HASHING, SEARCHING, SKETCHING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Rina Panigrahy
December 2006

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Rajeev Motwani) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Tim Roughgarden)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Amin Saberi)

Approved for the University Committee on Graduate Studies.

Preface

The Information Age has enabled the search for information in ways never imagined before. The simplest search function may be an *exact search* where the input query is expected to exactly match the search object. But some search criteria are fuzzy – for instance image search, news search, and similar document search – making the search problem much harder. One common approach is to convert such a search object into a mathematical representation such as a point (vector) in a high dimensional space. The search for a similar object then becomes a nearest neighbor search in a high dimensional space.

Hashing is a simple and effective method for exact search that uses a random hash function to map items into buckets, often viewed as throwing balls into bins. A variant of hashing called *locality-sensitive* hashing that tends to map similar objects to the same hash bucket, can be used to perform nearest neighbor search. A related notion is *sketching* that is used to transform a large complex object into a small 'sketch' – often a tiny bitmap – so that similarity between the sketches can be used to estimate the similarity between the original objects.

In this thesis we study algorithms for different kinds of search using hashing and sketching, and some fundamental limits of what can be realized using some of these approaches. For exact search, we will see how variants of balls-and-bins processes can be used to derive space efficient methods for maintaining hash tables. For similarity search, we will see a variant of locality-sensitive hashing that uses linear space and how the underlying ideas can be used in the kd-tree data structure for improved performance. We will also probe the fundamental limits of some of these approaches by showing lower bounds on their performance.

Acknowledgement

I wish to thank my Advisor Rajeev – it was his constant support through the last 9 years that I have been able to maintain in touch with Theory and algorithms while I was working in Industry. I also thank again Prof David Karger MIT for advising me during my years at MIT and for encouraging me to continue working on theory and for introducing me to Prof Motwani after I decided to leave MIT after my masters.

I wish to thank all my collaborators – especially when I was away from school – Tomas Feder, Moses Charikar, Suresh Venkatasubramanian, Piotr Indyk, An Zhu, Liadan O'Callaghan, and Gagan Agarwal to name a few. I wish to thank Samar Sharma at Cisco Systems for interesting research discussions during this period. I would like to thank the students at stanford – Krishnaram Kenthapadi, Dilys Thomas, David Arthur, Ying Xu, Sergei Vassilvitskii, and Shubha Nabar.

No amount of gratitude to my parents would be sufficient. I am very fortunate to be where I am today because of their constant support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Information Age has enabled the search for information in ways never imagined before. We make use of fast automated search when we perform a web search, database search or a network search.

In practice, the search criteria for the desired information are usually complex involving many criteria. At an operational level, the simplest search function may be an *exact search* – for instance, searching for a database record where the key has a specified value. Numerous efficient algorithms exist for exact search and this kind of search is generally easily handled. In other cases we may be performing a *range search*, say when we search for a restaurant in a certain radius of a location.

But some search criteria are fuzzy – for instance, image search, news search, similar document search. These are search examples where the criteria is often not stated precisely but understood at a subjective level. For example, an image search may involve finding the image of a person that best matches a given photograph. In practice, the search may be complex and involve a conjunction of several criteria: some exact, some range search, some fuzzy, and others operations specific to the problem.

For most search criteria, in terms of data structures and algorithms, there are two primary fundamental underlying approaches: Trees and Hashing

1. *Trees:* Trees are based upon the idea of recursive pruning of search space; divide the search space into smaller regions at each decision branching point, resulting in about

$O(\log n)$ search 'depth.' Several variants of trees have been proposed – binary trees, B-trees, Range trees and Kd-trees to name a few.

2. *Hashing:* The idea behind hashing is to use an apparently random function that maps objects to an apparently random location in an array of buckets, and hope that not too many objects collide at the same bucket. This random mapping of objects significantly reduces the search space in one shot. The objective of these random hash functions is to randomly throw objects (thought of as balls) into buckets (thought of as bins). Analysis of variants of such balls-and-bins techniques has resulted in new improved hashing methods. Since the search time in a bucket depends on the number of items in that bucket, the goal is to obtain as uniform a distribution of balls into bins as possible. To this end, it has been shown that instead of randomly throwing balls into bins, if each ball picks two bins at random and is placed into the least loaded of the two bins, then this process results in a significant improvement in the uniformity of the distribution.

Fuzzy search and the Curse of Dimensionality: The problem becomes much harder when we go from exact to fuzzy search, say image search. One common approach is to convert such a search object into a mathematical representation such as a point (vector) in a high dimensional space. As a simple example, an image could be divided into several regions and a number could be used to quantify the average color in a region – such a vector is called as a feature vector. The search for a similar object then becomes a nearest neighbor search in a high dimensional space. The problem of finding the nearest neighbor seems to suffer from the 'curse of dimensionality' – that is, the problem becomes very hard as the number of dimensions increase. One method to circumvent this problem is to relax the problem by finding an approximate near neighbor that is within some constant factor times the distance to the nearest neighbor; if the gap between the nearest and the second nearest neighbor is large this corresponds to the exact nearest neighbor. Also, this is meaningful in practice as typically when we are given a query point we are really interested in finding a neighbor that is much closer to the query point than the other points – the query point (say an image) really represents the 'same object' as the nearest neighbor we expect it to 'match' except that they may differ a little due to noise, or inherent errors in how well

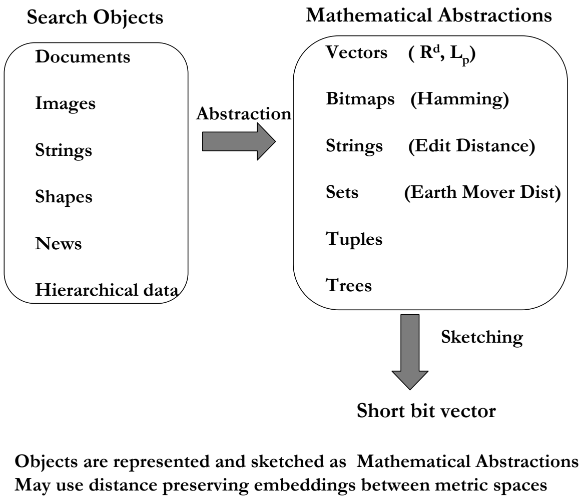

Figure 1.1: Various mathematical abstractions can be used to represent data objects. Embeddings between the different representations can be used to obtain concise sketches for the data.

points represents their objects, but it is expected to be quite far from the other points in the database which basically represent ‘different objects’ from the query point.

Metric Spaces and Embeddings: The Euclidean space is only one of the several mathematical abstractions that be used for similarity search (Figure 1.1); many other spaces with different distance measures can be used. Other candidate metric spaces could be: R^d with different distance measures such as the L_p norm, bitmaps with hamming distance measure, strings with an edit distance measure, sets with earth movers distance, and tree representations for hierarchical data. Given an algorithm for nearest neighbor search in one metric space, it is often possible to carry it to others by using embeddings; embeddings are functions that map points, usually from a complex space to a simpler space, while approximately preserving distances. There are several distance-preserving embeddings with varying degrees of approximation from complex spaces such as arbitrary graph-based metrics, string edit-distance metric, L_p metrics to simpler ones such as Euclidean and hamming space. Such embeddings allow algorithms in one space to be extended to algorithms to other spaces with a certain bounded degradation of the distance measures.

Sketching: Embeddings can also be used to ‘reduce’ high dimensional objects such as documents, news reports, images, geometric shapes to a simple bit string in the hamming space that preserve essential properties of the original objects – such a resulting bitmap is often called a 'sketch' of the object. Sketch similarity can be used to determine likeness of the original objects they represent. In many cases, given a database of n objects, it is possible to produce a sketch of size $O(\log n)$ for each object that is sufficient to estimate their similarity with a high degree of confidence. Since sketches often preserve some of the information in the original object, in some applications they may be used to perform approximate computations without storing the objects explicitly.

Bloom filter: A Bloom filter is a space efficient succinct sketch of a set that can be used to answer set membership, set containment, or set intersection queries with a high degree of confidence. It is an inexact representation of a set that allows for a small probability of false positives when queried; that is, it can sometimes say that an element is in the set when it is not. In return, a Bloom filter offers very compact storage: less than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set. Since bloom filters are compact, they can be stored closer to the processer and can be used for a quick membership check before resorting to secondary memory access and only if required. There has recently been a surge in the popularity of Bloom filters and its variants, especially in networking [16].

Locality Sensitive Hashing: Sketching and hashing can also be used for similarity search in a collection of high-dimensional objects; the idea is to use *locality-sensitive* hash functions that tend to hash similar objects to the same hash value (bucket) and dissimilar objects to different hash values. Contrast this with sketching where similar objects map to the similar bit maps.

Streaming: Hashing and sketching are also commonly used techniques in a stream computation where a large collection of objects arrive in sequence and a certain quantity, say the most frequent element, may need to be computed without storing all the objects explicitly; the idea is to use minimal space while scanning the stream.

In this thesis, we study algorithms for different kinds of search, hashing and sketching, and also the fundamental limits of what can be realized using some of these approaches. For exact search, we show how variants of balls-and-bins processes can be used to derive space efficient methods for maintaining hash tables, resulting in high space utilization. We will also see some new results on balls-and-bins processes when bins are chosen according to a given underlying graph structure and study the distribution of balls into bins in terms of graph properties. Balls-and-bins processes also result in more compact version of Bloom filters for approximate set operations.

We will also study the similarity search problem by viewing it as a nearest neighbor search in a high dimensional space. We will see a variant of locality sensitive hashing that uses linear space by searching for points in the neighborhood of the query point. This method is also applicable to the kd-tree data structure and results in an improved rate of success in finding an approximate nearest neighbor. We will also see tight lower bounds on the performance of the locality sensitive hashing technique for nearest neighbor search. In the area of streaming algorithms, we show tight lower bounds on the space required for finding frequent elements in a stream. We will conclude with a sketching algorithm for similarity search on hierarchical data represented as trees.

Specifically we obtain the following results:

1. *Exact Search Using Hashing:* An important objective in hashing is to reduce the number of collisions that result in many items landing in one bucket thus increasing the search time. A time space tradeoff is obtained by using larger hash tables that increase space overhead but result in fewer collisions. For exact search, we show how hash tables can be maintained with 83% space utilization, while requiring only a few (2) memory accesses per hash lookup.

2. *Balls-and-bins in graphs:* Balls-and-bins processes are central to the analysis of hashing. It is well known that if n balls are randomly inserted into n bins, then with high probability, the bin with maximum load contains $(1 + o(1)) \log n / \log \log n$ balls. Azar, Broder, Karlin, and Upfal [2] showed that instead of choosing one bin, if $d \geq 2$ bins are chosen at random and the ball inserted into the least loaded of the d bins, the maximum load reduces drastically to $\log \log n / \log d + O(1)$. We study the two-choice balls-and-bins process when balls are not allowed to choose any two random bins, but only bins that are connected by an edge in an underlying graph. We show that for n balls and n bins, if the graph is almost regular with degree n^ϵ, where ϵ is not too small, the previous bounds on the maximum load continue to hold. Further this does not hold for non-regular graphs even if the minimum degree is high.

3. *Counting Bloom Filters based on Multiple Choice Hashing:* A counting Bloom filter (CBF) generalizes a Bloom filter data structure so as to allow membership queries on a set that can be changing dynamically via insertions and deletions. As with a Bloom filter, a CBF obtains space savings by allowing false positives. Because of its use of counters, however, CBFs can in practice be prohibitively expensive in terms of space usage. Alternative constructions with the same functionality have been suggested, but they appear much more complex, and have not been tested in practice. We provide a simple hashing-based alternative that offers the same functionality as a CBF, but with much better space usage, generally reducing the space required by a factor of two or more for very practical settings.

4. *Hashing for Nearest-Neighbor Search:* We show a variant of *locality-sensitive* hashing for approximate nearest neighbor search in a Euclidean space that can compute the c-approximate nearest neighbor can be in time $\tilde{O}(n^{\rho})$ and near linear space where $\rho \approx 2.06/c$. A c-approximate nearest neighbor is a neighbor that is within distance at most c times the distance to the nearest neighbor.

5. *Lower bounds on Nearest-Neighbor Search using Hashing:* Locality sensitive hashing is a hash based method for nearest neighbor search that maps points in the space to a discrete space where nearby points out likely to get hashed to the same value and far off points are likely to get hashed to different values. More precisely, given parameter m that denotes an upper bound on the probability that two points at most r apart hash to the same bucket, and g a lower bound on the probability that two points more than cr apart hash to the same bucket, it is known [69] that such a hash function can find a c-approximate nearest neighbor in $\tilde{O}(d + n^{\rho})$ time using a data structure of size $\tilde{O}(n^{1+\rho})$ where $\rho = log(1/m)/log(1/g)$. We show lower bounds on the best possible values for ρ in l_1 and l_2 norm; for the l_1 norm we show that it is impossible to achieve $\rho \leq \frac{1}{2c}$ and for the l_2 norm we show a bound of $\frac{1}{2c^2}$

6. *Nearest-neighbor Search using kd-Trees:* Kd-trees are one of the most commonly used methods for nearest neighbor search. For low dimensions this data structure can be used for answering nearest neighbor queries in logarithmic time and linear space. However the performance seems to degrade as a number of dimensions becomes

larger than two. We show that a simple modification to the search algorithm on a kd-tree can be used to find the nearest neighbor in high d-dimensions. The modification consists of simply perturbing the query point before traversing the tree, and repeating this for a few iterations. For a certain database of random points, we show that kd-tree succeeds in finding a c-approximate nearest neighbor only with probability $e^{-\Omega(d/c)}$, where as the modified algorithm gives a much higher rate of success.

7. *Space lower bounds on finding frequent element in a stream:* It is well known that the space complexity of finding the frequent element in a stream of n elements is $\Omega(n)$ in the worst case. However in practice, often the element frequencies follow a well known distribution such as the zipfian or a normal distribution. We show, for any given distribution, tight lower bounds on the space complexity in terms of the moments of the distribution. Specifically we show a space lower bound of $\Omega(F_2/F_\infty^2)$ for finding the most frequent element in a stream.

8 *Sketching Algorithms for Trees:* We study sketching algorithms for computing similarities between hierarchical data. Specifically, we look at data objects that can be represented using leaf-labeled trees denoting a set of elements at the leaves organized in a hierarchy. We measure distance between such trees using the best possible super-imposition of the trees that minimizes the number of mismatched leaf labels. Our distance measure is also equivalent to an *Earth Mover's Distance*(EMD) measure since a leaf-labeled tree of height one can be viewed as a set. This definition can be recursively extended to multi-level trees. We propose algorithms to sketch trees and analyze them in the context of locality-sensitive hashing (LSH) where the probability of two sketches matching is high when two trees are similar and low when the two trees are far under the given distance measure. Specifically, we compute sketches of sets by propagating min-hash computations up the tree. Furthermore, we show that while propagating one min-hash per level gives poor results, propagating two min-hashes produces good sketches.

Chapter 2

Efficient Hashing with Lookups in two Memory Accesses

2.1 Introduction

Hashing is a simple and effective method for exact search. An important objective in hashing is to reduce the number of collisions since they cause many items to land in one bucket thus increasing the search time. A time space tradeoff is obtained by using larger hash tables that increase space overhead but result in fewer collisions. We show how hash tables can be used with high space utilization and few memory accesses per hash lookup.

The study of hashing is closely related to the analysis of balls and bin. One of the classical results in this area is that, asymptotically, if n balls are thrown into n bins independently and randomly then the largest bin has $(1 + o(1)) \ln n / \ln \ln n$ balls, with high probability. Azar et. al. [2] showed that instead of using a single hash function, if we randomly hash a ball into two bins and place it in the smaller of the two, then this dramatically lowers the maximum load on bins. This leads to the concept of two-way hash functions (figure 2.1) where the largest bucket contains $O(\log \log n)$ balls. The hash look up will now search in both the buckets an item hashes to. The intuition behind this exponential improvement in the maximum load is as follows: if p_i denotes the fraction of bins with load at least i, then observe that at any step the probability that a ball ends up in a bin with load at least i (to create a bin of at least $i + 1$) is p_i^2 as this can happen only if both bins have height at least

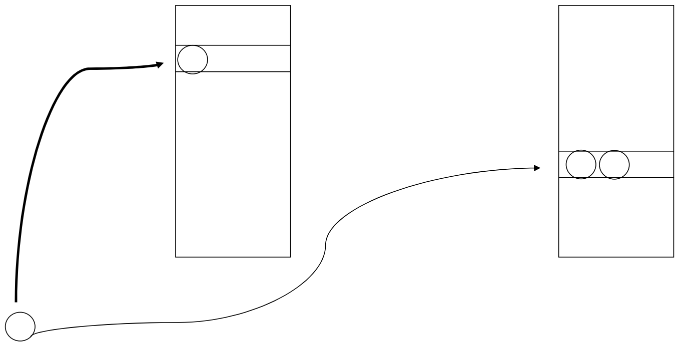

Figure 2.1: Balls and bins: If each ball is thrown into a random bin, the max load is $(1+o(1))\ln n/\ln\ln n$. Instead if each ball is inserted into least loaded of 2 randomly chosen bins the max load drops exponentially to $\log\log n + O(1)$

i; this suggests that the expected fraction of bins with load at least $i + 1$, $p_{i+1} \approx p_i^2$, giving the $O(\log\log n)$ growth. So dramatic is this improvement that it can be used in practice to efficiently implement hash lookups in packet routing hardware [20]. The two hash lookups can be parallelized by placing two different hash tables in separate memory components. However, to simplify our presentation and analysis, we will assume that only one hash table is used. We will also assume that the hash functions used are truly random.

Note that since an item may be placed in one of two buckets, we could potentially move an item after it has been initially placed to reduce maximum load. While it was known that if all the random choices are given in advance, balls could be assigned to bins with a maximum load of 2 with high probability [37], we show that this can be achieved on line while supporting hash up date operations. In fact, even more than n, up to $1.67n$, items can be stored in n buckets, with a maximum load of two items, by performing at most $\log\log n + O(1)$ moves during inserts, with high probability. Even if the space for two items are pre-allocated per bucket, as desirable in hardware implementations to avoid dynamic allocation, this represents only a 16.25% wastage of space - over 83.75% utilization. Memory utilization is a crucial issue in several hash implementations, especially hardware implementations where a large number of memory components consume critical resources of board space, ASIC pin count and power. Our algorithm requires a bfs (breadth first search) exploring at most $O(\log n)$ nodes with high probability and constant in expectation. Alternatively, to avoid a bfs, we show that one could simply perform a random walk of length $O(\log n)$ to maintain a maximum load of two provided $m < 0.65n$; for larger m this would give a constant load as long as $m = O(n)$.

