependent on the order of input data, the algorithms are tested on both sorted input data and random input data. The performance are measured by varying the error and the size of the incoming data stream. In the experiments, the prior knowledge of stream length is not required. 32-bit floating point is used as data type for the input stream to ensure that the data distribution range is large.

Sorted Input The performance of our algorithm, ZW07 and GK01 are compared using sorted and reverse sorted data. The performance of the merge and prune operations remain nearly the same since the input data to both merge and prune operations

is sorted. Therefore, sorted nature of input data does not affect their performance. However, the performance of sort routine on the block at level 0 is mainly dependent on the input data order. Fig. 5.4(a) demonstrates the performance of our biased quantile algorithm as a function of sorted stream data size using an error of 0.001. Observe that our biased quantile algorithm is comparable to the performance of an optimized uniform quantile computation algorithm (ZW07) at the same error and is around 55-75x faster than GK01 at the same error. Moreover, our algorithm is able to compute over 3.2M quantiles per second on large streams consisting of tens of millions of elements.

Fig. 5.4(b) highlights the performance of our algorithm on a stream with ten million elements as the error increases from 0.001 to 0.01 using a log-scale plot. As the error increases from 0.001 to 0.01, the performance of the algorithm improves from 3.2M to 4M quantiles per second. Note that the performance of GK01 improves as error increases. In general, the performance of biased quantile routines are expected to be slower than uniform quantile algorithms for the same error since biased quantile summaries are larger than uniform quantile summaries. The performance of our biased quantile algorithm is 16-75x faster than GK01 at same error and is comparable to ZW07.

Random Input The performance of the algorithm is also measured using random input data by varying the input data size and the error. Fig. 5.5(a) shows the performance of our algorithm as a function of stream size using an error of 0.001. In comparison to the performance of our algorithm on sorted data, our algorithm on random data is around 2x slower. This is mainly due to the slower performance of the sort routine on random data than sorted data. In practice, our algorithm is able to achieve around 1.4M quantiles per second on random data of size 10M with error 0.001. In terms of the space requirement, our algorithm uses 1.6MB to 4.4MB to construct the summary as the stream size varies from 1M to 10M elements. Fig. 5.5(b) shows the performance of our algorithm as a function of error. The error is varied from 0.001 to 0.01 on an input stream with 10M elements. As the error increases, the storage requirement for our



(a) *Summary construction time vs stream* (b) *Summary construction time vs error size*

Figure 5.5: Random Data: The performance of the summary construction time using our biased quantile algorithm and GK01 over random data. Fig. 5.5(a) shows the computational time as a function of the stream size on a log-scale for a fixed epsilon of 0.001. It is observed that our algorithm is able to compute 1.4-1.6M quantiles per second. In practice, our algorithm is over 30x faster than prior biased quantile algorithms.

algorithm reduces from 4.4MB to 1.5MB. Moreover, the performance of our algorithm improves from 1.4-1.6M quantiles per second as error increases from 0.001 to 0.01.

Table 5.2 shows the performance comparison of our algorithm with prior uniform quantile and biased quantile algorithms. In practice, our algorithm achieves significant performance improvement over prior biased quantile algorithms and comparable performance to uniform quantile algorithms.

Analysis

Our algorithm extends the framework developed for uniform quantiles to biased quantiles with poly-log storage requirement. The computational cost to maintain the biased quantile summary is proportion to the error and stream size. Overall, it has an average computational cost of $O(\frac{\log\log \epsilon n}{\epsilon})$ per element. The generality and high performance of our algorithm results in a tradeoff in space requirement as compared to prior biased quantile algorithms. Therefore, our algorithm is more suited for applications with distributed, high-speed data streams.