We also analyze the trade-off between the number of moves performed during inserts and the maximum load on a bucket. A solution requiring fewer moves may be more attractive in practice as moves may be expensive; also it may be desirable to avoid a bfs traversal that may be infeasible in hardware implementations. By performing at most h moves during inserts, we can maintain a maximum load of $O(\frac{\log\log n}{h\log(\log\log n/h)})$. So even by performing one move, we achieve a better bound than by performing no moves at all. This result holds even if the hash functions used are not truly random but are $O(\log n)$-way independent. Setting $h = O(\log\log n)$ implies that we can maintain a constant maximum bucket size even if $O(\log n)$-way independent hash functions are used. Several recent works [96] [48] demonstrate how such functions can be evaluated in constant time and implemented efficiently without using much storage.

This idea of moving items has been used earlier in cuckoo hashing [103], however, they allow only one item per bucket. With two hash tables this requires 100% memory overhead. They also show that the amortized insert time with cuckoo hashing is a constant. Fotakis et al [55] generalized the method to d-ary hashing, using d hash tables, and truly random hash functions, but still allowing only one item per bucket. They showed that with ϵ memory overhead, one can support hash lookups in $O(\ln 1/\epsilon)$ probes and constant amortized insert time assuming the hash functions used are truly random. Our use of bfs to find a bucket with empty space is similar to theirs. They also provide an alternate scheme that performs lookups in $O(\ln^2 1/\epsilon)$ probes while using polynomial hash functions of degree $O(\ln 1/\epsilon)$ that can be evaluated in constant time.

However, in practice, memory operations requiring more random accesses is more expensive than reading the same amount of memory in few accesses and larger bursts. For most forms of memory such as DRAMs and disks, the latency of the initial random access is much higher than that of fetching data from subsequent locations. Also, in hardware implementations, probing a large though constant number of tables will require as many memory components to be accessed efficiently in parallel. Our method involves two memory accesses and achieves a 83.75% memory utilization. Note that this utilization is what can be provably achieved and is not tight; although we can show an upper bound of 93% for our algorithm. At the same time, we should point out that our stated memory utilization is proved assuming that the hash functions used are truly random.

A closely related work [49] that appeared just after this work was published. studies the same algorithm as ours but for larger bucket sizes. They show that with buckets of size $O(1/\epsilon)$, and two hash tables, a dictionary data structure can be maintained with ϵ fraction space overhead. Further, they show that for buckets of size more than about $90/\epsilon$, inserts can be performed in constant expected time. Other related work includes the first static dictionary data structure with constant look up time by Fredman, Komlos and Szemeredi [54] that was generalized to a dynamic data structure by Dietzfelbinger et al. in [43] and [46]. In practice, however, these algorithms are more complex to implement than cuckoo hashing. Extensive work has been done in the area of parallel balls and bins [3] and the related study of algorithms to emulate shared memory machines (as for example, PRAMs) on distributed memory machines (DMMs) [47] [34] [91] [110]. This setting involves a parallel game of placing balls in bins (the so-called collision game) where all n balls participate in rounds of parallel attempts to assign balls to bins. In each round, you test both locations of every ball that has not been placed yet. If a ball has a location tested by at most some constant number of other balls, you place it. It has been shown in that $loglogn + O(1)$ rounds indeed suffice to place all n balls, with high probability [47] [34]. This however does not imply our result that $loglogn + O(1)$ moves are sufficient to maintain maximum load of 2 because of the different setting.

2.2 Overview of Techniques

Viewing buckets as bins and items as balls, we can look at the hashing process as if m balls are being assigned to n bins. For each ball two bins are chosen at random. If the bins are imagined to be the vertices of a graph, the two bins for a ball can be represented by an edge. This gives us a random graph G on n vertices containing m edges. By making this graph directed, we could use the direction of an edge to indicate the choice of the bin among the two for placing the ball. The direction of each edge is chosen online by a certain procedure. The load of a vertex (bucket) is equal to its in-degree. For each edge (item) insertion, the two-way hash algorithm directs the edge towards the vertex with the lower in-degree. During the hash process, say U is one of the vertices a ball gets hashed to. Observe that if VU is a directed edge, and if the load on V is significantly lower, we could perform a move from U to V, thus freeing up a position in U. Essentially, in terms of load, the new ball could be added to either U or V, whichever has a lower load. This principle could be generalized to the case where there is a directed path from V to U, and would result in performing moves and flipping the directions along all the edges on the path. If there is a directed sub-tree rooted at U, with all edges leading to the root, we could choose the least loaded vertex in this tree to incur the load of the new ball. With this understanding, we will say that W is a child of X if XW is a directed edge. So, our hash insert algorithm looks as follows.

- Compute the two bins U_1 and U_2 that the new item to be inserted hashes to.
- Explore vertices that can be reached from U_1 or U_2 by traversing along directed edges in the reverse direction.
- Among such vertices, find one, V, with low load that can be reached say from U_1.
- Add the new item to U_1 and perform moves along the path from U_1 to V so that only the load on V increases by one.

Let $s = 2m/n$ denote the average degree of the undirected random graph G. Note that the same graph G can be viewed as a directed or an undirected graph. Throughout the paper G refers to the undirected version unless stated otherwise or clear to be so from the context. Throughout the paper we will assume that s is a constant. It turns out that the success of our algorithm in maintaining low maximum load depends on the absence of dense subgraphs in this random graph. We show that such dense subgraphs are absent when $s < 3.35$, giving an algorithm that works with bucket size at most 2 and requiring at most $\log \log n + O(1)$ moves for inserts with high probability (section 2.3). Note that the bound of 3.35 for s may not tight but is provably no more than 3.72. We then analyze the trade off between number of moves during inserts and maximum bucket size using the technique of witness trees [34] [91] [3], making significant adaptations to our problem (section 2.4).

2.3 Constant Maximum Bucket Size

In this section we show that for $s < 3.35$ by performing at most $\log\log n + O(1)$ moves, we can ensure that with high probability no bucket gets more than 2 items.

For an insert, we search backwards from a given node in bfs order, traversing directed edges in reverse direction, looking for a node with load at most one. To simplify the analysis, we assume that during the backward search, the algorithm visits only 2 children for each node even if more may be present. We will show that by searching to a depth of $\log\log n + O(1)$, with high probability, we find a node with load at most one. First, we show that if the backward search is allowed to proceed to unlimited depth, the success of the algorithm is related to a certain property of the random graph G.

Lemma 2.3.1. *If the backward search during inserts is allowed to proceed to any depth, the above algorithm succeeds in inserting all m items while maintaining a maximum load of 2 if and only if the graph G does not have a subgraph with density greater than 2. Here density is the ratio of number of edges to vertices in the subgraph.*

Proof. Clearly, if there is such a subgraph, it is impossible to orient the edges so that the in-degree on every vertex in the subgraph is at most 2. So it is not possible to have inserted all elements and still have a load at most 2 on every vertex.

Conversely, if an insert does not succeed, it means the backward search does not find a node of load less than 2. Since the search was not limited to a bounded depth, it must have got stuck in a set of nodes all with load at least two and leading to each other by traversing edges in reverse direction. Then this set of nodes is a subgraph of density at least two. □

The existence of dense subgraphs in random graphs displays a *critical point* behavior; that is, there is a sharp threshold such that almost all random graphs with edge-density larger than the threshold value have such a subgraph and almost all with edge-density less than the threshold value have none. This is because the existence of a dense subgraph is a monotone property, and all such properties were shown to display a sharp threshold behavior by Friedgut and Kalai [53]. A closely related property, the existence of a k-core in random graphs, has been studied extensively and the threshold values have been pinned down exactly. A k-core is a maximal non-empty subgraph where every node has degree at least k. Pittel et al [104] showed that for the existence of a 3-core the critical value is about 3.35. Note that existence of a subgraph with density greater than 2 implies existence of a 3-core. This is because by iteratively deleting nodes with degree at most 2 we must be left with a non-empty 3-core as the number of deleted edges is at most than twice the number of deleted vertices, less than the total number of edges. This means that the threshold value for the existence of a 2-dense subgraph is at least 3.35. We will show that it lies between 3.35 and 3.71. Further, we will show that for $s \leq 3.35$, not only does an inserts succeed with high probability but also takes less than $\log\log n + O(1)$ moves. It is interesting that this value of s coincides with the threshold value for existence of a k-core, but not surprising as we use methods similar to that for k-core in lower bounding the threshold value. Although this value was also shown to be tight for the existence of 3-core by Pittel et al [104], it is unlikely to be so for the existence of 2-dense subgraph.

Since our strategy is to search for a node with load at most one, first we show that it is unlikely to get stuck in a situation where $o(n)$ nodes have been explored, each with load at least 2, and they all lead to one another with no new nodes to visit. This follows from the following lemma as if we do get stuck, we have found an induced subgraph where every node has in-degree at least two.

Lemma 2.3.2. *With high probability, $1 - O(1/n^2)$ there does not exist an induced subgraph of size $o(n)$ in G where every node has in-degree at least 2. This implies that the backward search cannot get stuck with high probability if it is allowed to proceed to any depth.*

Proof. If there is such a subgraph of x nodes, it must have at least $2x$ edges. We will show that the probability of such an event is negligible. Number of ways of choosing x vertices and $2x$ edges from the m edges is $\binom{n}{x}\binom{sn/2}{2x}$. Probability of a given edge falling in this subgraph is $\frac{\binom{x}{2}}{\binom{n}{2}} \leq \frac{x^2}{n^2}$ So the probability of finding such a subgraph of x nodes is

$$\begin{aligned}
&\leq \binom{n}{x}\binom{sn/2}{2x}(\frac{x^2}{n^2})^{2x} \\
&\leq (\frac{en}{x})^x(\frac{esn/2}{2x})^{2x}(\frac{x}{n})^{4x} \\
&\leq (\frac{e^3s^2x}{16n})^x
\end{aligned}$$

We need to sum of this expression over all possible values of x. Since x is at least 2 and at most $o(n)$, the total probability is $O(1/n^2)$.

□

Let us perform a bfs on the undirected graph G starting from a certain node V to a depth of h. Note that this is different from the backward search from the same node to a depth of h that also involves a bfs along directed edges in reverse direction. To distinguish between the two we will refer to the former as 'bfs on the undirected graph' and the latter as 'backward search'. Let $BFS_h(V)$ denote the subgraph visited by the bfs on the undirected graph to a depth of h. Clearly the nodes visited in the backward search to a depth of h will be a subset of those visited in $BFS_h(V)$ to a depth of h. We will compare this bfs on the random undirected graph G to a branching process. Since $sn/2$ edges are randomly thrown into the graph G on n vertices, each of the total of sn endpoints of these edges are chosen randomly. If we ignore the possibility of forming self loops and choose these endpoints independently, a node will have k edges incident on it with probability $\alpha_k = \binom{sn}{k}(1/n)^k(1-1/n)^{sn-k} \approx e^{-s}s^k/k!$ (accurate for large n and $k << n$ and can be safely used in summations). This probability is asymptotically accurate even if we condition on a certain subgraph with at most $o(n)$ nodes and edges as it makes a negligible difference in the ratio of remaining nodes and edges.

Consider a branching process where each node has k children with this probability α_k; this branching process is completely separate from the bfs and simply constructs a tree where each node has k children with this probability α_k. Let BRT_h be the tree obtained by running such a branching process to a depth of h. We will later show that assuming no cycles are found during the bfs to depth h, the tree $BFS_h(V)$ that is obtained has asymptotically the same distribution as that of BRT_h. If $BFS_h(V)$ is a tree and only contains nodes with load at least two, then one can embed a complete, balanced binary tree of depth h in it. We will show that the probability of this event is close the probability of the being able to embed a complete, balanced binary tree of depth h in BRT_h. The next two lemmas show that it is unlikely to be able to embed a complete, balanced binary tree of depth h in BRT_h if $s \leq 3.35$.

Lemma 2.3.3. *Let p_i be the probability that a complete, balanced binary tree of depth i can be embedded in the tree BRT_i obtained by running the branching process to depth i. Then $p_{i+1} = 1 - e^{-p_i s}(1 + p_i s)$*

Proof. We compute p_i recursively. Look at a node at height $i + 1$. At least two of its children must satisfy the recursive property which happens with probability p_i. If there are k children, probability that less than 2 of them satisfy the property is $(1-p_i)^k + kp_i(1-p_i)^{k-1}$. Probability of having k children $= \alpha_k$

So,

$$\begin{aligned}
p_{i+1} &= \sum_{k\geq 2} \alpha_k(1 - (1 - p_i)^k - kp_i(1 - p_i)^{k-1}) \\
&= \sum_{k} \alpha_k(1 - (1 - p_i)^k - kp_i(1 - p_i)^{k-1}) \\
&= 1 - \sum_{k} \alpha_k(1 - p_i)^k - \sum_{k} \alpha_k k p_i(1 - p_i)^{k-1} \\
&= 1 - e^{-s} \sum_{k} \frac{s^k}{k!}(1 - p_i)^k - e^{-s} \sum_{k} \frac{s^k}{k!} k p_i(1 - p_i)^{k-1} \\
&= 1 - e^{-s}e^{s(1-p_i)} - e^{-s} p_i s \sum_{k\geq 1} \frac{s^{k-1}}{(k-1)!}(1 - p_i)^{k-1} \\
&= 1 - e^{-s}e^{s(1-p_i)} - e^{-s} p_i s e^{s(1-p_i)} \\
&= 1 - e^{-p_i s}(1 + ps)
\end{aligned}$$

Note that in this computation the approximation of α_k as $e^{-s}s^k/k!$ need not be used; each summations can be computed with the exact value of $\alpha_k = \binom{sn}{k}(1/n)^k(1 - 1/n)^{sn-k}$. This still does not affect the final value. □

Lemma 2.3.4. *For any $s < s_0 \approx 3.35$, the probability that a complete, balanced binary tree of depth h can be embedded in BRT_h, can be made smaller than $1/n^c$, for any constant c, by choosing $h = \log\log n + O(1)$*

Proof. As long as s is such that $1 - e^{-ps}(1 + ps)$ is always less than p for any $p \in (0, 1]$, the sequence p_i is monotonically decreasing. If ps is very small this expression is close to $p^2 s^2$ as e^{-ps} can be approximated as $1 - ps$. If $p_i - p_{i+1}$ is at least some small positive constant, in a constant number of steps p can be made smaller than $1/(10s)$, after which it starts decreasing quadratically each step with the recursion $p_{i+1} = p_i^2 s^2$ that is equivalent to $p_{i+1}s^2 = (p_i s^2)^2$. So after this point, in $\log\log n + O(1)$ steps, the probability should drop below $1/n^c$.

We want that for any $p \in (0, 1]$

$$1 - e^{-ps}(1 + ps) < p \Leftrightarrow e^{ps} < \frac{1 + ps}{1 - p}$$

By writing both sides as a Taylor series in p and comparing, we see that this is satisfied if

$$s^2/2 < s + 1 \Leftrightarrow s < \sqrt{3} + 1 < 3.74$$

The exact value of s_0 is determined by setting it to the smallest value of s for which the function $f(x) = 1 + xs - e^{xs}(1 - x)$ satisfy the condition $f(x) > 0$ in the interval $[0, 1]$ □

A better value of $s_0 \approx 3.35$ is obtained by plotting graphs for the functions $f(x) = 1 + xs - e^{xs}(1 - x)$ in the interval $[0, 1]$ showing that $f(x) > 0$ for $s \leq 3.35$ in this interval.

This value of s is tight; that is, for $s > 3.36$ it can be shown that p converges to 0.5, implying that it is possible to embed a binary tree.

Next we extend this result on the tree obtained from the branching process to any tree that may be obtained by the bfs.

Lemma 2.3.5. *With high probability, $1 - O(1/n^{c-1})$, there does not exist a node V in G so that the bfs from V to depth $h = \log\log n + O(1)$ does not encounter any cycles and results in a tree containing a complete, balanced binary tree of depth h embedded in it. (Note that the bfs could be performed from an edge UV where the first level of bfs from the root V does not visit U. This is a technical detail that will be used later.)*

Proof. We will argue that if the bfs results in a tree, its distribution is asymptotically same as that produced by the branching process. First note that the total number of nodes visited is small as compared to n, as the maximum degree d is $O(\log n)$ with high probability and the values of h in consideration is $O(\log\log n)$, and so the total number of nodes, d^h, is $(\log n)^{O(\log\log n)}$.

Even if we condition on the existence of a certain subgraph with at most $o(n)$ nodes and edges it makes a negligible difference in the ratio of remaining nodes and edges. So during the bfs, after exploring say at most x nodes and edges (x is at most $(\log n)^{O(\log\log n)}$), the conditional probability that the next node to be expanded has k (k is at most $O(\log n)$) edges emanating from it all of which lead to new nodes, is very close to α_k. It can be verified that the conditional probability is at most $\binom{n}{k}\binom{sn/2}{k}k!(\frac{2}{(n-x-1)^2})^k(1-\frac{2(n-x)}{(n-1)^2})^{sn/2-x}$ – number of ways of choosing k child nodes and edges to those nodes is at most $\binom{n}{k}\binom{sn/2}{k}k!$; probability that one of the k edges leads to the chosen child is at most $1/\binom{n-x}{2}$; probability that each of the remaining $sn/2-x$ edges are not incident on this node is at least $(n-x)/\binom{n}{2}$ as at least $n-x$ edge positions are forbidden. This upper bound differs from α_k by at most a multiplicative factor of $1+O(kx/n)$, for the small values of k and x under consideration. So the probability that the bfs and the branching process produce identical trees of a given structure with at most x nodes, differ by at most a multiplicative factor of $1+O(kx^2/n) = 1+o(1)$. So by applying this argument to all possible trees that can have a complete, balanced binary tree of depth h, embedded in it, we can conclude that since with high probability of $1-O(1/n^c)$, BRT_h cannot have a complete binary tree embedded in it, same must be true about $BFS_h(V)$ even if it were a tree. Clearly this can be extended to all vertices V with high probability of $1-O(1/n^{c-1})$. □

So far we have only considered the case that $BFS_h(V)$ is a tree. Let us prove that it is very unlikely that the bfs finds too many edges that create cycles, where by cycle-creating edges we mean the edges that lead to already visited nodes during the search..

Lemma 2.3.6. *With high probability,* $1-O(1/n^c)$ *there does not exist a subgraph of* $x \leq c\log n$ *nodes in G with at least* $x+O(c)$ *(precisely,* $x+c(4+\log(s/2))$*) edges.*

Proof. If there is such a subgraph of x nodes, we will show that the probability of such an event is negligible. Number of ways of choosing x nodes and $x+u$ edges from the $sn/2$

edges is $\binom{n}{x}\binom{sn/2}{2+u}$. Probability of a given edge falling in this subgraph is $\frac{\binom{x}{2}}{\binom{n}{2}} \leq \frac{x^2}{n^2}$ So the total probability is

$$\begin{aligned}
&\leq \binom{n}{x}\binom{sn/2}{x+u}(\frac{x^2}{n^2})^{2x+2u} \\
&\leq (\frac{en}{x})^x(\frac{esn/2}{x+u})^{x+u}(\frac{x}{n})^{2x+2u} \\
&\leq e^{2x+u}(s/2)^{x+u}(\frac{x}{n})^u \\
&\leq (\frac{es}{2})^u e^{2x}(\frac{s}{2})^x(\frac{x}{n})^u \\
&\leq (\frac{es}{2})^u e^{2c\log n}(\frac{s}{2})^{c\log n}(\frac{c\log n}{n})^u \\
&\leq (\frac{es}{2})^u n^{c(2+\log(s/2))}(\frac{c\log n}{n})^u
\end{aligned}$$

By setting $u = c(4 + \log(s/2))$ this becomes $O(1/n^c)$ □

The following lemma shows that a bfs to a depth of $o(\log n)$ can not encounter more than $5c$ edges that create cycles.

Lemma 2.3.7. *For $s < s_0 \approx 3.35$, with high probability, $1 - O(1/n^c)$, in a subtree T of G with height $o(\log n)$ there cannot be $5c$ edges in G that are between nodes in T but are not edges of T.*

Proof. For if there were, then consider the tree spanning end-points of these $5c$ edges from G not in T, obtained by taking the union of all the paths from these end-points to the root. As the number of endpoints of these $5c$ edges is at most $10c$ and each requires at most $o(\log n)$ edges to connect to the root, the size, x, of this spanning tree is clearly less than $c\log n$.

Adding the $5c$ edges to the spanning tree gives us at least $x+5c$ edges. By lemma 2.3.6, for $s < 4$, this is unlikely and has probability at most $O(1/n^c)$. □

Now we will show that a large, complete binary tree cannot be embedded in G.

Lemma 2.3.8. *With high probability, $1 - O(1/n^c)$, it is not possible to embed a complete, balanced binary tree B of height $h = \log\log n + O(1)$ in the random graph G.*

Proof. Assume that we can embed such a binary tree B rooted at V in G. Perform a bfs from V to a depth of h. By lemma 2.3.7, at most $5c$ cycle creating edges can be found with high probability in $BFS_h(V)$. Let $BFS'_h(V)$ denote the tree obtained by deleting these $5c$ edges from $BFS_h(V)$. There must be some node V' in B at depth at most $\log(5c)+1$ so that the binary subtree rooted at that node is still intact in $BFS'_h(V)$; that is it does not contain any of the $5c$ deleted edges. Let B' denote the binary subtree of B rooted at V'. Now look at the at most $10c$ paths from the endpoints of these deleted edges to V. Since any single path can intersect at most 2 nodes at a certain level in B', there must be some node V'' at depth $\log(20c) + 1$ in B' that is not on any of these $10c$ paths. Also, at least one of the two children of V'' in B' (say W) must also be a child of V'' in $BFS'_h(V)$, as V'' has at most one parent in $BFS'_h(V)$. Look at the binary subtree B'' of B' rooted at W. The height of B'' differs from that of B by at most $\log(5c) + \log(20c) + 3$. Also the bfs from the edge $V''W$ (that is, the first level of the bfs from W does not visit V'') is free of cycles as otherwise V'' is on one of the $10c$ paths. Further it has a complete, balanced binary tree B'' embedded in it. By choosing h large enough we can ensure that height of B'' is at least that required by lemma 2.3.5 giving a contradiction □

We are now ready to prove that during an insert a backward search to a depth of $\log\log n + O(1)$ must find with high probability a node with load less than 2. The total search time is at most $O(\log n)$.

Theorem 2.3.1. *For $s < s_0 \approx 3.35$, with high probability, $1 - O(1/n^2)$, during an insert, if we traverse backward to a depth of $\log\log n + O(1)$, we will have found a node with load less than 2, with high probability, while searching at most $O(\log n)$ nodes. The expected time for this search is $O(1)$.*

Proof. Assume that during an insert, we don't find a node of load less than 2. Then since with high probability by lemma 2.3.2 we cannot get stuck after a few levels and by lemma 2.3.7 we cannot encounter more than $5c$ cycle producing edges, there must be a node at depth $\log(5c) + 1$ so that the backward search under that does not find any cycles. This gives a complete binary tree of height $\log\log n + O(1)$, contradicting lemma 2.3.8.

The expected depth of search is constant as can be seen by the quadratic drop of p_i with i. □

This proves that inserts can be made while maintaining a maximum load of 2, with high probability. The algorithm works even if the number of items, m is greater than n as long as $2m/n \leq 3.35$. Even if the two entries in each buckets are statically allocated, we can achieve a memory utilization $m/(2n)$ of $3.35/4 > 83.75\%$. Thus the memory wastage is only 16.25%.

Note that our value of $s = 3.35$ may not be tight for maintaining a maximum load of two as the calculation was done based on existence of a complete binary tree, which may not be necessary for the existence of a 2-dense subgraph nor for being able to perform inserts in $\log\log n + O(1)$ moves. It is easy to show, however, that for $s > 3.72$, it is impossible to maintain a maximum load of two. This is because for such a random graph, by deleting isolated nodes and nodes of degree one, we end up with a non-empty component with density greater than 2.

Generalizing to constant bucket size larger than 2: Our analysis for maximum bucket size of 2 can be generalized to any constant maximum load i. It turns out that the best provable memory utilization remains around 80% for initial value of $i > 2$ and then drops for larger i.

2.3.1 Random Walk.

The previous algorithm performs a bfs. An alternate algorithm is to simply perform a random walk to look for a lightly loaded node. We show that for $m < 0.65n$, a random walk of length $O(\log n)$ will reveal a node with load at most 1. Again, first we ignore the possibility of running into cycles.

Theorem 2.3.2. *With high probability,* $1 - O(1/n^2)$*, for any* $s < 1.3$*, a random walk of length* $O(\log n)$ *will find a node with load at most two.*

Proof. We will show that the probability of finding a long path where every node has load at least two is negligible. Probability that a node has k children is $e^{-s}s^k/k!$. Number of ways of choosing the next node on the path is at most k. So, for each node, number of ways of choosing next node weighted by probability is $\sum_{k\geq 2} \frac{e^{-s}s^k}{k!} = s(1 - e^{-s})$. As long as this is less than one, the probability of finding a path of length $O(\log n)$ is negligible. This is true for $s < 1.3$. □

Theorem 2.3.3. *With high probability,* $1 - O(1/n^2)$*, for* $m = n$*, a random walk of length* $O(\log n)$ *will find a node with load at most* 4.

Proof. We will show that the probability of finding a long path where every node has load at least four is negligible. Probability that a node has k children is $e^{-s}s^k/k!$, where in this case $s = 2$. Number of ways of choosing the next node on the path is at most k. So, for each node, number of ways of choosing next node weighted by probability is $\sum_{k\geq 4} \frac{e^{-2}2^k}{k!}$. Since this is less than one, the probability of finding a path of length $O(\log n)$ is negligible. □

To address the possibility of cycles, on finding an edge that produces one, we simply backtrack by the fewest possible number of edges and continue our search as if in a DFS. We will show that this backtracking can not happen too often. If this happens c times, we get a graph of size at most $O(\log n) + c$ that has c more edges than nodes. By choosing c the be large enough constant, we can satisfy the condition of lemma 2.3.6, proving that this is unlikely.

2.4 Generalizing to fewer moves

So far we have looked at the number of moves required to maintain a constant load. Here we examine the maximum load when fewer bins are explored. In particular, we could examine only the two bins and their children. So, if an item gets hashed to say U_1 and U_2, we could examine only U_1, U_2 and the children of U_1 and U_2, and pick the least loaded of these to bear the new load. This would require at most one move. Instead of examining the children to a depth of one, we could explore all the descendants to a depth of h by performing a bfs along directed edges in the reverse direction. By restricting the search to a depth of h, we ensure that at most h moves are required. In this section we upper bound the maximum load when all descendants up to depth h are examined during inserts.

The basic intuition is that if the load of the new item is borne by a node with load i, then each of examined nodes must have at least i children. So we must have explored roughly a total of i^h nodes, each with a load of at least i. If p_i is the probability of a node having load at least i, then assuming these events are independent, they happen with probability $p_i^{i^h}$. This gives us approximately, $p_{i+1} = p_i^{i^h}$, and so $p_i = 2^{-\Omega(i-1)!^h}$. p_i becomes $o(1/n^c)$

for $i > O(\frac{\log\log n}{h\log(\log\log n/h)})$ We give a more formal proof of this result without making the independence assumption.

Our proof is based on the *witness tree* approach – one of earliest uses of this approach can be found in [34] [91] [3]. Consider an event that leads to a load of $6l$ at a certain node. For this event to happen, we will show that there must exist a tree of large size obtained by tracing all the events that must have happened earlier. The approach however requires significant adaptation to our problem as the directions of the edges change over time. To simplify the exposition, we will state the proof assuming $m = n$ ($s = 1$); essentially, the same proof works for any constant s.

Construction of the witness graph: Whenever the load of a node X becomes i, there must be a unique edge whose insertion causes this to happen. Say U_1U_2 was this edge; that is, U_1 and U_2 are the bins to which the item got hashed. Look at the directed graph when this edge was being added. During the insertion, a backward search to a depth of h was performed from both U_1 and U_2. Say the node X was obtained by traversing back from U_1 to depth of at most h. We will say that the edge U_1U_2 is the *ith contributing-edge* of X, U_2 is the *ith contributing-peer* of X, and the directed path from X to U_1 along which moves were made, is the *ith contributing-path* for X. Since X is a node with minimum load among the ones visited, it must be the case that all the nodes at depth at most h from U_2 must have load at least $i - 1$. Note that the contributing edge U_1U_2 must be newer than and therefore distinct from all the edges traversed in backward search from U_2. Also for each node and each value of i the *ith*-contributing edge has to be unique.

The witness graph is obtained by recursively chasing contributing edges for nodes visited in the backward search from the contributing peer U_2. First we make a simplifying assumption that during the construction of the witness graph, we never run into cycles, always leaving the graph as a tree. Later, as in section 2.3, we will argue that the number of edges that produce cycles is few enough that they can be ignored. Our goal is to obtain a large witness tree with high degree nodes and argue that such a subgraph is unlikely to exist in G. The problem is that even with our assumption of not encountering cycles, it is still possible to visit earlier nodes through contributing paths, as contributing paths could completely consist of edges in the visited subtree, not leading to any new nodes. We overcome this issue by computing the witness tree by the following recursive procedure.

For a given i-contributing edge U_1U_2:

- Say, U_2 is the *ith* contributing-peer corresponding to this edge; that is, the load for this insert was taken by some node under U_1.
- Look at the subtree, T, (must be a tree by assumption of not encountering cycles) obtained by performing a backward search to depth h from node U_2 when the insertion took place. Look at the set of leaves, L, of this subtree. At that time each node in T has load at least $i-1$. Since each internal node in L has at least $i-1$ children the number of edges in L is at most $2|L|$.
- Look at the set S of all edges that are j-contributing edges for some node in L, for either $j = i-1$ or $i-2$ or $i-3$. Essentially an edge $e \in S$ if and only if there is a node $V \in L$ and a $j \in \{i-1, i-2, i-3\}$ such that e is the j-contributing edge for V. Since the subtree T has at most $2|L|$ edges and the set S has $3|L|$ edges, there must be a set Q of at least $|L|$ edges in S that are outside the subtree T. As all the contributing paths leading to these edges are older than the edge U_1U_2 that connects the subtree to the rest of the witness tree, and since by assumption no cycles are encountered, these edges in Q must be outside the entire witness tree constructed so far. To avoid cycles, the corresponding contributing paths must branch off T before reaching the contributing edge in Q.
- Repeat recursively for each j-contributing-edge in Q, where $j \geq i-3$

We chop the recursion depth down to l. Also, during the backward bfs, for each node, we pick only l children even if more may be present. Essentially, the witness tree looks like a tree of sub-trees linked by contributing paths. Each subtree has l^h "children" subtrees and no node or edge is repeated. The height of this tree in terms of number of subtrees is l. View all edges in this tree as undirected.

For large enough l, we will show that such a witness tree cannot exist with high probability.

Lemma 2.4.1. *Assuming no cycles are encountered while constructing the witness tree, probability that such a witness tree exists for* $l > \frac{\log\log n}{h\log(\log\log n/h)} + O(1)$ *is at most* $O(1/n^c)$*, where* c *is any given constant.*

Proof. We will calculate the probability by multiplying the the total number of possible such trees with their individual probabilities. Note that all vertices, except those on the contributing paths, have at least l children.

Ways of choosing l children: For a given node, number of ways of choosing these children is $\binom{n}{l}$; number of ways of assigning edges is at most n^l; and the probability of realizing an assignment of edges is at most $(\frac{2}{n^2})^l$. So the total probability of a given node having l children is $\binom{n}{l}n^l(\frac{2}{n^2})^l < (\frac{2e}{l})^l$.

Ways of choosing a contributing path: As for the other nodes, these can only be on contributing paths of length at most h from a node to its contributing-peer. As pointed earlier, all such contributing paths of length at most h must branch off the subtree they originate from. For a given contributing path this branching off point can be chosen in at most l^h ways.

Number of ways of choosing the rest of the path of length at most h weighted by probability $\leq$ (number of ways of choosing h vertices) $\times$ (number of ways of choosing h edges) $\times$ (probability of these edges falling in the right place) $\leq n^h n^h (\frac{2}{n^2})^h \leq 2^h$ So, total number of ways of choosing a contributing path weighted by probability is at most $2^h l^h = (2l)^h$.

Total probability: Each subtree has at least l^{h-1} nodes that have l children each, and each subtree is rooted at one contributing path. So number of ways of choosing each subtree weighted by probability is $(\frac{2e}{l})^{ll^{h-1}}(2l)^h \leq (\frac{4e}{l})^{l^h}$.

Total number of such subtrees is at least $l^{(l-1)h}$. So total number of ways of choosing witness trees weighted by probability is $(\frac{4e}{l})^{l^h l^{(l-1)h}} = (\frac{4e}{l})^{l^{lh}}$. We need to choose l such that this probability is $o(1/n^c)$. This is achieved by setting l to $\frac{\log\log n}{h\log(\log\log n/h)} + O(1)$ □

So far we have assumed that the construction of the witness graph does not encounter any cycle producing edges. We will prove that it is very unlikely that it has too many edges that lead to cycles. Again as in section 2.3, using lemma 2.3.7 we argue that instead of starting with a node of load $6l$, if we start with a node of load $6l + 5c$ and attempt to construct the witness tree to a recursion depth of $l + 1$, it is very unlikely to encounter more than $5c$ cycle producing edges. Since the node under the root contributing-peer has more than $5c$ children, at least one of them must be such that the witness graph construction under that node is free of cycle-producing edges, giving the desired result. This proves the following theorem.

Theorem 2.4.1. *By searching to a depth h, with high probability,* $1 - O(1/n^c)$*, an insert will not lead to a load of more than* $6\frac{\log\log n}{h\log(\log\log n/h)} + O(1)$*, for any constant c.*

Note that by setting $h = O(\log\log n)$, we get that the maximum load is a constant, which is consistent with theorem 2.3.1. We will now show that this holds true even without the assumption that the hash functions used are truly random.

2.4.1 Using $c\log n$-Universal Hash Functions.

We will now argue that theorem 2.4.1 holds even if we use $c\log n$-universal ($c\log n$-way independent) hash functions instead of truly random hash functions. The essential idea is to extend the argument of the low probability of existence of a witness tree. Our argument in section 2.4 was along the following lines: we outlined the "shape" of the witness tree and showed that the sum of the probabilities over all such trees in the random graph G is negligible. Since the witness trees we construct are of size $l^{lh} = O(\log n)$, and the new hash functions we use are $c\log n$-way independent, for some large enough constant c, the probability of finding a given witness tree (of size at most $c\log n$) in the graph is the same as before. So the earlier bound on the total probability of finding any witness tree, obtained by summation, still holds. Next, we used lemma 2.3.7, to handle the case when we find cycle producing edges while constructing the witness graph. Lemma 2.3.7 makes use of lemma 2.3.6 which is still true with $c\log n$-way independent hash functions as lemma 2.3.6 is only concerned with $O(\log n)$ edges in subgraphs of size $O(\log n)$. So we have shown:

Corollary 2.4.1. *The guarantees stated in theorem 2.4.1 holds even if* $c\log n$*-universal hash functions are used instead of truly random hash functions, for some large enough constant c.*

Chapter 3

Balanced Allocation on Graphs

3.1 Introduction

In the last chapter we saw how the analysis of balls and bins can be applied to hashing and online load balancing. To review the main results in this area, recall the well known result that if n balls are randomly thrown into n bins, with high probability, the bin with maximum load contains $(1 + o(1))\frac{\log n}{\log\log n}$ balls [61]. Azar, Broder, Karlin, and Upfal [2] showed that instead of choosing one bin, if $d \geq 2$ bins are chosen at random, the maximum load reduces drastically to $\frac{\log\log n}{\log d} + O(1)$ (figure 3.1). Vöcking [112] showed that asymmetry helps in load balancing (figure 3.2); when bins have equal size, if ties are broken to the left, then the maximum bin size drops to $\frac{\log\log n}{d \ln \phi_d} + O(1)$ where ϕ_d is a constant > 1 that approaches $\ln 2$ for large d. He also showed that the above bound is tight, that is, the maximum load is at least $\frac{\log\log n}{d \ln \phi_d} - O(1)$ even if the d bins are chosen from an arbitrary fixed distribution, and irrespective of the policy used to decide the bin out of the d choices for a given ball. Berenbrink *et al.* [12] extended these results to the case when the number of balls m is greater than the number of bins n showing that the maximum height is at most $\frac{\log\log n}{\ln d} + O(1)$ above the average when d bins are chosen at random and similar results when ties are broken asymmetrically. As we saw in the last chapter, the maximum load can also be reduced by moving previously inserted balls into their alternate bin choice(s) when a new ball is inserted [103, 55]; with two choice load balancing, by performing at most h moves per insert, we can maintain a maximum load of $O(\frac{\log\log n}{h \log(\log\log n/h)})$ for n balls and n

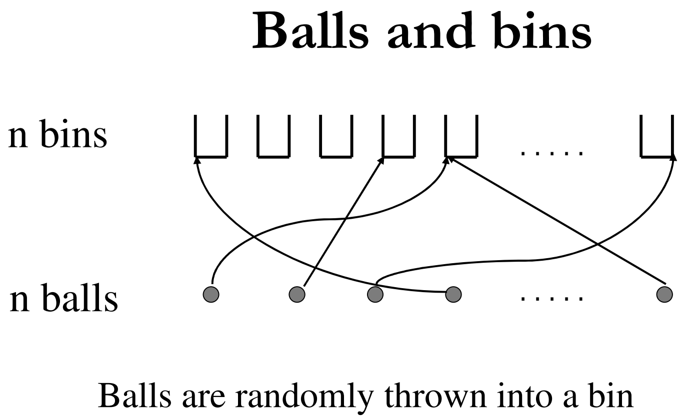

Figure 3.1: Balls and bins: If each ball is thrown into a random bin, the max load is $(1+o(1))\ln n/\ln\ln n$. Instead if each ball is inserted into least loaded of d randomly chosen bins the max load drops exponentially to $\frac{\log\log n}{\log d} + O(1)$

bins [99]. Further there has been a lot of work on parallel balls and bins where balls are thrown parallelly into bins in rounds [3, 34, 47].