Algorithm	General	Additive	Performance
Our BQ	yes	yes	1.4M qps
CKMS05(BQ)	yes	no	0.1M qps*
CKMS06(BQ)	no	yes	1M qps*
ZW07(UQ)	yes	yes	3M qps
GK01/GK04UQ	yes	yes	0.04M qps (GK01)

Table 5.2: This table shows the properties of the different biased quantile (BQ) and uniform quantile (UQ) algorithms. All the algorithms do not make assumptions on stream sizes or input data ranges except CKMS06 which requires knowledge of input data ranges. Moreover, all the biased quantile algorithms except CKMS05 specify pruning operations to add error on existing summaries and can be applied to sensor networks. Our biased quantile algorithm is both general and applicable to sensor networks. Moreover, it achieves higher performance in terms of quantiles per second (qps) than prior BQ algorithms running on similar hardware. * Performance numbers of CKMS05 and CKMS06 are obtained from [3,4].

Chapter 6

Conclusions

In this thesis, I have designed efficient algorithms for mining two types of emerging massive sequence-based scientific datasets: static sequences from genomic datasets for biological research and dynamic sequences from streaming and sensor datasets for environmental and ecological research. The algorithms utilize block-wise decomposition to "divide" the long sequences into blocks to "conquer" the challenges posed by the large-scale, noisy, or distributive, online sequence-based datasets. I have applied these algorithms on real and synthetic datasets to demonstrate the effectiveness and efficiency of the algorithms. I have also compared the performance of some of the algorithms to prior state-of-the-art algorithms. In some cases, the approaches speed up the process substantially.

In the following sections, I summarize the algorithms and describe avenues for future investigation.

6.1 Mining Genomic Datasets

Genome-wide single-nucleotide polymorphism (SNP) arrays provide a comprehensive view of genome variation and serve as powerful resources for genetic and biomedical studies. In this thesis, I studied the problem of inferring the genetic variation patterns of recombinant strains using the SNP panels obtained from the strains. This problem

plays an important role in solving many genetics problems such as reconstructing the genealogy and gene association studies. Particularly, I investigated two problems as summarized below.

6.1.1 Inferring Segmentation Structure of Recombinant Genotype Sequences

Summary

I studied the problem of inferring the fragmental structure of recombinant strains given a set of founder sequences. This is a critical problem of understanding the ancestral structure of strains from experimental genetic resources, which are usually derived through generations of breeding starting with a set of isogeneic founder animals. Particularly, solving this problem can help analyzing the genome variation structure exhibited in Pre-CC strains in the Collaborative Cross project.

I proposed the Minimum Segmentation model, which captures the set of the minimum number of segments for a recombinant strain given a set of founder haplotype sequences, where each segment is attributable to one of the founders. I consider segmenting genotype sequences instead of haplotype sequences. Genotype sequence is more difficult to handle algorithmically due to the heterozygous sites but less expensive to obtain experimentally. I proposed block-based dynamic programming algorithms which run in polynomial time. The algorithms can effectively handle the biologically relevant constraints as well as the noise in the data such as genotyping errors, point mutations, missing values, etc. The experimental results on real and synthetic data demonstrate good performance of the algorithms.

Future Work

Many research opportunities lie ahead. There may be multiple minimum segmentation solutions to the same piece of data. It is highly desirable to summarize the patterns and compute the set of representative solutions. In addition, combinatorial optimization solutions are subject to the limitations of incapability of providing probabilistic analysis. Projects are also underway for deriving the probabilistic inference of the ancestral structure of the strains.

6.1.2 Inferring Genome-wide Mosaic Structure

Summary

Meiotic recombination events during the evolutionary process result in a mosaic structure over the genome. Inferring such a fine-scale mosaic structure is important for many genetics problems such as gene association studies. I proposed the Minimum Mosaic model that captures a minimum number of breakpoints to generate the haplotypes within extant populations. The resulting blocks are compatible where no recombinations can be inferred within a block according to the FGT. I proposed a novel graph algorithm constructed on blocks of sequences to compute the minimum mosaic structure of genomes. The experimental results on real NIESH mouse strains datasets demonstrate that the efficiency of the algorithm on genome-wide analysis.