A natural extension of the earlier work is to consider a graph-based model where balls are not allowed to choose any two random bins but only bins that are connected by an edge in a given underlying graph. While the earlier studies use a complete underlying graph, an understanding of balls and bins processes over arbitrary graphs is not only theoretically interesting but also meaningful in pratical scenarios. For example, if bins are arranged in a line, what if balls are only allowed to choose two random bins that are close-by, say at most distance Δ apart – here the underlying graph has edges only when the bins are at most distance Δ apart. How small can Δ be so that the earlier results continue to hold? This is useful from a practical point of view because the cost of accessing the two bins may depend on the distance between them. For example, if the bins represent hash buckets of a hash table stored on disk, then after having accessed the first bucket, the time required to access the second bucket depends on how far the head has to move from the first to the second bucket, as the access time is dominated by the seek time. Or, it may be easier to read memory in large bursts and accomplish the read of the two buckets that are at most Δ apart in one random access. Similarly when d random bins are explored, it may be beneficial to group the accesses into as few bursts as possible.

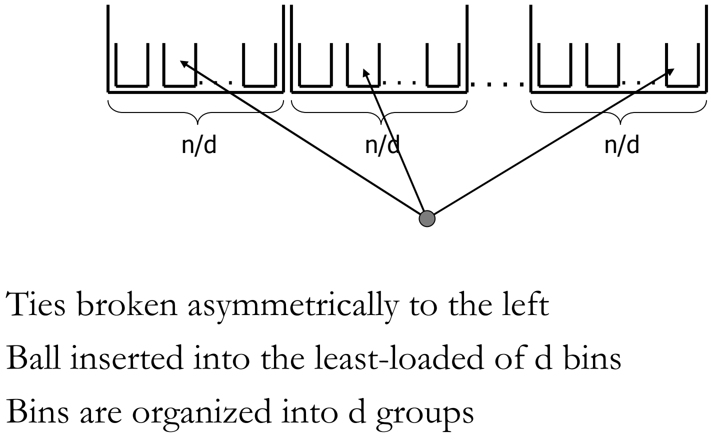

Figure 3.2: Asymmetric load balancing: If ties are broken asymmetrically, say to the left, the max load drops further to $O(\frac{\log\log n}{d})$. This means even for d as small as $\log\log n$ will result in a constant max load.

Another example could be the load-balancing of requests in a network setting. To apply the two-choice heuristic, a client may make a request to a server. The server may then query another random server and the client may be serviced by the least-loaded of the two servers. However it may be inefficient for the server to query any other server; perhaps it only knows about a few servers, or it is close to only a few servers, or only a few other servers may be able to service the client's request. So instead it may be connected to Δ neighboring servers and could only query one of these at random. This amounts to the following question: If each ball picks a random bin and that bin again selects a random "neighboring" bin and the ball is inserted into the least-loaded of the two bins, then what is the maximum load? This is equivalent to picking a random edge in the graph for each ball. For what graphs, do the results for two choices continue to hold? First we show a lower bound of $\Omega(\log\log n + \frac{\log n}{\log(\Delta \log n)})$ for maximum bin size where Δ is the average degree of the graph; in particular if the average degree is n^ϵ for some constant ϵ, then the lower bound implies a maximum load of $\Omega(\log\log n + 1/\epsilon)$. However, if the graph is not regular, we show that the maximum bin size can be much higher; even for graphs with minimum degree n^ϵ, it can be $\Omega(\log n/\log\log n)$. On the other hand, for Δ-regular graphs, the maximum bin size is $\log\log n + O(\frac{\log n}{\log(\Delta/\log^4 n)}) + O(1)$, which almost matches the lower bound (Section 3.2). This means that the earlier bound on the maximum load with two choice load balancing (on a complete graph) continues to hold on almost regular graphs with degree as low as n^{ϵ} for any constant ϵ – the maximum load is $\log\log n + O(1/\epsilon) + O(1)$. So even if the graph has degree only $n^{\Omega(1/\log\log n)}$, the maximum load is still $O(\log\log n)$. We also show that when the number of balls m is more than the number of bins n, the earlier results by Berenbrink *et al.* [12] do not carry over; even for graphs of degree n^{ϵ} where ϵ is constant, the maximum load can be $m/n + \omega(\log\log n)$. We achieve similar results when moves are allowed (Section 3.3).

Further, Vöcking's observation that the maximum load of $O(\frac{\log\log n}{d})$ can be achieved by breaking ties asymmetrically requires d random bin choices. Is it possible to achieve similar bounds when the d choices may not be random, but preferably close together or in a few groups where bins within a group are close-by? Such an access pattern may be more efficient especially when bins represent memory locations in a hash table. We show that such bounds can be achieved by making two random accesses and querying $d/2$ contiguous bins in each access, even if ties are not broken asymmetrically. By partitioning a sequence of n bins into $2n/d$ groups, each of $d/2$ consecutive bins, if each ball chooses two groups at random and inserts the new ball into the least-loaded bin in the lesser loaded group, then the maximum load is $2\log\log n/d + O(1)$ with high probability (Section 3.4). This is surprising as it shows that querying d bins in two groups of size $d/2$ outperforms querying d random bins! Furthermore, it also turns out that this partitioning into aligned groups of size $d/2$ is also essential in achieving this bound. Instead of choosing two aligned groups, if we simply choose two disjoint but possibly unaligned random sets of $d/2$ consecutive bins – which only seems like a simple, natural variation – then the maximum load jumps to $\frac{\log\log n}{\log d}$! Similarly we also show that it is important to insert into the lesser loaded group. In fact, we can get arbitrarily close to Vöcking's bound of $\frac{\log\log n}{d\ln\phi_d} + O(1)$ by using c random groups of size d/c each and breaking ties asymmetrically. Moreover, our result implies that we can achieve constant load by making two burst accesses of $\log\log n$ bins each, provided these accesses are made from a set of disjoint groups; otherwise the maximum bin size becomes $\frac{\log\log n}{\log\log\log n}$.

3.2 Two choice Load Balancing on Δ-regular Graphs

One method to obtain bounds on maximum load with two choice load balancing is to use the layered induction technique [2] that recursively bounds the fraction of bins, p_i with load i. For a new ball to fall at a height of $i + 1$ (that is, into a bin with at least i balls), both of its bin choices must have at least i balls, which happens with probability at most p_i^2. This essentially implies that $p_{i+1} \approx p_i^2$, implying a quadratic drop with height, giving a maximum load of $\log \log n$ with high probability. This technique fails on arbitrary graphs, because the probability that both the bin choices for a subsequent ball insert have load at least i is no longer p_i^2 as the two bins are not chosen independently and randomly. Alternatively the witness tree method tracks the occurrence of a bin with a high load using a suitable tree of events and shows that such a tree is unlikely to occur. This type of analysis was introduced in the context of PRAM simulations [34, 91] and later adapted for balls and bins [33, 28, 112]. Using this method, we analyze the maximum load when the underlying graph is Δ-regular.

First we prove a lower bound on the maximum load for a Δ-regular graph.

Theorem 3.2.1. *Given a Δ-regular graph with n nodes representing n bins, if n balls are thrown into the bins by choosing a random edge and placing into the smaller of the two bins connected by the edge, then the maximum load is at least* $\Omega(\log \log n + \frac{\log n}{\log(\Delta \log n)})$ *with high probability of* $1 - 1/n^{\Omega(1)}$.

Proof. The $\Omega(\log \log n)$ term in the lower bound follows from Vöcking's lower bound [112] for placement of n balls into n bins if each ball picks two bins at random using any arbitrary but fixed distribution. The second term follows by analyzing the ratio of the number of balls to the number of edges: since there are n balls and $n\Delta/2$ edges, with high probability, some edge will get $\frac{\log n}{\log(\Delta \log n/2)}$ balls (this is like throwing n balls into $n\Delta/2$ bins). Hence at least one of the endpoints of this edge will get $\frac{\log n}{2\log(\Delta \log n/2)}$ balls. □

This implies that if the bins are arranged in a line and if each ball chooses two bins that are at most distance Δ apart, the maximum load is at least $\Omega(\log \log n + \frac{\log n}{\log(\Delta \log n)})$.[1] In

[1] While the lower bound applies to the algorithm that always picks two random bin choices at most distance Δ apart when bins are arranged in a line, it is easy to generalize to any algorithm that chooses two bins for every ball that are distance Δ apart in expectation.

particular, when Δ is *polylog*(n) the maximum bin load is $\Omega(\frac{\log n}{\log\log n})$. If $\Delta = n^\epsilon$ for some constant ϵ, then the lower bound implies a maximum load of $\Omega(\log\log n + 1/\epsilon)$.

The following theorem gives an almost matching upper bound on the maximum load for Δ-regular graphs.

Theorem 3.2.2. *Given a Δ-regular graph with n nodes representing n bins, if n balls are thrown into the bins by choosing a random edge and placing into the smaller of the two bins connected by the edge, then the maximum load is* $\log\log n + O(\frac{\log n}{\log(\Delta/\log^4 n)}) + O(1)$ *with high probability of* $1 - 1/n^{\Omega(1)}$.

This holds even if the graph is almost regular, that is, each node has degree, $\Theta(\Delta)$.

Corollary 3.2.1. *Given a n^ϵ-regular graph with n nodes representing n bins, if n balls are thrown into the bins by choosing a random edge and placing into the smaller of the two bins connected by the edge, then for any $\epsilon > 8\log\log n/\log n$, the maximum load is* $\log\log n + O(1/\epsilon) + O(1)$ *with high probability of* $1 - 1/n^{\Omega(1)}$.

The basic idea behind the witness tree method is to start with a ball at a large height and to construct a shallow tree with about $\log n$ nodes and depth about $\log\log n$ where nodes are the bins and an edge represents the two bin choices of a ball and argue that such a tree is unlikely to exist. While the construction as described may not necessarily produce a tree, we show later that the graph obtained has only a few extra edges in addition to a tree structure.

Construction of the witness graph: The root of the witness graph is the node with load $l+c$ where c is a constant. For each of the top l balls in this node, there must be an alternate bin choice. These alternate bin choices are set to be the children of the root node. The edges are labeled by the corresponding balls. Similarly we recurse for each of the l children. For a parent node with load x, the i^{th} ball from the top at height $x - i + 1$ must have had an alternate bin choice with load at least $x - i$; this is the i^{th} child of the parent node. The i^{th} child, which has load at least $l - i + c$, will be expanded down further to $l - i$ children corresponding to the alternate bin choices for its top $l - i$ balls. We continue this recursion as long as a node has load greater than c and stop when it equals c which corresponds to the leaf nodes. We also label each node (bin) with the set of lowest c balls. These balls are distinct from the balls corresponding to the edges incident on this node. Clearly, this witness graph must be a subgraph of the original underlying graph on the bins.

It is possible that during this process, we run into cycles, that is, the child of some node may be an already existing node in the tree. For now, let us assume that this does not happen and the graph constructed is truly a tree. We will later argue that there cannot be too many cycle producing edges. Intuitively this is true because we are unlikely to run into cycles when a small number of $\log n$ nodes are explored as edges are chosen randomly.

Assuming that no cycles are found, after this process, we have a tree where each node with load $i + c$ has i children with loads $i - 1 + c, i - 2 + c, \ldots, c$ respectively. Each edge corresponds to a distinct ball and each node is associated with c additional distinct balls. Each edge indicates a distinct ball whose bin choices are exactly the end points of the edge. It is easy to check inductively that the total number of nodes in this tree is exactly 2^l (the recurrence relation for the number of nodes is $f(l) = f(l-1) + \ldots + f(0) + 1$ where $f(0) = 1$).

We now show that there cannot exist a set of $\log n$ nodes connected by a tree where each edge represents a distinct ball and further each node has a constant c number of distinct additional balls.

Lemma 3.2.1. *The probability that there exists a set of* $\Omega(\log n)$ *bins connected by a tree where each edge represents a distinct ball and further each node has an additional set of* c *distinct bins is at most* $1/n^{\Omega(1)}$

Proof. We will bound the probability of existence of such a tree by counting the number of such possible trees and multiplying by the probability that a given tree exists. The total number of different "shapes" (two shapes are the same if they are isomorphic) for a rooted tree on m nodes is at most 4^m [80].

Choosing bins and balls for nodes and edges: For a given shape, the root can be chosen in n ways. Once a node has been chosen, a given child can be chosen in Δ ways. Hence the total number of ways of choosing all the nodes is at most $n\Delta^{m-1}$. For each of the $m - 1$ edges, the ball can be chosen in at most n ways and the probability that it falls in this edge out of the total of $n\Delta/2$ choices is $2/n\Delta$. So, for a given shape, the total probability multiplied by the number of ways of choosing nodes and edges in the tree is at most $n\Delta^{m-1} n^{m-1} (2/n\Delta)^{m-1} = n2^{m-1}$.

Choosing additional c balls per node: In addition, for a given set of m nodes, we need c distinct balls to be associated with each node. This means that there are a total of cm balls that are associated with nodes in the tree, that is, each of these balls is chooses some node in the tree. These cm balls can be chosen in $\binom{n}{cm}$ ways. Since each node has Δ edges incident on it, the m nodes of the tree have a total of $m\Delta$ edges and each of these cm balls must choose one of these edges. This probability is at most $\binom{n}{cm}(\frac{m\Delta}{n\Delta})^{cm} \leq (e/c)^{cm}$.

Putting it all together, the total probability of finding such a tree is at most $4^m n2^{m-1}(e/c)^{cm} \leq n[8(e/c)^c]^m$. The idea is to choose c to be a large enough constant so that this probability is $1/n^{\Omega(1)}$; this can be achieved by setting m to $\Omega(\log n)$. □

Remark 3.2.1. *While the witness tree arguments in prior work [90, 112] do not look at all possible shapes of trees on m nodes, we use this approach as it extends easily to the witness tree analysis in the case when balls moves are allowed (Section 3.3).*

Since a witness tree with root of load $l + c$ has 2^l nodes, for $l = \log\log n + \Omega(1)$ the number of nodes becomes $\Omega(\log n)$; by Lemma 3.2.1, this is unlikely with high probability. So far we have assumed that the witness graph is a tree, ignoring the possibility of having cycles. We will show that the probability of finding many cycle-producing edges is very low. During the construction of the witness graph, an edge that leads to an existing node is called a cycle-producing edge (such edges are called *pruning* edges in past work [90, 112]). Let p be the number of cycle-producing edges.

The next lemma shows that there cannot exist a witness graph with too many cycle-producing edges.

Lemma 3.2.2. *The probability that there exists a set of $O(\log^2 n)$ bins connected by a tree where each edge represents a distinct ball, and there are additional $p = \frac{k \log n}{\log(\Delta/\log^4 n)}$ edges between these nodes representing distinct bins, and further each node has an additional set of c distinct bins is at most $1/n^{\Omega(1)}$ for sufficiently large constants k and c.*

Proof. As in proof of Lemma 3.2.1, we will bound the probability by counting the number of ways of choosing the balls, bins and edges and multiplying by the probability of each case. Let us consider a tree with m nodes and p additional edges. For a given shape of the tree, the additional p edges can be chosen in at most m^{2p}.

For each of the p additional edges, balls can be chosen in n ways and the probability that the ball chooses the edge is $2/n\Delta$, giving a total probability of $n^p(2/n\Delta)^p = (2/\Delta)^p$.

As in proof of Lemma 3.2.1, the number of ways of choosing shapes for the tree, bins for the nodes of the tree and balls for the edges and c balls for each node, multiplied by the probability of each case is at most $n[8(e/c)^c]^m$.

So the total probability of finding a witness graph with m nodes and p cycle-producing edges is at most $m^{2p}(2/\Delta)^p n[8(e/c)^c]^m = (2m^2/\Delta)^p n[2(e/c)^c]^m$.

If $p = \Omega(\frac{\log n}{\log(\Delta/\log^4 n)})$, then this probability is $1/n^{\Omega(1)}$. □

We are now ready to prove the main theorem.

Proof of Theorem 3.2.2: To take into account the possibility of cycle-producing edges, instead of starting with a node of load $l + c$, we start with a node of load $l + c + p$ where $p = k\frac{\log n}{\log(\Delta/\log^4 n)}$. At the first level, we expand to p children corresponding to the top p balls. The i^{th} child of the root will have $l+p-i+c$. For this child, we treat $c' = p-i+c$ and expand as before till every leaf has load exactly c', giving exactly 2^l nodes in this subtree (where nodes may be repeated). The total number of nodes, $M = 2^l p+1 = p\log n+1 \leq k\log^2 n+1$ for $l = \log\log n$. The probability that there are more than p cycle-producing edges is at most $1/n^{k-1}$. If there are less than p cycle-producing edges, then during the construction of the witness graph, for at least one of the p children, the construction below it is free of cycle-producing edges. This gives a witness tree where the root node has load at least $l+c$. By Lemma 3.2.1, the probability of this occurrence is at most $1/n^{\Omega(1)}$. So the probability of finding a witness graph whose root node has load $\log\log n + p + c$ is at most $1/n^{\Omega(1)}$. □

We also show that this result does not hold for non-regular graphs even if the minimum degree is Δ. The counter-example is a complete bipartite graph with $n - \Delta$ nodes on one side and Δ on the other.

Lemma 3.2.3. *There exist graphs with minimum degree n^ϵ for any constant $\epsilon < 1$ such that after insertion of n balls by the same process as above, the maximum load of a bin is $\Omega(\frac{\log n}{\log\log n})$ with high probability of* $1 - 1/n^{\Omega(1)}$.

Proof. Consider the complete bipartite graph with $n - \Delta$ nodes on one side (left) and Δ on the other (right). We break the n inserts into $\log n$ phases of $n/\log n$ inserts each. The essential idea is to show that for most of the early phases, after i phases, each of the Δ nodes on the right side has load at least i and at least $1/(4\log n)^i$ fraction of the left side vertices has load i as long as $n/(4\log n)^{i+1} > \Delta \log n$. We prove this inductively. Assuming it is true at the end of the i^{th} phase, in the first $n/(2\log n)$ inserts in the $(i+1)^{st}$ phase, at least $n/(4\log n)^{i+1}$ of them in expectation choose the left vertex to be of load at least i. This by assumption is at least $\Delta \log n$ and hence, if ties are broken to the right, then with high probability, all the Δ vertices on the right side would have been chosen and hence would have incurred a load of at least $i+1$. For the remaining $n/(2\log n)$ inserts of the phase, in expectation, at least $n/(4\log n)^{i+1}$ of the left vertices with load i are chosen and they will all incur achieve a load of at least $i+1$. This goes on as long as $n/(4\log n)^i > \Delta \log n$ which means that the maximum load is at least $i = \frac{\log(n/(\Delta \log n))}{\log\log n}$. For $\Delta = n^\epsilon$, this is $\Omega(\frac{\log n}{\log\log n})$. All expectation results can be shown to hold with high probability using Chernoff bounds. □

We now show that when the number of balls m is more than the number of bins n, the earlier results by Berenbrink *et al.* [12] do not carry over; even for graphs of degree n^ϵ where ϵ is constant, the maximum load can be $m/n + \omega(\log\log n)$.

Lemma 3.2.4. *There exist n^ϵ-regular graphs for any constant $\epsilon < 1$ such that after insertion of m balls by picking edges at random, the maximum load of a bin is $m/n + \omega(\log\log n)$ with high probability of $1 - 1/n^{\Omega(1)}$ for sufficiently large m.*

Proof. The graph we use is a collection of $n^{1-\epsilon}$ cliques of size n^ϵ. Some clique must get $m/n^{1-\epsilon} + \sqrt{m \ln n^{1-\epsilon}/n^{1-\epsilon}}$ balls when $m \geq n \ln n$ as this is like throwing m balls into $n^{1-\epsilon}$ bins [107]. Hence some bin in this clique must get $m/n + \sqrt{m \ln n^{1-\epsilon}/n^{1+\epsilon}}$ balls. The result follows by setting $m = \Omega(n^{1+\epsilon})$. □

3.3 Balls and bins with moves

We saw in the last chapter that if moves are allowed during inserts of balls then the maximum load can be reduced further [103, 55]. On a complete graph, by performing at most h moves per insert, we can maintain a maximum load of $O(\frac{\log\log n}{h\log(\log\log n/h)})$ [99]. In particular by performing up to $\log\log n$ moves we achieve a constant maximum load.

We now extend these bounds on maximum bin size with at most h moves per insert to Δ-regular graphs. We again use the same method as in section 3.2.2 but using a different witness tree. We use the same witness tree as described in [99] (or see Appendix 3.5.2). Starting from a node with load $4l + c$, we get a witness tree with l^{th} nodes assuming no cycles are found during the construction. Further each node has a set of additional c balls. So for $l = \Omega(\frac{\log\log n}{h \log(\log\log n/h)})$ the number of nodes becomes $\Omega(\log n)$.

Lemmas 3.2.1 and 3.2.2 continue to hold as they only refer to random choices of edges for balls which are same as before. So again there cannot be a witness tree with $\Omega(\log n)$ nodes or a witness graph with more than $p = k\frac{\log n}{\log(\Delta/\log^4 n)}$ cycle-producing edges. Again as before if we start with a node with load $4l + c + k\frac{\log n}{\log(\Delta/\log^4 n)}$ then at least one of the children of the root must be such that the witness graph construction under that node is free of cycle-producing edges, giving the following desired result.

Theorem 3.3.1. *Given a Δ-regular graph with n nodes representing n bins, if n balls are thrown into the bins by choosing a random edge for each ball and performing at most h moves as described above, then the maximum load is* $O(\frac{\log\log n}{h\log(\log\log n/h)} + \frac{\log n}{\log(\Delta/\log^4 n)})$ *with high probability of* $1 - 1/n^{\Omega(1)}$.

3.4 Choosing bins in groups

Vöcking [112] observed that asymmetry helps in load balancing and showed that if ties are broken to the left when n balls are inserted into n bins with d random choices per ball, the maximum load is $\frac{\log\log n}{d \ln \phi_d} + O(1)$ where ϕ_d approaches $\ln 2$ for large d.[2] This is a significant improvement over the $\frac{\log\log n}{\log d}$ bound. He also showed that the above bound is tight, that is, even if the d bins are chosen from an arbitrary fixed distribution, the maximum load is at least $\frac{\log\log n}{d \ln \phi_d} - O(1)$ irrespective of the policy used to decide the bin out of the d choices for a given ball.

We achieve a similar bound without making d random bin choices as in Vöcking's result. Our algorithm makes two random accesses and chooses $d/2$ consecutive bins from each access. If the bins represent memory locations in a hash table then two burst accesses

[2] $\phi_d = \lim_{k\to\infty} \sqrt[k]{F_d(k)}$ where $F_d(k)$ is a generalization of Fibonacci sequence defined recursively as $F_d(k) = \sum_{i=1}^{d} F_d(k-i)$ and $F_d(k) = 0$ for $k \leq 0$ and $F_d(1) = 1$.

may be more efficient than d random accesses. The maximum load of any bin is $\frac{\log\log n}{d \ln \phi_2} + O(1) = \frac{1.38 \log\log n}{d} + O(1)$.

Our algorithm works as follows. Group the n bins into disjoint groups of $d/2$ consecutive bins each. Call these super-bins. Pick two super-bins at random and select the super-bin with the lesser total load (breaking ties to the left) where the total load of a super-bin is the sum of the loads of its $d/2$ bins. Then place the ball into the least-loaded of the $d/2$ bins in the super-bin.

Theorem 3.4.1. *If n balls are inserted into n bins by the above algorithm, the maximum load of any bin is $\frac{\log\log n}{d \ln \phi_2} + O(1)$ with high probability. Instead of using two random groups of size $d/2$, if we use $c \geq 2$ groups of size d/c, then the maximum load of any bin is $\frac{\log\log n}{d \ln \phi_c} + O(1)$.*

Note that this approaches Vöcking's bound as c becomes large. Moreover, our result implies that we can achieve constant load by making two burst accesses of $\log\log n$ bins each.

Furthermore, it also turns out that this grouping into groups of size $d/2$ is essential in achieving this bound. It would seem that the bound of Theorem 3.4.1 would continue to hold even if all bins are not grouped into super-bins of size d, but instead the algorithm picks two random bins at least d apart and d consecutive bins starting from each of the chosen bins. Surprisingly this turns out to be false, i.e., the distribution becomes much worse if instead of choosing from well-aligned groups, the algorithm is allowed to choose two random disjoint sets of d consecutive bins. For $d = \log\log n$, instead of getting a constant load, we get a maximum load of $\frac{\log\log n}{\log\log\log n}$.

Theorem 3.4.2. *Instead of grouping the n bins into aligned groups of size $d/2$, if we pick two disjoint random sets of $d/2$ consecutive bins, then the maximum load is $(1 - o(1))(\frac{\log\log n}{\log d})$ with high probability at least $1 - o(1)$. As before, the algorithm picks the set with the smaller total load and places the ball into the least loaded bin in the selected set. A random set of $d/2$ consecutive bins may be chosen by first choosing a random bin and picking the $d/2$ bins to its right.*

Similarly it turns out that it is also important to choose the lesser loaded of the two chosen super-bins, that is, even with aligned bins, if the algorithm inserts into the least

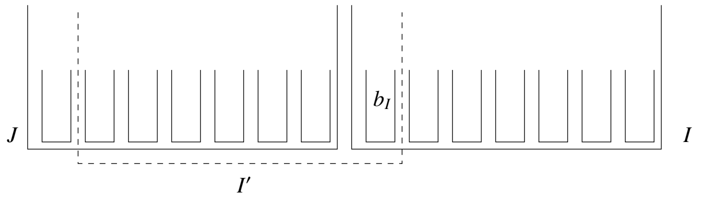

Figure 3.3: Unaligned super-bins: I and J are aligned, I' is unaligned.

loaded bin in the two super-bins without considering the total load of the super-bins, then the maximum load of a bin becomes higher.

Theorem 3.4.3. *If we place the ball into the least-loaded of the d bins (that is, both the super-bins put together) in the above scheme without looking at the total loads of the each of the super-bins, then the maximum load of any bin is* $(1 - o(1))(\frac{\log\log n}{\log d})$ *with high probability at least* $1 - o(1)$.

We now prove Theorem 3.4.1 by analyzing super-bin loads using two-choice load-balancing results.

Proof of Theorem 3.4.1: Since each insert into a super-bin always goes into the least-loaded bin, the loads of any two bins in a super-bin differ by at most 1. Hence it is sufficient to analyze the maximum load of a super-bin. This is equivalent to throwing n balls into $2n/d$ bins by choosing two bins at random, breaking ties to the left. Berenbrink *et al.* [12] extended Vöcking's result when the number of balls m' is greater than the number of bins n' showing that the maximum bin size is at most $\frac{m'}{n'} + \frac{\log\log n'}{d \ln \phi_d} + O(1)$. Hence the maximum load of a super-bin is $\frac{d}{2} + \frac{\log\log(2n/d)}{2 \ln \phi_2} + O(1)$. Thus the maximum load of any bin is at most $2 + \frac{\log\log n + O(1)}{d \ln \phi_2}$. Similarly if the super-bins are of size d/c, the maximum load of any bin is at most $\frac{\log\log n}{d \ln \phi_c} + O(1)$. □

We now prove Theorem 3.4.2. The intuition behind why the load increases when the groups are not aligned is the following: Let us artificially group the bins into aligned super-bins of size d although this is not what the algorithm does. Let us estimate the fraction of bins with load at least $i + 1$, p_{i+1} recursively. Earlier for a super-bin I to grow from i to $i + 1$, the second super-bin that was chosen must have at least i balls. This is no more true

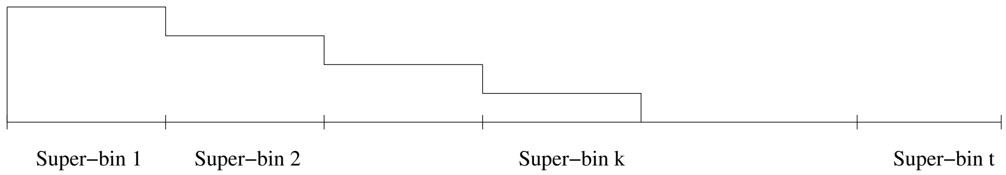

Figure 3.4: k-step: all bins in the i^{th} super-bin have load $k-i$ if $i \geq k$ and 0 otherwise.

if super-bins need not be aligned. Consider two adjacent aligned super-bins I and J with loads i and $j = \lfloor i/d \rfloor d$ respectively. If $j = kd$, all bins in I have load either k or $k+1$ and all bins in J have load k (see Figure 3.3). If i is not a multiple of d, then it is possible that the bin b_I in I next to J has load k. If the algorithm chooses the set I' of d consecutive bins with the last bin b_I in I and the adjacent $d-1$ bins from J, then note that load of I' is j. If the algorithm chooses to insert into super-bin I', it could very well place the new ball in b_I, as all bins in I' have equal load. This event happens if the other super-bin chosen had load at least j. The intuition is that since j could be as low as $i-d$ and probabilities drop quickly, the probability that the d bins next to I have load k each and that I' is one of the super-bins chosen by the algorithm and the other super-bin chosen has load at least j is much larger than p_i^2. Essentially the second super-bin need not have load i; it could have load j that could be as small as $i-d+1$. Informally speaking, the probability that the super-bin J has load j is p_j; the probability that the second super-bin chosen has load at least j is also roughly p_j; probability that the super-bin I has load i is p_i. So the probability that the load of I increases by one in a step could be about $p_i p_j p_j$, which is much lower than p_i^2, as j can be as small as $i-d$. While this reasoning is not formal, to convert this into an inductive proof, we will need the neighboring super-bin of J to have load $j-d$ and its neighboring super-bin to have load $j-2d$ and so on.

For ease of exposition, we will use groups of size d instead of $d/2$. With a total of n bins, we have n/d super-bins. Let us group super-bins into n/td *blocks* of t consecutive super-bins each. (We will later set t to be about $4d \log n$.)

Definition 1. *A block of super-bins is a k-*step *if the first super-bin has all bins with load k, the next has all bins with load $k-1$, and so on and the last $t-k$ super-bins have all empty bins. Formally all bins in l^{th} super-bin in the group have load $max(0, k-l)$ (Figure 3.4).*

We will track the number of k-steps in our bin configurations as inserts proceed in rounds of n/t inserts. Let q_r denote the fraction of blocks that are r-steps after r rounds of n/t inserts each. The following lemma provides a lower bound on q_r recursively.

Lemma 3.4.1. *Expected value of* $q_r \geq (\frac{1}{10t^3d^2})^{rd} \cdot \prod_{i=0}^{r-1} q_i^d \cdot q_{r-1}$

Proof. We will bound the fraction of $r-1$-steps that become r-steps after a round of n/t inserts, only considering r-steps that are formed from $r-1$-steps by exactly a specific sequence of rd insertion steps. Each of the bins in the r^{th} through the first super-bin receive exactly one ball each in that order. Further the bins in the first super-bin receive balls in the left to right order. Precisely, the sequence of inserts is as follows:

1. The r-th super-bin is chosen as one of the two choices for insert and the insert ends up in this super-bin. This happens d times making all bins in this super-bin to have load 1.

2. Similarly d inserts happen in the $r-1$-st super-bin in the block causing all its bins to have load 2. Similarly for the $r-2$-nd, $r-3$-rd and so on till the second super-bin. After this, the block is almost an r-step except that the first super-bin needs to add an item to each of its bins.

3. We will insist that the last super-bin is handled exactly by the following process. Observe that all bins in the first and second super-bin have load $r-1$. Let S_m denote the sequence of d bins which are exactly the last $d-m$ bins from the first super-bin and the remaining m from the second.

   (a) First S_0 (same as the first super-bin) is chosen for an insert and the new item goes into the first bin in S_0, which clearly is one of the least populated bins, as required for insert. Next, S_1 is selected for an insert and the new item goes into the first bin in S_1. And so on till S_{d-1}. In this way, all the bins in the first super-bin have load r making the block an r-step.

Note that we require exactly these rd inserts within the block in this sequence among the total of n/t inserts.

First let us find the number of $r-1$-steps that are chosen exactly rd times in n/t inserts. Probability that a given $r-1$-step gets chosen exactly rd times is $\binom{n/t}{rd}(1/n)^{rd}(1-(t+$

$1)d/n)^{n/t-rd}$. This is $\geq (1/(trd)^{rd}e^{-2d})$. So the expected fraction of $r-1$-steps that get exactly the correct number of choices is $\geq (1/(10trd)^{rd})$.

Assuming an $r-1$-step gets chosen exactly rd times in that order, let us now estimate the probability that all the rd inserts go into the desired bins. The last d inserts go to the right bins with probability at least $\frac{q_{r-1}}{td}$. This is because there are at least $\frac{q_{r-1}n}{td}$ unaligned-super-bin choices out of the total n that have all bins of size at least $r-1$ and if the second choice is made from any of these, then it will clearly be larger than the choice in the $r-1$-step in consideration and so the insert can happen in the right bin in the $r-1$-step. Similarly the i^{th} set of d inserts go to the right bins with probability at least $\frac{q_{i-1}}{td}$.

Hence $q_r = (\frac{1}{10trd})^{rd} \cdot \prod_{i=0}^{r-1}(\frac{q_i}{td})^d \cdot q_{r-1} \geq (\frac{1}{10t^3d^2})^{rd} \cdot \prod_{i=0}^{r-1} q_i^d \cdot q_{r-1}$.

While we assumed that ties are broken in our favor, the proof goes through even if we assume that the ties are broken randomly. □

This can be converted into a high probability bound with probability at least $1-2/\log n$ (see Appendix 3.5.2), giving the slightly weaker recurrence $q_r \geq (\frac{1}{4})^d \cdot (\frac{1}{10t^3d^2})^{rd} \cdot \prod_{i=0}^{r-1} q_i^d \cdot q_{r-1}$ as long as the right hand side of this bound is at least $\frac{\log n}{n/(td)}$ and $t \geq 4d\log n$. Now we are ready to prove Theorem 3.4.2.

Proof of Theorem 3.4.2: Now that we have the recurrence on q_r, we show inductively that $q_r \geq 1/(40t^3d^2)^{(d+2)^r}$. Let $Q_r = \prod_{i=0}^{r} q_r$. Multiplying both sides of the recurrence by Q_{r-1}, we get that $Q_r \geq (\frac{1}{40t^3d^2})^{rd} \cdot (Q_{r-1})^{d+1} \cdot q_{r-1} \geq (\frac{1}{40t^3d^2})^{rd} \cdot (Q_{r-1})^{d+2}$. Inductively it is easy to check that $Q_r \geq (\frac{1}{10t^3d^2})^{(d+3)^r}$. The recurrence can be used as long as $q_r \geq \frac{\log n}{n/(td)}$ which holds as long as $Q_r \geq \frac{\log n}{n/(td)}$. This holds as long as $r \leq \log_{d+3}\log_{40t^3d^2}\frac{n}{td\log n} \leq \frac{\log\log\frac{n}{td\log n} - \log\log(t^3d)}{\log(d+3)}$. Setting $t = 4d\log n$, we get that $r = (1-o(1))(\frac{\log\log n}{\log d})$. At this value of r, there are at least $\log n$ r-steps, completing the proof. The total proabability of exceeding the bound is at most $O(r/\log n)$ which is $o(1)$. □

Next we prove Theorem 3.4.3.

Proof Sketch of Theorem 3.4.3: All the bins in a super-bin with total load i have load either exactly $\lfloor i/d \rfloor$ or $\lceil i/d \rceil$. The crucial idea is that it is no more true that in order for a super-bin I with total count i, the probability of adding an extra ball is lower than p_i^2

where p_i is the fraction of super-bins with load i. This is because if i is just smaller than a multiple of $d/2$, then any bin with load j where j is the multiple of $d/2$ just less than i could be chosen as one of the two choices along with I and the next ball assignment could still go to I. This means that $p_{i+1} \approx p_i p_{\lfloor i/d \rfloor d}$. Analyzing this recursion gives the result. A formal proof can be obtained by using $\log n$ rounds of $n/\log n$ ball inserts as in the proof of Lemma 3.2.3. □

3.5 Micellaneous technical details

3.5.1 High probability version of Lemma 3.4.1

Here we convert the expected value of q_r to a high probability bound for q_r.