Future Work

There are many promising directions for further investigation. Genome-wide mosaic structure describes the genetic variation resulting from the meiotic recombination events. It is also important to consider the reverse problem, namely, reconstructing the revolutionary history and the founder set given the Minimum Mosaic. The reconstructed revolutionary history can provide an estimation of the upper bound on the number of

recombination events given the inferred set of founder sequences. The problem can be optimized for either the minimum number of recombination events or the minimum number of founder sequences. Besides the recombination considered in this thesis, gene-conversion is an important and more common form of recombination. It would be interesting to develop more robust models which also incorporates gene conversion in the mosaic inference.

6.2 Mining Streaming and Sensor Network Environmental Datasets

Resent hardware advances enabled the collection of enormous scientific observations in the form of streaming or sensor network datasets. In this thesis, I focus on an important problem of analyzing such large-scale, online, or distrusted datasets. Particularly, I studied clustering and Order-statistics computation over data streams, which are important analysis for capturing data distribution of datasets.

6.2.1 Clustering Distributed Data Streams

Summary

Clustering over distributed data streams faces many challenges such as limited battery, distributed processing, and online processing, etc. Efficient resource-aware algorithms which provide approximation solution with bounded error are highly desirable for these applications. I present algorithms for approximate k-median clustering over distributed data streams in three different routing topology settings: topology-oblivious, height-aware, and path-aware. The algorithms maintain efficient summary structure at each node by dividing the incoming streams into blocks and updating the summary using effecient compress and merge operators. The algorithms reduce the max per node data

transmission to $polylog(N)$. The topology oblivious algorithm runs without any prior knowledge of the routing topology and the distribution of the stream data. The performance of the topology oblivious algorithm can be further improved if the height of the routing tree is given(height-aware algorithm), or if the routing path of each node to the root is known (path-aware algorithm). In practice, the methods significantly reduce the data transmission requirements on both synthetic and real datasets to a small fraction of the overall volume of the streams and are also well below the theoretical bounds.

Future Work

There are many promising avenues for future work. It is desirable to improve the algorithms with better storage bound. The algorithms can also be extended to perform sliding window computations in sensor network model. Furthermore, the techniques proposed can be used for computing higher order primitives for spatial computations, as well as other distributed mining applications.

6.2.2 Fast Algorithms for Approximate Order-Statistics Computation in Data Streams

Summary

I presented fast algorithms for computing approximate quantiles and biased-quantiles for streams.

The algorithms for approximate quantile computation are based on simple block-wise merge and sort operations which significantly reduces the update cost performed for each incoming element in stream. In order to handle unknown size of the stream, the incoming streams are divided into sub-streams of exponentially increasing sizes. Summaries are constructed efficiently using limited space on the sub-streams. For both fixed sized and arbitrary sized streams, the algorithm has an average update time complexity of

$O(\log \frac{1}{\epsilon} \log \epsilon N)$. I also analyzed the performance of prior algorithms. I evaluated the algorithms on different data sizes and compared them with optimal implementations of prior algorithms. In practice, the algorithm can achieve up to 300× improvement in performance. Moreover, the algorithm exhibits almost linear performance with respect to stream size and performs well on large data streams.

In addition, I presented a novel approximate biased quantile algorithm for handling large, high-speed data streams. The algorithm maintains a decomposable summary structure to deterministically answer approximate biased quantile queries. Efficient sampling and merge operations are used to maintain the summary structure. In practice, the algorithm requires poly-log space to maintain the summary and has an update cost of $\log \log n$ where n is the current stream size. The algorithm is also applicable to sensor networks and is able to achieve significant performance improvement over prior algorithms.

Future Work

There are many interesting problems for future investigation. The algorithms can be extended for fast computation of quantiles and biased quantiles over sliding windows. It is also interesting to design generalized framework for incremental streaming computation using the techniques of block-wise merge and compression algorithms, for either single data stream or distributed data streams.