The essential step is bounding the fraction of $r-1$-steps that receive exactly rd inserts in the right order. While we proved earlier that the expected fraction is $\geq (1/(10trd)^{rd})$, we now show that the deviation from this mean is small and hence Chebyshev inequality can be applied to obtain high probability bound. Let X_i denote the indicator variable for the event that the i^{th} $(r-1)$-step gets the desired sequence of inserts.

Lemma 3.5.1. *$P(X_iX_j) \leq (1+\epsilon)P(X_i)P(X_j)$ where $\epsilon = 4d/t$.*

Proof. $P(X_i) = P(X_j) = \binom{n/t}{rd}(1/n)^{rd}(1-(t+1)d/n)^{n/t-rd}$

$P(X_iX_j) \leq \binom{n/t}{rd}\binom{n/t-rd}{rd}(1/n)^{2rd}(1-2td/n)^{n/t-2rd}$

Comparing the terms, we get that $\frac{P(X_iX_j)}{P(X_i)P(X_j)} \leq \frac{\binom{n/t-rd}{rd}}{\binom{n/t}{rd}} \frac{(1-2td/n)^{n/t-2rd}}{(1-(t+1)d/n)^{2(n/t-rd)}}$

After simplification, this is less than $(1+4d/t)$. □

Lemma 3.5.2. *For any random variable $R = \sum X_i$, if $P(X_iX_j) \leq (1+\epsilon)P(X_i)P(X_j)$, then the variance of R, $var(R) \leq \mu + \epsilon\mu^2$, where μ denotes $E[R]$.*

Proof. For any random variable $R = \sum X_i$, variance of R equals $E[(\sum X_i)^2] - E[\sum X_i]^2 \leq \sum E[X_i^2] + 2\epsilon \sum E[X_i]E[X_j] \leq \sum E[X_i] + \epsilon(\sum E[X_i])^2 = \mu + \epsilon\mu^2$ □

Let μ denote the expected number of $(r-1)$-steps that receive the desired rd inserts. So the probability that less than $\mu/2$ get the desired rd inserts is at most $var(R)/(\mu^2/4) \leq 1/\mu + \epsilon$.

Assuming $q_r \frac{n}{td} \geq \log n$ and $t \geq 4d \log n$, this probability is at most $2/\log n$. So with probability at least $(1 - 2/\log n)$, half μ will get the desired number of inserts. After that, by Chernoff bounds, assuming $\mu/2 \geq \log n$, we get that at least half of $(\prod_{i=0}^{r-1} \frac{q_{r-1}}{td})^d$ fraction will become r-steps, except for an exponentially low probability. Chernoff bound is applicable as long as the expected fraction of r-steps, $q_r \frac{n}{td}$ is at least $\log n$. So both the high probability bounds apply as long as $q_r \frac{n}{td} \geq \log n$, giving the slightly weaker recurrence $q_r \geq \frac{1}{4} \cdot (\frac{1}{10t^3d^2})^{td} \cdot q_{r-1}^{td+1}$.

3.5.2 Construction of the witness graph for balls and bins with moves

Whenever the load of a node X becomes i, there must be a unique edge whose insertion causes this to happen. Say U_1U_2 was this edge; that is, U_1 and U_2 are the bins to which the item got hashed. Look at the directed graph when this edge was being added. During the insertion, a backward search to a depth of h was performed from both U_1 and U_2. Say the node X was obtained by traversing back from U_1 to depth of at most h. We will say that the edge U_1U_2 is the *ith contributing-edge* of X, U_2 is the *ith contributing-peer* of X, and the directed path from X to U_1 along which moves were made, is the *ith contributing-path* for X. Since X is a node with minimum load among the ones visited, it must be the case that all the nodes at depth at most h from U_2 must have load at least $i - 1$. Note that the contributing edge U_1U_2 must be newer than and therefore distinct from all the edges traversed in backward search from U_2. Also for each node and each value of i the *ith*-contributing edge has to be unique.

The witness graph is obtained by recursively chasing contributing edges for nodes visited in the backward search from the contributing peer U_2. First we make a simplifying assumption that during the construction of the witness graph, we never run into cycles, always leaving the graph as a tree. We will argue that the number of edges that produce cycles is few enough that they can be ignored. Our goal is to obtain a large witness tree with high degree nodes and argue that such a subgraph is unlikely to exist in G. The problem is that even with our assumption of not encountering cycles, it is still possible to visit earlier nodes through contributing paths, as contributing paths could completely consist of edges in the visited subtree, not leading to any new nodes. We overcome this issue by computing the witness tree by the following recursive procedure.

For a given i-contributing edge U_1U_2:

- Say, U_2 is the *ith* contributing-peer corresponding to this edge; that is, the load for this insert was taken by some node under U_1.
- Look at the subtree, T, (must be a tree by assumption of not encountering cycles) obtained by performing a backward search to depth h from node U_2 when the insertion took place. Look at the set of leaves, L, of this subtree. At that time each node in T has load at least $i - 1$. Since each internal node in L has at least $i - 1$ children the number of edges in L is at most $2|L|$.
- Look at the set S of all edges that are j-contributing edges for some node in L, for either $j = i - 1$ or $i - 2$ or $i - 3$. Essentially an edge $e \in S$ if and only if there is a node $V \in L$ and a $j \in \{i - 1, i - 2, i - 3\}$ such that e is the j-contributing edge for V. Since the subtree T has at most $2|L|$ edges and the set S has $3|L|$ edges, there must be a set Q of at least $|L|$ edges in S that are outside the subtree T. As all the contributing paths leading to these edges are older than the edge U_1U_2 that connects the subtree to the rest of the witness tree, and since by assumption no cycles are encountered, these edges in Q must be outside the entire witness tree constructed so far. To avoid cycles, the corresponding contributing paths must branch off T before reaching the contributing edge in Q.
- Repeat recursively for each j-contributing-edge in Q, where $j \geq i - 3$

Starting from a node with load $4l + h'$, we chop the recursion depth down to l. Also, during the backward bfs, for each node, we pick only l children even if more may be present. Essentially, the witness tree looks like a tree of sub-trees linked by contributing paths. Each subtree has l^h "children" subtrees and no node or edge is repeated. The height of this tree in terms of the number of subtrees is l; so the total number of nodes is l^{lh}

Chapter 4

An Improved Construction for Counting Bloom Filters

4.1 Introduction

So far we have seen how variants of balls and bins processes can be used to construct efficient data structures for exact search. In this chapter we will see how similar methods can be used for a compact data structure to support set operations. A Bloom filter is an inexact representation of a set that allows for false positives when queried; that is, it can sometimes say that an element is in the set when it is not. In return, a Bloom filter offers very compact storage: less than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set. There has recently been a surge in the popularity of Bloom filters and variants, especially in networking [16]. One variant, a counting Bloom filter [52], allows the set to change dynamically via insertions and deletions of elements. Counting Bloom filters have been explicitly used in several papers, including for example [111, 85, 40, 52, 106, 23].

In this chapter, we present a new construction with the same functionality as the counting Bloom filter, based on d-left hashing. We call the resulting structure a d-left counting Bloom filter, or dlCBF. For the same fraction of false positives, the dlCBF generally offers a factor of two or more savings in space over the standard solution, depending on the parameters. Moreover, the construction is very simple and practical, much like the original

Bloom filter construction. As counting Bloom filters are often used in settings where space and computation are limited, including for example routers, we expect that this construction will prove quite useful in practice.

4.2 Background

4.2.1 Bloom Filters and Counting Bloom Filters

We briefly review Bloom filters; for further details, see [16]. A Bloom filter represents a set S of m elements from a universe U using an array of n bits, denoted by $B[1], \ldots, B[n]$, initially all set to 0. The filter uses a group H of k independent hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, n\}$ that independently map each element in the universe to a random number uniformly over the range. (This optimistic assumption is standard and convenient for Bloom filter analyses.) For each element $x \in S$, the bits $B[h_i(x)]$ are set to 1 for $1 \leq i \leq k$. (A bit can be set to 1 multiple times.) To answer a query of the form "Is $y \in S$?", we check whether all $h_i(y)$ are set to 1. If not, y is not a member of S, by the construction. If all $h_i(y)$ are set to 1, it is assumed that y is in S, and hence a Bloom filter may yield a *false positive*.

The probability of a false positive for an element not in the set is easily derived. If p is the fraction of ones in the filter, it is simply p^k. A standard combinatorial argument gives that p is concentrated around its expectation

$$\left(1 - (1 - 1/n)^{mk}\right) \approx \left(1 - \varepsilon^{-km/n}\right).$$

These expressions are minimized when $k = \ln 2 \cdot (n/m)$, giving a false positive probability f of $f \approx (1/2)^k \approx (0.6185)^{n/m}$. In practice, k must be an integer, and both n/m (the number of bits per set element) and k should be thought of as constants. For example, when $n/m = 10$ and $k = 7$ the false positive probability is just over 0.008.

Deleting elements from a Bloom filter cannot be done simply by changing ones back to zeros, as a single bit may correspond to multiple elements. To allow for deletions, a *counting Bloom filter* (CBF) uses an array of n counters instead of bits; the counters track the number of elements currently hashed to that location [52]. Deletions can now be safely done by decrementing the relevant counters. A standard Bloom filter can be derived from a counting Bloom filter by setting all non-zero counts to 1. Counters must be chosen large enough to avoid overflow; for most applications, four bits suffice [20, 52]. We generally use the rule of four bits per counter when comparing results of our data structure with a standard CBF, although we do note that this could be reduced somewhat with some additional complexity.

4.2.2 Related Work on Counting Bloom Filters

The obvious disadvantage of counting Bloom filters is that they appear quite wasteful of space. Using counters of four bits blows up the required space by a factor of four over a standard Bloom filter, even though most entries are zero. Some work has been done to improve on this. The spectral Bloom filter was desigend for multi-sets, but also considers schemes to improve the efficiency of storing counters [111]. A paper on "optimal" Bloom filter replacements is another work in this vein [98], introducing a data structure with the same functionality as a counting Bloom filter that is, at least asymptotically, more space-efficient.

While this problem has received some attention, the previous work does not appear to give useful solutions. Spectral Bloom filters are primarily designed for multi-sets and skewed streams; we do not know of experiments or other evidence suggesting they are appropriate as a replacement for a CBF. The alternatives suggested in [98] do not appear to have been subject to experimental evaluation. Moreover, the schemes suggested in both of these papers appear substantially more complex than the standard counting Bloom filter scheme. This simplicity is not just useful in terms of ease of programming; for implementations in hardware, the simplicity of the Bloom filter scheme translates into very straightforward and clean hardware designs.

Our goal is to provide a scheme that maintains the simplicity of the original counting Bloom filter construction, and further is backed by experimental results demonstrating that the scheme is likely to be very useful in practice. We believe our work is novel in these regards. We point out that our motivation for our general approach came about when considering generalizations of Bloom filters for state machines. See [14] for more details.

4.2.3 Background: d-Left Hashing

Our approach makes use of d-left hashing, a variation of the balanced allocations paradigm [2] due to Vöcking [112], which we now recall. Normally this setting is described in terms of balls and bins; here, for consistency, we use the terms elements and buckets. We have a hash table consisting of n buckets. Initially they are divided into d disjoint subtables of n/d buckets. (For convenience we assume n/d is an integer.) We think of the subtables as running consecutively from left to right. Each incoming element is hashed to give it a collection of d possible buckets where it can be placed, one in each subtable. We generally assume in the analysis that these choices are uniform and independent. Each incoming element is placed in the bucket containing the smallest number of elements; in case of a tie, the element is placed in the bucket of the leftmost subtable with the smallest number of elements. In order to search for an element in the hash table, the contents of d buckets must be checked. Note that, if the bucket size is fixed a priori, there is the possibility of overflow. Various combinatorial bounds on the resulting maximum load have been proven [12, 112].

For our purposes, we are more interested in obtaining precise estimates of d-left hashing under constant average load per bucket. For the case where elements are only inserted, this can be obtained by considering the fluid limit, corresponding to the limiting case where the number of elements and buckets grow to infinity but with the ratio between them remaining fixed. The fluid limit is easily represented by a family of differential equations, as described in [20, 94]. The advantage of using the fluid limits in conjunction with simulation is that it provides insight into how d-left hashing scales and the probability of overflow when fixed bucket sizes are used.

Analyzing the behavior with deletions is somewhat more problematic in this framework, as one requires a suitable model of deletions. Good insight can be gained by the following approach. Suppose that we begin by inserting m elements, and then repeatedly, at each time step, delete an element chosen uniformly at random and then insert a new element. The corresponding fluid limit equations can be easily derived and are very similar to the insertion-only case. We can run the family of equations until the system appears to reach a steady state distribution.

4.3 The d-left CBF construction

4.3.1 The Framework

Our goal is to design a structure that allows membership queries on a set S over a universe U that can change dynamically via insertions and deletions, although there will be an upper bound of m on the size of the set. A query on $x \in S$ should always return that $x \in S$; a query on some $y \notin S$ could give a false positive. The target false positive rate is ϵ. We allow for data structures that may with very small probability reach a failure condition, such as the overflow of a counter, at some point in its lifetime. Preferably, the failure probability is so small that it should not occur in practice. The failure condition should, however, be apparent, so that an appropriate response can be taken if necessary.

A standard counting Bloom filter offers one possible solution to this problem. Our alternative has a different starting point. It is a folklore result (see [16]) that if the set S is static, one can achieve essentially optimal performance by using a perfect hash function and fingerprints. One finds a perfect hash function $P : U \to [|S|]$, and then stores at each location an $f = \lceil \log 1/\epsilon \rceil$ bit fingerprint in an array of size $|S|$, computed according to some (pseudo-)random hash function H. A query on z requires computing $P(z)$ and $H(z)$, and checking whether the fingerprint stored at $P(z)$ matches $H(z)$. When $z \in S$ a correct response is given, and when $z \notin S$ a false positive occurs with probability at most ϵ; this uses $m\lceil \log 1/\epsilon \rceil$ bits.

The problem with this approach is that it does not cope with changes in the set S, and perfect hash functions are generally too expensive to compute for most applications. To deal with this, we make use of the fact, recognized in [20], that using d-left hashing provides a natural way to obtain an "almost perfect" hash function. The resulting hash function is only almost perfect in that instead of having one set element in each bucket, there can be several, and space is not perfectly utilized. A strong advantage, however, is that it can easily handle dynamically changing sets. The resulting construction meets our goals of being a substantial improvement over Bloom filters while maintaining simplicity.

4.3.2 The Construction of a d-left Counting Bloom Filter

We first present a seemingly natural construction of a dlCBF that has a subtle flaw; we then demonstrate how this flaw can be corrected. To begin, we use a d-left hash table, where each bucket consists of many cells, each cell being a fixed number of bits meant to hold a fingerprint and a counter. As we want to avoid pointers to keep our representation as small as possible, we use a fixed number of cells per bucket, so that our hash table may be viewed as a large bit array.

We store a fingerprint for each element. The fingerprints are essentially compressed by taking advantage of how they are stored. Specifically, each fingerprint will consist of two parts. The first part corresponds to the *bucket index* the element is placed in. We assume the bucket index has range $[B]$, where in this setting we use $[x] = \{0, 1, \ldots, x - 1\}$. The second part is the remaining fingerprint, which we refer to as the *remainder*, and is stored explicitly. We assume the remainder has range $[R]$.

For example, if we were just using a single hash function, and a single hash table with B buckets, we would use a hash function $H : U \to [B] \times [R]$. The m elements of S would be stored by computing $H(x)$ for each $x \in S$ and storing the appropriate remainders in a cell for each bucket. In order to handle deletions in the case that two (or more) elements might yield the same bucket and remainder, each cell would also contain a small counter. A false positive would occur if and only if for a query $y \notin S$ there existed $x \in S$ with $H(x) = H(y)$.

Using a single hash function yields the problem that the distribution of the load varies dramatically across buckets, essentially according to a Poisson distribution. Since we use a fixed number of cells per bucket, to avoid overflow requires a small average load as compared to the maximum load, leading to a lot of wasted spaced. Using d-left hashing dramatically reduces this waste.

We now explain the subtle problem with using d-left hashing directly. Let us suppose that our hash table is split into d subtables, each with B buckets. To use d-left hashing, we would naturally use a hash function $H : U \to [B]^d \times [R]$, giving d choices for each element, and store the remainder in the least loaded of the d choices (breaking ties to the left). The problem arises when it comes time to delete an element from the set. The corresponding remainder might be found in more than one of the d choices, as the same remainder might have been placed by another element in another of these d buckets at some later point in time. When this happens, we do not know which copy to delete.

It is worth making this clear by framing a specific example. Suppose that when an element x is inserted into the table, its d choices correspond to the first bucket in each subarray, and its remainder is a. Suppose further that the loads are such that the remainder is stored in the last subarray. Now suppose later than an element y is inserted into the table, its d choices correspond to the ith bucket in the ith subarray for each i, and its remainder is also a. Notice that, because the remainder a was placed in the first bucket of the *last* subarray for x, when y is placed, this remainder a will not be seen in any of y's buckets. Now suppose that, due to y, the remainder a is placed in the first bucket of the first subarray. Finally, consider what happens when we now try to delete x. The appropriate remainder a now appears in two of x's buckets, the first and the last, and there is no way to tell which to delete. Deleting both would lead to false negatives for queries on the element y; such occurrences happen too frequently to allow this approach. Failing to delete would leave garbage in the table, causing it to fill and leading to increased false positives.

We solve this problem by breaking the hashing operations into two phases. For the first phase, we start with a hash function $H : U \to [B] \times [R]$; this gives us the true fingerprint $f_x = H(x)$ for an element. For the second phase, to obtain the d locations, we make use of additional (pseudo)-random permutations $P_1, \ldots, P_d$. Specifically, let $H(x) = f_x = (b, r)$. Then let

$$P_1(f_x) = (b_1, r_1), P_2(f_x) = (b_2, r_2), \ldots, P_d(f_x) = (b_d, r_d).$$

The values $P_i(f_x)$ correspond to the bucket and remainder corresponding to f_x for the ith subarray. Notice that for a given element, the remainder that can be stored in each subarray can be different; although this is not strictly necessary, it proves convenient for implementation. When storing an element, we first see whether in any bucket b_i the remainder r_i is already being stored. If so, we simply increment the corresponding counter. We point out that these counters can be much smaller than counters used in the standard CBF construction, as here collisions are much rarer; they occur only when H gives the same result for multiple elements. Also, as we show in Claim 4.3.1, only one remainder associated with f_x is stored in the table at any time, avoiding any problem with deletions. If r_i is not already stored, we store the remainder in the least loaded bucket according to the d-left scheme.

The following simple claims demonstrate the functionality of this dlCBF construction. When bounding the probabilty of a false positive below, we ignore the negligible probabilities of counter or bucket overflow, which must be considered separately.

Claim 4.3.1. *When deleting an element in the set, only one remainder corresponding to the element will exist in the table.*

Proof. Suppose not. Then there is some element $x \in S$ whose remainder is stored in subtable j to be deleted and at the same time another element $y \in S$ such that $P_i(f_x) = P_i(f_y)$ for $i \neq j$. Since the P_i are permutations, we must have that $f_x = f_y$, so x and y share the same true fingerprint. Now suppose without loss of generality that x was inserted before y; in this case, when y is inserted, the counter in subtable j associated with the remainder of x would be incremented, contradicting our assumption. □

Claim 4.3.2. *A false positive for a query z occurs if and only if $H(z) = H(x)$ for some $x \in S$.*

Proof. If z gives a false positive, we have $P_i(f_x) = P_i(f_z)$ for some $x \in S$. But then $H(x) = H(z)$. □

Claim 4.3.3. *The false positive probability is* $1 - (1 - 1/BR)^{|S|} \approx m/BR$.

Proof. The probability that there is no false positive for z is the probability that no $x \in S$ has $H(x) = H(z)$, and this expression corresponds to that probability. □

We have thereby avoided the problem of finding two possible fingerprints to delete when handling deletions. In return, however, we have introduced another issue. Our process is no longer exactly equivalent to the d-left hashing process we have analyzed, since our d choices are no longer independent and uniform, but instead determined by the choice of the permutations. In any instantiation, there are really only BR collections of d choices available, not a full B^d. Fortunately this problem is much less significant, at least in practice. Intuitively, this is because the dependence is small enough that the behavior is essentially the same. We verify this with simulations below.

4.3.3 Additional Practical Issues

In practice we recommend using simple linear functions for the permutations; for example, when $H(x)$ can has range $[2^q]$, we suggest using

$$P_i(H(x)) = aH(x) \bmod 2^q$$

for a chosen uniformly at random from the odd numbers in $[2^q]$. The high order bits of $P_i(H(x))$ can be used for the bucket, and the low order bits for the fingerprint. (Because of the dependence of these hash functions, using the low order bits for the buckets is less effective; the same groups of buckets will frequently be chosen. Also, although even $H(x)$ values are then placed only in even buckets, and similarly for odd $H(x)$ values, since $H(x)$ values are (pseudo)-random the effect is negligible.) In this case, it is harder to see why the system behavior should necessarily follow the fluid limit, since the dependence among the bucket choices is quite strong with such limited hash functions. However, our simulations, discussed below, suggest the fluid limit is still remarkably accurate. (Some theoretical backing for this comes from the the recent results of [83])

Using simple invertible permutations P_i may allow further advantages. For example, when inserting an element, it may be possible to move other elements in the hash table, as long as each element is properly placed according to one of its d choices. (Allowing such movement of elements was the insight behind cuckoo hashing [103], and subsequent work based on cuckoo hashing, including [99].) Intuitively, such moves allow one to rectify previous placement decisions that may have subsequently turned out poorly. Recent work has shown that even limited ability to move existing elements in the hash table can yield better balance and further reduced loads [103, 99]. In particular, such moves may be useful as an emergency measure for coping with situations where the table becomes overloaded with more elements than the initial design suggested; specifically, the moves could prevent overflow. By using simple invertible permutations P_i, one can compute f_x from a value $P_i(f_x)$, and move it to another location given by $P_j(f_x)$. We have not studied this approach extensively, as we believe the costs outweigh the benefits for our target applications, but it may be useful in future work.

4.4 A Comparison with Standard Counting Bloom Filters

Roughly speaking, our experience has been that for natural parameters, our dlCBF uses half the space or less than standard CBF with the same false positive probability, and it appears as simple or even simpler to put into practice. We now formalize this comparison. Suppose, for convenience, that we are dynamically tracking a set of m elements that changes over time.

For m elements, a standard CBF using cm counters, each with four bits, as well as the theoretically optimal $k = c \ln 2$ hash functions, gives a false positive probability of approximately $(2^{-\ln 2})^c$ using $4cm$ bits. (This is slightly optimistic, because of the rounding for k.) The probability of counter overflow is negligible.

A comparable system using our dlCBF would use four subarrays, each with $m/24$ buckets, giving an average load of six elements per bucket. The method of Section 4.2.3 shows that providing room for eight elements per bucket should suffice to prevent bucket overflow with very high probability. Each cell counter should allow for up to four elements with the same hash value from the hash function H in the first step to prevent counter overflow with high probability. This can be done effectively with 2 bit counters, as long as one has a sentinel cell value, say 0, that cannot be a fingerprint but represents an empty cell. (We ignore the minimal effect of the sentinel value henceforth.) With an r bit remainder, the false positive probability is upper bounded by $24 \cdot 2^{-r}$, and the total number of bits used is $4m(r + 2)/3$. (For convenience, we think of $r \geq 5$ henceforth, so that our upper bound on the probability is less than 1.) Alternatively, one can think in the following terms: to obtain a false positive rate of $f = 24 \cdot 2^{-r}$, one needs to use $(4 \log_2(1/f) + 20 + 4 \log_2 3)/3$ bits per element. We note that the constant factor of $4/3$ in the leading term $4 \log_2(1/f)/3$ could be reduced arbitrarily close to 1 by using buckets with more items and filling them more tightly; the corresponding disadvantages would be a larger constant term, and more cells would need to be examined on each lookup when trying to find a matching remainder.

Equating $c = (r + 2)/3$, the two approaches use the same amount of space. But comparing the resulting false positive probabilities, we find

$$(2^{-\ln 2})^{(r+2)/3} > 24 \cdot 2^{-r}$$

for all integers $r \geq 7$. Indeed, the more space used, the larger the ratio between the false positive probability of the standard CBF and the dlCBF. For $r = 14$ and $c = 16/3$, for example, the two structures are the same size, but the false positive probability is over a factor of 100 smaller for the dlCBF. Moreover, the dlCBF actually uses less hashing than a standard CBF once the false positive probability is sufficiently small.

Alternatively, we might equalize the false positive probabilities. For example, using 9 4-bit counters (or 36 bits) per element with six hash functions in a standard CBF gives a false positive probability of about 0.01327. Using 11-bit remainders (or 52/3 bits per element) with the dlCBF gives a smaller false positive probability of approximately 0.01172. The dlCBF provides better performance with less than $1/2$ the space of the standard CBF for this very natural parameter setting.

4.5 Simulation Results

4.5.1 A Full Example and Comparison

We have implemented a simulation of the dlCBF in order to test its performance and compare to a standard CBF. We focus here on a specific example, and extrapolate from it. We chose a table with 4 subarrays, each with 2048 buckets, and each bucket with 8 cells, for a total capacity of 2^{16} elements. Our target load is $3 \cdot 2^{14} = 49152$ elements, corresponding to an average load of six items per bucket. The approach of Section 4.2.3 suggests that bucket overload is sufficiently rare (on the order of 10^{-27} per set of elements) that it can be ignored.

We must also choose the size of the remainder and the number of counter bits per cell. In our example we have chosen 14 bit fingerprints, which as per our analysis of Section 4.4 should give us a false positive rate of slightly less than $24 \cdot 2^{-14} \approx 0.001465$. We use 2 bit counters per cell to handle cases where a hash value is repeated. The total size of our structure is therefore exactly 2^{20} bits.

In our construction, we use a "strong" hash function for the first phase (based on drand48), and random linear permutations exactly as described in Section 4.3.3 for the second phase.

For every experiment we perform, we do 10000 *trials*. In each trial, we initially begin with a set of 49152 elements, which we represent with the dlCBF; this corresponds to an average of six elements per bucket. We then simulate 2^{20} time steps, where in each time step we first delete an element from the current set uniformly at random, and then we insert a new element chosen uniformly at random from the much larger universe. This test is meant to exemplify the case where the dlCBF always remains near capacity, although the underlying set is constantly changing; it also matches the setting of our fluid limit equations. We test to make sure counter and bucket overload do not occur. After the 2^{20} time steps, we consider 10000 elements not in the final set, and see how many give false positives, in order to approximate the false positive rate that would be observed in practice.

We first consider the issue of overflow in the hash table. Over the 10000 trials, overflow never occurred. In fact, the fourth subarray *never*, over all insertions and deletions, had any buckets with eight elements, so overflow was never even an immediate danger. More concretely, we note that the fluid limit provides a very accurate representation of what we see in the simulation (even though we are using simple random linear permutations). Specifically, after all of the random insertions and deletions, we examine the bucket loads, and consider their distribution. As we can see in Table 4.5.1, the fluid limit matches the simulations extremely well, providing a very accurate picture of performance.

We now turn consider the cell counter. Recall that this counter is necessary to track when multiple elements have the same first round hash value. Over all 10000 trials, the largest this counter needed to be was 4. That is, on six of the trials, there were at some time four extant elements that shared the same hash value. This requires only two bits per cell counter (again assuming a sentinel value). While more precise calculations can be made, it is easy to bound the probability of counter flow: for any set of m elements, the probability that any five will share the same hash value is at most $\binom{m}{5}\left(\frac{1}{|B||R|}\right)^5$, which for our parameters is approximately $5.62\mathrm{e}-17$. Again, for most practical settings, counter overflow can be ignored, or if necessary a failsafe could be implemented.

Finally, we consider false positives. Over the 10000 trials, the fraction of false positives ranged from 0.00106 to 0.00195, with an average of slightly less than 0.001463. This matches our predicted performance almost exactly.

We emphasize again that this is a specific example, and the various performance metrics

	Simulation Results	Steady State (Fluid limit)
Load ≥ 1	1.0000	1.0000
Load ≥ 2	0.9999	0.9999
Load ≥ 3	0.9990	0.9990
Load ≥ 4	0.9920	0.9920
Load ≥ 5	0.9502	0.9505
Load ≥ 6	0.7655	0.7669
Load ≥ 7	0.2868	0.2894
Load ≥ 8	0.0022	0.0023
Load ≥ 9	0.0000	1.681e-27

Table 4.1: Simulation results with $6n$ elements being placed into n buckets using four choices, compared to the differential equations. The simulation results give the fraction of buckets with load at least k for each k up to 9; the results are based on the final distribution of elements after 2^{20} deletions and insertions, averaged over 10000 trials. No bucket obtained a load of 9 at any time over all 10000 trials.

could be improved in various ways. False positives could be reduced by simply increasing the fingerprint size; space utilization could be improved by using fewer, larger buckets.

We now compare with a simulation of a standard CBF. We choose to compare by trying to achieve nearly the same false positive rate We use 13.5 counters per element (with 9 hash functions), or 663552 counters for 49152 elements. At four bits per counter, this works out to 2654208 bits, over 2.5 times the size of our dlCBF. Again, we performed 10000 trials, each having 2^{20} deletions and insertions after the initial insertion of the base set. The largest counter value obtained was 13, which appears in just one of the 10000 trials; counter values of 12 were obtained in 16 trials. These results match what one would obtain using a back-of-the-envelope calculation based on the Poisson approximation of the underlying balls and bins problem, as in e.g. Chapter 5 of [92] (or see also [52]). The approximate probability that a counter is hashed to by 16 elements (giving an overflow) in an optimal CBF configuration, where the expected number of hashed elements per counter is $\ln 2$, is approximately $e^{-\ln 2}(\ln 2)^{16}/(16!) \approx 6.79 \cdot 10^{-17}$, roughly the same as the counter overflow probability for the dlCBF.

Over the 10000 trials, the fraction of false positives ranged from 0.00108 to 0.00205, with an average of slightly less than 0.001529. Again, this matches our predicted performance almost exactly, and it is just slightly higher then the dlCBF. As in Section 4.4, our conclusion is that the dlCBF can provide the same functionality as the standard CBF, using much less space without a significant difference in complexity.

4.5.2 Additional Simulation Results

Suppose that we add more elements to the initial set, so that the average load per bucket is 6.5 instead of 6. (Specifically, in our simulations, we had 8192 buckets and 53428 elements, with each bucket having a capacity of 8 elements.) Using the fluid limit differential equations, we find that (in the steady state limit) the fraction of buckets with load at least 9 is approximately $2.205e-08$. We would therefore expect in our experiments, with just over 1 million deletions and insertions, that some bucket would reach a load of 9 (causing an overflow if buckets have capacity 8) slightly over 2 percent of the time. This indeed matches our simulations; over 10000 trials, we had an overflow condition in 254 trials. This example demonstrates the fact that in general the equations are quite useful for determining the threshold number of elements over which overflow is likely to occur.

These overflows could be mitigated by allowing elements to be moved, as discussed in section 4.3.3. We have implemented and tested this functionality as well. Specifically, we have focused on a simple improvement: if all the buckets associated with an inserted element are at capacity, we see if any of the items in just the *first* subarray can possibly be moved to another of its choices to resolve the overflow. This additional failsafe allowed us to handle an average load per bucket of 6.75 (or 55296 elements), without overflow in 10000 trials. A potential bucket overflow occured between 40 to 100 times in each trial, but even this limited allowance of movement allowed the potential overflows to be resolved. Greater loads could be handled by allowing more movement, and this is certainly worthy of further experimentation. For many applications, however, including the router-based applications we are considering, we believe that movement should remain a failsafe or a very rare special case. The small loss in space utilization is compensated for by simplicity of design.

4.6 Conclusion

We have demonstrated via both analysis and simulation that the dlCBF provides the same performance as a standard CBF using much less space. We believe the dlCBF will become a valuable tool in routing hardware and other products where the functionality of the counting Bloom filter is required.

One interesting area we hope to examine in future work is how to make the dlCBF responsive to large variations in load. The ability to move fingerprints offers one approach. Another interesting possibility is to dynamically changing the size of the fingerprint stored according to space needs.

A more general question relates to the use of d-left hashing. Since d-left hashing provides a natural way to obtain an "almost perfect" hash function, where else could it be used effectively to improve on a scheme that calls for perfect hashing?

Chapter 5

Entropy based Nearest Neighbor Search in High Dimensions

5.1 Introduction

Having seen algorithms for search with exact match we now turn our attention to similarity search. One common approach is to convert the search object into a mathematical representation such as a point (vector) in a high dimensional euclidean space. For example a complex search object such as an image could be divided into several regions and a number could be used to quantify the average color in a region – such a vector is called as a feature vector. The search for a similar object then becomes a nearest neighbor search in a high dimensional space. In this chapter we study the problem of finding the nearest neighbor of a query point in the high dimensional Euclidean space: given a database of n points in a d dimensional space, find the nearest neighbor of a query point. This fundamental problem arises in several applications including data mining, information retrieval, and image search where distinctive features of the objects are represented as points in $\mathbb{R}^d$ [105, 109, 30, 39, 57, 51, 102, 50].

One of the earliest data structures proposed for this problem that is still in common use is the kd-tree [15] that is essentially a hierarchical decomposition of space a long different dimensions. For low dimensions the kd-tree can be used for nearest neighbor queries in logarithmic time and linear space. However the performance seems to degrade as a number of dimensions becomes larger than two. For high dimensions, the exact problem of nearest neighbor search seems to suffer from the curse of dimensionality; that is, either the running time or the space requirement grows exponentially in d. For instance Clarkson [26] makes use of $O(n^{\lceil d/2 \rceil (1+\delta)})$ space and achieves $O(2^{O(d \log d)} \log n)$ time. Meiser [87] obtains a query time of $O(d^5 \log n)$ but with $O(n^{d+\delta})$ space.

The situation is much better for finding an approximate solution whose distance from the query point is at most $1 + \epsilon$ times its distance from the nearest neighbor. [7, 78, 69, 82]. Arya et. al. [7] use a variant of kd-trees that they call BDD-trees ((Balanced Box-Decomposition trees) that performs $(1 + \epsilon)$-approximate nearest neighbor queries in time $O(d\lceil 1 + 6d/\epsilon \rceil^d \log n)$ and linear space.

For arbitrarily high dimensions, [82] provide an algorithm for finding an $(1+\epsilon)$-approximate nearest neighbor of a query point in time $\tilde{O}(d \log n)$ using a data structure of size $(nd)^{O(1/\epsilon^2)}$. Since the exponent of the space requirement grows as $1/\epsilon^2$, in practice this may be prohibitively expensive for small ϵ. Indeed, since even a space complexity of $(nd)^2$ may be too large,perhaps it makes more sense to interpret these results as efficient, practical algorithms for c-approximate nearest neighbor where c is a constant greater than one. Also, this is meaningful in practice as typically when we are given a query point we are really interested in finding a neighbor that is much closer to the query point than the other points – the query point (say an image) really represents the 'same object' as the nearest neighbor we expect it to 'match' except that they may differ a little due to noise, or inherent errors in how well points represents their objects, but it is expected to be quite far from the other points in the database which basically represent 'different objects' from the query point.

For these parameters, Indyk and Motwani [69] provide an algorithm for finding the c-approximate nearest neighbor in time $\tilde{O}(d + n^{1/c})$ using an index of size $\tilde{O}(n^{1+1/c})$ (while their paper states a query time of $\tilde{O}(dn^{1/c})$, if d is large this can easily be converted to $\tilde{O}(d + n^{1/c})$ by dimension reduction); with a data structure of near linear size, for the hamming space, the algorithms in [69, 82] require a query time of $n^{O(\log c/c)}$. To put this in perspective, finding a 2-approximate nearest neighbor requires time $O(\sqrt{n})$ and an index of size $O(n\sqrt{n})$. The exponent was improved slightly in [42] for c in $[1, 10]$ – instead of $1/c$ it was β/c where β is a constant slightly less than 1 for $c < 10$; for example when $c = 2$ they can reduce the exponent to approximately 0.42 implying a running time of $n^{0.42}$ and an index of size $n^{1.42}$. Their simulation results indicate that while locality sensitive hashing gives faster query time over other data structures based on *kd*-tree, it also comes at the expense of using a lot more space. They work with the following decision version of the c-approximate nearest neighbor problem: given a query point, and a parameter r for the distance to its nearest neighbor, find any neighbor of the query point that is that distance at most cr. It is well known that the reduction to the decision version adds only a logarithmic factor in the time and space complexity [69, 63].

In their formulation, they use a locality sensitive hash function that maps points in the space to a discrete space where nearby points out likely to get hashed to the same value and far off points out likely to get hashed to different values. Precisely, given parameter m that denotes an upper bound on the probability that two points at most r apart hash to the same bucket and g a lower bound on the probability that two points more than cr apart hash to the same bucket, they show that such a hash function can find a c-approximate nearest neighbor in $\tilde{O}(d + n^{\rho})$ time using a data structure of size $\tilde{O}(n^{1+\rho})$ where $\rho = log(1/m)/log(1/g)$.