Bibliography

Agarwal, P. K., Har-Peled, S., and Varadarajan, K. R. (2005). Geometric approximation via coresets. 66

Aggarwal, C. (2005). On abnormality detection in spuriously populated data streams. In *Proceedings of ACM SIAM Conference on Data Mining*. 58

Aggarwal, C. C., Han, J., Wang, J., and Yu, P. S. (2003). A framework for clustering evolving data streams. In *Proceedings of 29th VLDB Conference*. 58, 61

Aggarwal, C. C., Han, J., Wang, J., and Yu, P. S. (2004). A framework for projected clustering of high dimensional data streams. In *Proceedings of 30th VLDB Conference*. 61

Agrawal, R. and Swami, A. (1995). A one-pass space-efficient algorithm for finding quantiles. 85

Ahmad, Y. and etintemel, U. (2004). Network-aware query processing for stream-based applications. In *Proceedings of VLDB*. 58

Arasu, A. and Manku, G. S. (2004). Approximate counts and quantiles over sliding windows. In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 286–296. 14, 83, 84, 85

Babcock, B. and Olston, C. (2003). Distributed top-k monitoring. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 58

Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., and Zdonik, S. (2003). Scalable distributed stream processing. In *Proceedings of First Biennial Conference on Innovative Data Systems Research(CIDR 2003)*. 58

Churchill, G. A., Airey, D. C., Allayee, H., Angel, J. M., Attie, A. D., Beatty, J., Beavis, W. D., Belknap, J. K., Bennett, B., Berrettini, W., Bleich, A., Bogue, M., Broman, K. W., Buck, K. J., Buckler, E., Burmeister, M., Chesler, E. J., Cheverud, J. M., Clapcote, S., Cook, M. N., Cox, R. D., Crabbe, J. C., Crusio, W. E., Darvasi, A., Deschepper, C. F., Doerge, R. W., Farber, C. R., Forejt, J., Gaile, D., Garlow, S. J., Geiger, H., Gershenfeld, H., Gordon, T., Gu, J., Gu, W., d. H. G., Hayes, N. L., Heller, C., Himmelbauer, H., Hitzemann, R., Hunter, K., Hsu, H. C., Iraqi, F. A., Ivandic, B., Jacob, H. J., Jansen, R. C., Jepsen, K. J., Johnson, D. K., Johnson, T. E., Kempermann, G., Kendziorski, C., Kotb, M., Kooy, R. F., Llamas, B., Lammert, F., Lassalle, J. M., Lowenstein, P. R., Lu, L., Lusis, A., Manly, K. F., Marcucio, R., Matthews, D., Medrano, J. F., Miller, D. R., Mittleman, G., Mock, B. A., Mogil, J. S., Montagutelli, X., Morahan, G., Morris, D. G., Mott, R., Nadeau, J. H., Nagase, H., Nowakowski, R. S., O'Hara, B. F., Osadchuk, A. V., Page, G. P., Paigen, B.,

Paigen, K., Palmer, A. A., Pan, H. J., Peltonen-Palotie, L., Peirce, J., Pomp, D., Pravenec, M., Prows, D. R., Qi, Z., Reeves, R. H., Roder, J., Rosen, G. D., Schadt, E. E., Schalkwyk, L. C., Seltzer, Z., Shimomura, K., Shou, S., Sillanp, M. J., Siracusa, L. D., Snoeck, H. W., Spearow, J. L., Svenson, K., Tarantino, L. M., Threadgill, D., Toth, L. A., Valdar, W., de Villena, F. P., Warden, C., Whatley, S., Williams, R. W., Wiltshire, T., Yi, N., Zhang, D., Zhang, M., Zou, F., and Consortium., C. T. (2004). High-resolution haplotype structure in the human genome. *Nat. Genet.*, pages 1133–1137. 20

Considine, J., Li, F., Kollios, G., and Byers, J. (2004). Approximate aggregation techniques for sensor databases. In *Proceedings of the International Conference on Data Engineering (ICDE04)*. 61

Cormode, G., Korn, F., Muthukrishnan, S., and Srivastava, D. (2005). Effective computation of biased quantiles over data streams. In *Proc. of International Conference on Data Engineering*, pages 20–32. 15, 58, 83, 84, 85, 99, 113

Cormode, G., Korn, F., Muthukrishnan, S., and Srivastava, D. (2006). Space- and time-efficient deterministic algorithms for biased quantiles over data streams. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 263–272. 15, 83, 84, 85

Dally, M., Rioux, J., Schaffner, S., Hudson, T., and Lander, E. (2001). High-resolution haplotype structure in the human genome. *Nat. Genet.*, pages 229–232. 7, 22

Drouin, G., Prat, F., Ell, M., and Clark, G. (1999). Detecting and characterizing gene conversions between multigene family members. *Mol. Biol. Evol*, pages 1369–1390. 9, 44

Forman, G. and Zhang, B. (2001). Distributed data clustering can be efficient and exact. In *ACM KDD Explorations special issue on Scalable Data Mining Algorithms*. 59, 60

Frahling, G. and Sohler, C. (2005). Coresets in dynamic geometric data streams. In *Proc. 37th ACM Symposium on Theory of Computing*, pages 209–217. 62

Gabriel, S., Schaffner, S., Nguyen, H., Moore, J., Roy, J., Blumenstiel, B., Higgins, J., DeFelice, M., Lochner, A., Faggart, M., Liu-Cordero, S., Rotimi, C., Adeyemo, A., Cooper, R., Ward, R., Lander, E., Daly, M., and Altshuler, D. (2002). The structure of haplotype blocks in the human genome. *Science*, pages 2225–2229. 7, 8, 22, 44

G.F., W. (1998). Phylogenetic profiles: a graphical method for detecting genetic recombination in homologous sequences. *Mol. Biol. Evol*, pages 326–335. 9, 44

Greenwald, M. B. and Khanna, S. (2001). Space-efficient online computation of quantile summaries. In *Proc. of ACM SIGMOD Record*, pages 58–66. 14, 15, 83, 84, 85, 91, 95, 112

Greenwald, M. B. and Khanna, S. (2004). Power-conserving computation of order-statistics over sensor networks. In *PODS*. 61, 83, 86, 89, 95, 100, 104, 105, 112

Guha, S., Mishra, N., Motwani, R., and O'Callaghan, L. (2000). Clustering data streams. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 359–366. 61, 62

Gusfield, D. (2002). Haplotyping as perfect phylogeny: conceptual framework and efficient solutions. In *Proc. of RECOMB*, pages 166–175. 8, 22

Gusfield, D., Eddhu, S., and Langley, C. (2004). Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. *J. Bioinf. Comput. Biol.*, pages 173–213. 8, 22

Har-Peled, S. and Kushal, A. (2005). Smaller coresets for k-median and k-means clustering. In *Proceedings of the 21st annual symposium on computational geometry*, pages 126–134. 62

Har-Peled, S. and Mazumdar, S. (2004). Coresets for k-means and k-median clustering and their applications. In *ACM Symposium on Theory of Computing*. 60, 61, 62, 71

Hein, J. (1990). Reconstructing evolution of sequences subject to recombination using parsimony. *Math. Biosci*, pages 185–200. 9, 44

Hein, J. (1993). A heuristic method to reconstruct the history of sequences subject to recombination. *J. Mol. Evol*, pages 396–405. 9, 44

Holmes, E. C., Worobey, M., and Rambaut, A. (1999). Phylogenetic evidence for recombination in dengue virus. *Mol. Biol. Evol*, page 405. 9, 44

Hudson, R. and Kaplan, N. (1985). Statistical properties of the number of the recombination events in the history of a sample of dna sequences. *Genetics*, pages 147–164. 8, 42, 43, 45

Jain, R. and Chlamtac, I. (1985). The p^2 algorithm for dynamic calculation for quantiles and histograms without storing observations. *Communications of the ACM*, 28(10):1076–1085. 85

Jakobsen, I. B., Wilson, S. E., and Easteal, S. (1997). The partition matrix: exploring variable phylogenetic signals along nucleotide sequences alignments. *Mol. Biol. Evol.*, pages 474–484. 9, 44

Januzaj, E., Kriegel, H., and Pfeifle, M. (2003). Towards effective and efficient distributed clustering. In *Workshop on Clustering Large Data Sets (ICDM2003)*. 60

Keralapura, R., Cormode, G., and Ramamirtham, J. (2006). Communication-efficient distributed monitoring of thresholded counts. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 58

Kreitman, M. (1983). Nucleotide polymorphism at the alcohol dehydrogenase locus of drosophila melanogaster. *Nature*, pages 412–417. 53

Lin, X., Lu, H., Xu, J., and Yu, J. X. (2004). Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *Proc. of of the 20th International Conference on Data Engineering*, page 362. 14, 83, 84, 85

Lole, K. S., Bollinger, R. C., Paranjape, R. S., Gadkari, D., Kulkarni, S. S., Novak, N. G., Ingersoll, R., Sheppard, H. W., and Ray, S. C. (1999). Statistical properties of the number of the recombination events in the history of a sample of dna sequences. *J. Virol*, pages 152–160. 9, 44

Madden, S., Franklin, M. J., Hellerstein, J., and Hong, W. (2002). Tag: a tiny aggregation service for ad-hoc sensor networks. In *Proc. of the 5th Symp. Operating Systems Design and Implementation(OSDI 02)*. 13, 14, 59

Manku, G. S., Rajagopalan, S., and Lindsay, B. G. (1998). Approximate medians and other quantiles in one pass and with limited memory. In *Proc. of ACM SIGMOD Record*, pages 426–435. 14, 83, 84, 85, 98

Manku, G. S., Rajagopalan, S., and Lindsay, B. G. (1999). Random sampling techniques for space efficient online computation of order statistics of large datasets. *ACM SIGMOD*, pages 251–262. 83, 85

Martin, D. and Rybicki, E. (2000). Rdp: detection of recombination amongst aligned sequences. *Bioinformatics*, pages 562–563. 9, 44

Maynard, S. J. and Smith, N. H. (1998). Detecting recombination from gene trees. *Mol. Biol. Evol.*, pages 590–599. 9, 44

Moore, K., Zhang, Q., McMillan, L., Wang, W., and de Villena, F. P. (2008). Genome-wide compatible snp intervals and their properties. In *UNC Tech Report*. 46

Mott, R., Talbot, C. J., Turri, M. G., Collins, A. C., and Flint, J. (2000). A new method for fine-mapping quantitative trait loci in outbred animal stocks. *Proc. Natl. Acad. Sci.*, pages 12649–12654. 7, 21, 22

Munro, J. I. and Paterson, M. (1980). Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323. 83, 84

Myers, S. R. and Griffiths, R. C. (2003). Bounds on the minimum number of recombination events in a sample history. *Genetics*, pages 375–394. 8, 43

N.C., G. and Holmes, E. (1997). A likelihood method for the detection of selection and recombination using nucleotide sequences. *Mol. Biol. Evol*, pages 239–247. 9, 44

O'Callaghan, L., Mishra, N., Meyerson, A., and Guha, S. (2002). Streaming data algorithms for high-quality clustering. In *Proceedings of IEEE International Conference on Data Engineering*. 61

Olston, C., Jiang, J., and Widom, J. (2003). Adaptive filters for continuous queries over distributed data streams. In *Proc. of ACM SIGMOD International Conference on Management of Data 2003*, pages 563–574. 14, 58, 59

Paterson, M. (1997). Progress in selection. *Scandinavian Workshop on Algorithm Theory*. 84

Patil, N., Berno, A., Hinds, D., Barrett, W., Doshi, J., Hacker, C., Kautzer, C., Lee, D., Marjoribanks, C., McDonough, D., Nguyen, B., Norris, M., Sheehan, J., Shen, N., Stern, D., Stokowski, R., Thomas, D., Trulson, M., Vyas, K., Frazer, K., Fodor, S., and Cox, D. (2001). Blocks of limited haplotype diversity revealed by high-resolution scanning of human chromosome 21. *Science*, page 5547. 8, 44

Posada, D. (2002). Evaluation of methods for detecting recombination from data sequences: empirical data. *Mol. Biol. Evol.*, pages 1198–1212. 9, 44

Schwartz, R., Halldorson, B., Bafna, V., Clark, A., and Istrail, S. (2003). Robustness of inference of haplotyp block structure. *J. Comput. Biol.*, pages 13–19. 8, 22

Sharfman, I., Schuster, A., and Keren, D. (2006). A geometric approach to monitoring threshold functions over distributed data streams. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 58

Shrivastava, N., Buragohain, C., Agrawal, D., and Suri, S. (2004). Medians and beyond: new aggregation techniques for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249, New York, NY, USA. ACM Press. 86

Silberstein, A., Braynard, R., and Yang, J. (2006). Constraint chaining: On energy effcient continuous monitoring in sensor networks. In *Proc. of ACM SIGMOD International Conference on Management of Data 2006*, pages 157–168. 14, 59

Song, Y. and Hein, J. (2005). Constructing minimal ancestral recombination graphs. *J. Comput. Biol.*, pages 147–169. xiii, 53, 54

Song, Y. S., Wu, Y., and Gusfield, D. (2005). Efficient computation of close lower and upper bounds on the minimum number of recombinations in biological sequence evolution. *Bioinformatics*, pages i413–i422. 8, 53

Stephens, J. C. (1985). Statistical methods of dna sequence analysis: detection of intragenic recombination or gene conversion. *Mol. Biol. Evol*, pages 539–556. 9, 44

Szatkiewicz, J., Beane, G., Ding, Y., Hutchins, L., de Villena, F., and Churchill, G. (2008). An imputed genotype resource for the laboratory mouse. *Mamm. Genome*, page 199. 55

Szewczyk, R., Polastre, J., Mainwaring, A., and Culler, D. (2004). Lessons from a sensor network expedition. In *Proc. of the 1st European Workshop on Wireless Sensor Networks (EWSN '04)*, pages 307–322. 10

Threadgill, D. W., Hunter, K. W., and Williams, R. W. (2002). Genetic dissection of complex and quantitative traits: from fantasy to reality via a community effort. *Mamm. Genome*, pages 175–178. 20

Ukkonen, E. (2002). Finding founder sequences from a set of recombinants. In *Proc. of WABI*, pages 277–286. 7, 20, 21, 22, 24

Willett, R. M., Martin, A. M., and Nowak, R. D. (2004). Adaptive sampling for wireless sensor networks. In *Proc. of ISIT04*. 14, 59

Worobey, M. (2001). A novel approach to detecting and measuring recombination: new insights into evolution in viruses, bacteria, and mitochondria. *Mol. Biol. Evol*, pages 1425–1434. 9, 44

Wu, Y. and Gusfield, D. (2007). Improved algorithms for inferring the minimum mosaic of a set of recombinants. In *Proc. of CPM*, pages 150–161. 7, 20, 21, 22, 24

Zhang, Q. and Wang, W. (2007). A fast algorithm for approximate quantiles in high speed data streams. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management (SSDBM)*. 112

Zhang, Y., Lin, X., Xu, J., Korn, F., and Wang, W. (2006). Space-efficient relative error order sketch over data streams. 85

Zhu, Y. and Shasha, D. (2003). Efficient elastic burst detection in data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 336–345. 58