Their approach is to construct several hash tables to ensure that the query point hashes to the same bucket as its nearest neighbor in at least one table. Our approach is different – we use one (or a few) hash table and hash several randomly chosen points in the neighborhood of the query point showing that at least one of them will hash to the bucket containing its nearest neighbor. We show that the number of randomly chosen points in the neighborhood of the query point q required depends on the entropy of the hash value $h(p)$ of a random point p at distance r from q, given q and the locality sensitive hash function h chosen randomly from the hash family. Precisely, we show that if the entropy $I(h(p)|q, h) = M$ then we can find the approximate nearest neighbor in $\tilde{O}(d + n^{\rho})$ time and near linear space $\tilde{O}(n)$ where $\rho = M/\log(1/g)$. Here $I(h(p)|q, h)$ denotes the entropy of $h(p)$ for a random point p at distance r from q given the query point q and the specific hash function h from the hash family in use. Alternatively we can build a data structure of size $\tilde{O}(n^{1/(1-\rho)})$ to answer queries in $\tilde{O}(d)$ time. By applying this analysis to the locality sensitive hash functions in [69, 82, 42] and adjusting the parameters we show that the c nearest neighbor can be computed in time n^{ρ} and near linear space where $\rho \approx 2.06/c$ as c becomes large. For $c = 2$, ρ turns out to be about $n^{0.69}$. Note that $I(h(p)|q, h)$ can be much lower than $I(h(p)|h(q))$ – the latter corresponds to guessing $h(p)$ from $h(q)$ and can lead to much slower algorithms. For example in the Euclidean case an algorithm based on the latter entropy would give a much higher value of ρ of about $\Theta(\log c/c)$, but using both h and q in conjunction instead of just $h(q)$ gives us the improved results. We also show that if the points are chosen randomly from a spherical gaussian distribution (section 5.5) the value of ρ can be improved to about $1.47/c$

A major advantage of such a small index of size $\tilde{O}(n)$ is that the entire index could possibly fit in main memory making all memory accesses RAM accesses instead of the much slower disk accesses. This suddenly increases the number of possible accesses in the same query time by a factor of 1000's ! If there is a unique c-approximate nearest neighbor – which may be typical in practice as the nearest neighbor may be close enough that the gap between the nearest neighbor and the second nearest neighbor may be larger than a factor of c – we argue that only $2\log n$ bits of storage are required in the index for each point for large enough values of c. So even with a million entries, we need only an index of size 5MB which is a trivial amount of RAM space in today's PCs.

Application of our techniques to the L1 norm does not result in any improvement over the previous results – with linear space we get a value of the value of ρ about $\log(c)/c$ matching the bounds in [82, 69].

5.2 Review

5.2.1 Problem statement

Given a set S of n points in d-dimensions, construct a data structure that given a query finds the nearest (or a c-approximate) neighbor. A c-approximate near neighbor is a point at distance at most c times the distance to the nearest neighbor. Alternatively it can be viewed as finding the exact nearest neighbor when the second nearest neighbor is more than c times the distance to the nearest neighbor.

We also work with the following decision version of the c-approximate nearest neighbor problem: given a query point and a parameter r indicating the distance to its nearest neighbor, find any neighbor of the query point that is that distance at most cr. We will refer to this decision version as the (r, cr)-nearest neighbor problem and a solution to this as a (r, cr)-nearest neighbor. It is well known that the reduction to the decision version adds only a logarithmic factor in the time and space complexity [69, 63]. Unless otherwise stated, we will use the l_2 norm in R^d.

We will now review some of the representative approaches used for nearest neighbor search. While there may be others, we specifally mention a few that represent the broad approaches.

5.2.2 Kd-Trees

Although many different flavors of kd-trees have been devised, their purpose is always to hierarchically decompose space into a relatively small number of cells such that no cell contains too many input objects. Givena query, we traverse down the hierarchy until we find the cell containing a object considered to be the matching object for a query.

Typical algorithms construct kd-trees by partitioning point sets recursively along different dimensions. Each node in the tree is defined by a plane through one of the dimensions that partitions the set of points into left/right (or up/down) sets, each with half the points of the parent node. These children are again partitioned into equal halves, using planes through a different dimension. Partitioning stops after $\log n$ levels, with each point in its own leaf cell. The partitioning loops through the different dimensions for the different levels of the tree, using the median point for the partition. Kd-trees are known to work well in low dimensions but seem to fail as the number of dimensions increase beyond three.

5.2.3 BDD Trees

BDD-trees (Balanced Box-Decomposition trees) [7] are extensions of kd Trees with additional auxiliary data structures with the following guarantees on computing an $(1 + \epsilon)$-approximate nearest neighbor.

Theorem 5.2.1. *The* $(1 + \epsilon)$ *-approximate nearest neighbor among n points in* R^d *with* l_p *norm can be computed using a BDD-tree in time* $O(d\lceil 1 + 6d/\epsilon\rceil^d \log n)$

Just like kd-trees, BDD trees hierarchically decompose space into regions of O(d) complexity defined by axis aligned hyper rectangles with the following differences:

1. The rectangles are fat – that is, the ratio between the longest and shortest sides is bounded.
2. Space is recursively subdivided it into a collection of cells, each of which is either a d-dimensional rectangle or the set theoretic difference of the rectangles, one enclosed within the other.

Search Algorithm: The search algorithm starts just as in a kd-tree by a simple descent down the tree. Upon reaching a leaf node, we enumerate the leaf cells in increasing order of distance from the query point, where the distance to a cell is measured as the distance to the nearest point in the cell from the query point. Let p denote the closest points seen so far. As soon as the distance from the query point to the current cell exceeds $d(p, q)/(1 + \epsilon)$ it follows that the search can be terminated and p can be reported as an approximate nearest neighbor to the query point. The enumeration of the cells in increasing distance from the query point can be performed by using an auxiliary heap.

Proof Intuition: They show that the number of cells visited in the search depends on d and ϵ, but not on n. To see the intuition, considered the last leaf cell to be visited that did not cause the algorithm to terminate. If we let r denote the distance from the query point q to the cell, and p denote the closed data point encountered so far, then because we do not terminate, we know that the distance from q to p is at least $r(1 + \epsilon)$. We could not have seen a leaf cell of diameter less than $r\epsilon$ up to now, since the associated data point would necessarily be closer to q that p. This provides a lower bound on the sizes of these cells seen. The fact that cells are fat and a simple packing argument provide an upper bound on the number of cells encountered.

5.2.4 Locality Sensitive Hashing

Indyk and Motwani [69] use locality sensitive hashing to provide an algorithm for finding the c-approximate nearest neighbor in time $\tilde{O}(d + n^{1/c})$ using an index of size $\tilde{O}(n^{1+1/c})$ (while their paper states a query time of $\tilde{O}(dn^{1/c})$, if d is large this can easily be converted to $\tilde{O}(d + n^{1/c})$ by dimension reduction)

A locality sensitive hash function (figures 5.1 and 5.2) is a random function from a hash family that it maps points in the given space to a discrete space where nearby points are

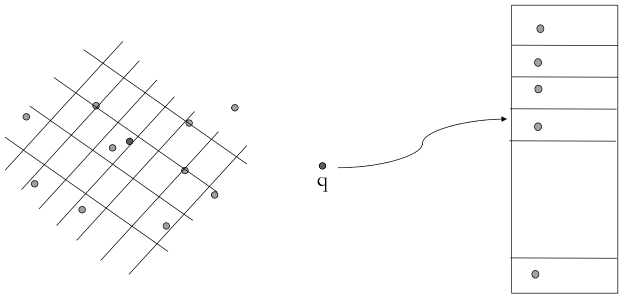

Figure 5.1: Locality sensitive hashing: Nearby points are more likely to hash to same bucket as compared to distant points.

likely to get hashed to the same value and far off points out likely to get hashed to different values.

Definition 2. *A family $\mathcal{H}$ of hash functions from a metric space to a discrete range is called (r_1, r_2, m, g)-sensitive if for any two points p, q that are at most distance r_1 apart, $Pr_{h \in \mathcal{H}}[h(q) = h(p)] \leq m$, and for any two points p, q that are at least distance r_2 apart, $Pr_{h \in \mathcal{H}}[h(q) = h(p)] \geq g$,*

An (r, cr, m, g)-sensitive hash function can be used to compute the c-approximate nearest neighbor in time $\tilde{O}(d + n^{\rho})$ time using a data structure of size $\tilde{O}(n^{1+\rho})$ where $\rho = \log(1/m)/\log(1/g)$. We also need to assume that $\log n > \log(1/g)$.

The (r, cr)-nearest neighbor is obtained as follows: Pick $k = \log n/\log(1/g)$ random hash functions $h_1, h_2, .., h_k$. For each point p in the database compute $H(p) = (h_1(p), h_2(p), .., h_k(p))$ Store these values $H(p)$ for all points in a hash table so as to save space for the values that are absent. Construct n^{ρ} such hash tables independently. To search for the nearest neighbor of a query point q, for each hash table, compute $H(q)$ and examine all points that hashed to the same bucket limiting the search to at most 2 points.

Theorem 5.2.2. *[69] Given a (r, cr, m, g)-sensitive hash family with $\log n > \log(1/g)$, with probability at least $n^{-\rho}$ a query point q at distance at most r from its nearest neighbor p hashes to the same value as q; that is, $H(q) = H(p)$. The expected number of points at distance more than cr from the query point that hash to the same value as q is atmost* 1

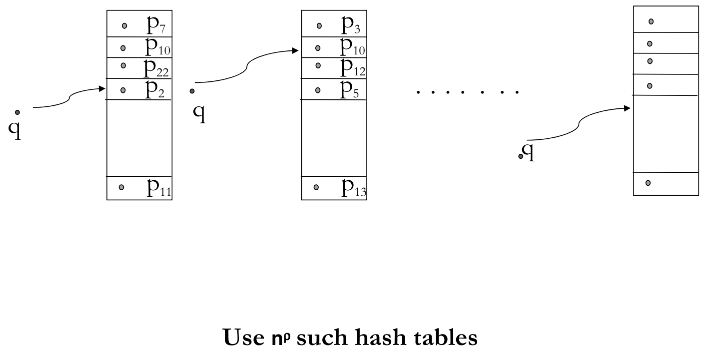

Figure 5.2: Locality sensitive hashing: Many hash tables are used to increase the chance of finding the near neighbor while searching.

Proof. Probability that the query point q hashes to the same value as p is at least $m^k = m^{\log n/\log(1/g)} = n^{-\log(1/m)/\log(1/g)} = n^{-\rho}$.

Probability that one point at distance more than cr from q hashes to $H(q)$ is at most $g^k = 1/n$. So out of n possible such points, number of points that hash to the same value as q is atmost 1 . □

The above theorem implies that with constant probability, the algorithm will succeed in finding the (r, cr)-nearest neighbor over the n^ρ hash tables. By using $O(\log n)$ instances of such a data structure the algorithm can be made to succeed with high probability.

The locality sensitive hash function [69, 42] is obtained by dividing the real line into equal sized intervals of a certain size D and adding a random shift. Precisely, the point p is hashed to an integer $h(p) = \lfloor f(p)/D \rfloor = \lfloor (p.v+\beta)/D \rfloor$ where v is a random variable from the distribution $N^d(0, 1)$ and β is a random number in $[0, D]$. $H(p) = (h_1(p), h_2(p), ..., h_k(p))$. Essentially $H(p)$ is obtained by dividing R^k into a grid of cubes of side length D after a random rotation and shift.

5.2.5 Hamming metric and L1 Norm

Kushilevitz et al [82] prove results very similar to those in [69] but with simpler direct algorithms for the hamming metric that work well even if c is close to 1. Their result for the hamming metric easily extends to the l_1 or l_2 norms by distance preserving embeddings.

Their main theorem for the Hamming metric can be summarized as follows.

Theorem 5.2.3. *There's a data structure of size* $(nlogd)^{O(1/\epsilon^2)}$ *can be used to compute* $1+\epsilon$*-nearest neighbor queries in a d-dimensional hypercube in time* $\tilde{O}(d)$ *with high probability.*

The main idea behind their algorithm for a hypercube is to design a separation test that distinguishes between distances smaller than l and larger than $l(1+\epsilon)$. The test is simple: for a d-dimensional hypercube, the test consists of a subset C of about sd coordinates where each coordinate is chosen with probability $s = \frac{1}{2l}$, and then generating for each vector $v = (v_1, ..., v_d)$ a single bit $P(v) = \sum_{i \in C} r_i v_i (mod 2)$, where each r_i is a random bit.

Let p_1 be the probability that two far-off points (distance greater than $l(1+\epsilon)$) hash to the same bit value) and p_2 denote the same probability for close by points (distance less than l). Then they prove that $p_2 - p_1 \geq O(\epsilon)$. The data structure is obtained by using $t = \tilde{O}(1/\epsilon^2)$ such bits producing a t-bit vector $T(v)$ for every point v in the database. They then construct a table where each point v is recorded against $T(v)$ and all t-bit vectors close to $T(v)$.

5.3 Results

- $B(p, r)$: Let $B(p, r)$ denote the sphere of radius r centered at p a point in $\mathbb{R}^d$; that is the set of points at distance r from p.

- $I(X)$: For a discrete random variable X, let $I(X)$ denote its information-entropy. For example if X takes N possible values with probabilities $w_1, w_2, ..., w_N$ then $I(X) = I(w_1, w_2, .., w_N) = \sum I(w_i) = \sum -w_i \log w_i$

We use locality sensitive hash functions to map database points into a hash table. To find the approximate nearest neighbor of a query point, we hash several randomly chosen points in the vicinity of the query point and show that the approximate nearest neighbor is likely to be present in one of these buckets (figure 5.3).

We assume that the locality sensitive hash function has the following properties. Let M denote the entropy $I(h(p)|q, h)$ where p is a random point in $B(q, r)$. Here $I(h(p)|q, h)$ denotes the entropy of $h(p)$ given the query point and the specific hash function from the

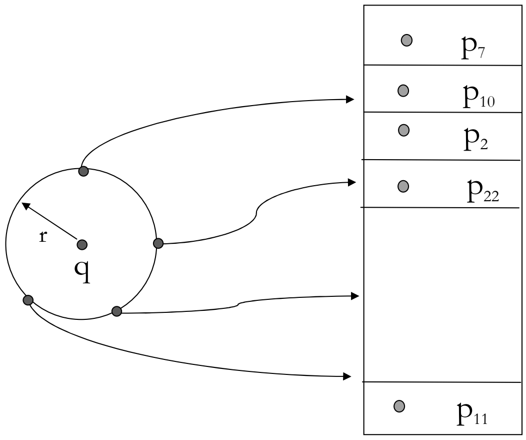

Figure 5.3: New Algorithm: Use only one hash table but search for many points in the neighborhood of the query point

hash family in use. Let g denote an upper bound on the probability that two points that are at least distance cr apart will hash to the same bucket. Note that after a random rotation and a random shift of the origin the nearest neighbor of q appears like a random point on $B(q, r)$. Our algorithm is simple:

Construction of the hash table: Pick $k = \log n/\log(1/g)$ random hash functions $h_1, h_2, .., h_k$. For each point p in the database compute (after random rotations and shifts for each hash function) $H(p) = (h_1(p), h_2(p), .., h_k(p))$ For each point p, store it in a table at location $H(p)$; use hashing to store only the the nonempty locations. Use *polylogn* such randomly constructed hash tables.

Search Algorithm: To find a point at distance at most cr from a query point q given that there is a neighbor at distance at most r from q, pick $\tilde{O}(n^\rho)$ random points v from $B(q, r)$ and search in the buckets $H(v)$. Here $\rho = M/\log(1/g)$.

Theorem 5.3.1. *With probability at least $\tilde{O}(1)$, if the nearest neighbor of the query point is at distance r, the search algorithm finds a neighbor at distance at most cr. With constant probability, no more than $\tilde{O}(n^\rho)$ time is spent searching points that are at a distance more* *than cr from q.*

By using *polylogn* hash tables our algorithms can be made to succeed with high probability.

Alternatively, we show that our methods can be used to construct a data structure of size $\tilde{O}(n^{1/(1-\rho)})$ to answer queries in $\tilde{O}(d)$ time.

By applying this analysis to the locality sensitive hash functions from [82, 69, 42] and adjusting the parameters we show that the c nearest neighbor can be computed in time $\tilde{O}(n^{\rho})$ and near linear space where $\rho \approx 2.09/c$ as c becomes large. For $c = 2$, ρ turns out to be about 0.69.

We start in section 5.4 with preliminaries including a crucial lemma that states the number of random samples required for an arbitrary random variable to guess its specific value. To simplify the exposition of the basic principles, in section 5.5 we study a random instance of the nearest neighbor problem in Euclidean space where the points in the database are chosen randomly from a spherical gaussian distribution. In section 5.6 we prove the main theorems applicable to nearest neighbor search for arbitrary point sets and derive algorithms for nearest neighbor search in Euclidean space. Finally, in section 5.7 we discuss some computational issues relevant for practical implementation.

5.4 Preliminaries

First let us go through some notations.

- $N(\mu, r), \eta(x)$: Let $N(\mu, r)$ denote the normal distribution with mean μ and variance r^2 with probability density function given by $\frac{1}{r\sqrt{2\pi}}e^{-(x-\mu)^2/(2r^2)}$. Let $\eta(x)$ denote the function $\frac{1}{\sqrt{2\pi}}e^{-x^2/2}$.

- $N^d(p, r)$: For the d-dimensional Euclidean space, for a point $p = (p_1, p_2, ..., p_d) \in \mathbb{R}^d$ let $N^d(p, r)$ denote the normal distribution in $\mathbb{R}^d$ around the point p where the ith coordinate of a random point has the normal distribution $N(p_i, r/\sqrt{d})$ with mean p_i and variance r^2/d. It is well known that this distribution is spherically symmetric around p. A point from this distribution is expected to be at root-mean squared distance r from p; in fact, for large d its distance from p is close to r with high probability (see for example lemma 6 in [69])

- $erf(x), \Phi(x)$: The well-known error function $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-x^2}\, dx$, is equal to the probability that a random variable from $N(0, 1/\sqrt{2})$ lies between $-x$ and x. Let $\Phi(x) = \frac{1-erf(x/\sqrt{2})}{2}$. For $x \geq 0$, $\Phi(x)$ is the probability that a random variable from the distribution $N(0, 1)$ is greater than x.

- Use $\alpha \approx 1.303$ to denote the constant: $\int_0^\infty I(\Phi(x), 1 - \Phi(x))\, dx$. The approximate value of this integral has been computed using Matlab.

- Projection: We will use the following commonly used projections that map points in Euclidean space to real numbers. Let v denote a random vector from the distribution $N^d(0, \sqrt{d})$. Then for any point $p \in \mathbb{R}^d$, the projection $f(p) = v.p$ is distributed according to the normal distribution $N(0, ||p||)$ where $||p||$ is the Euclidean distance of p from the origin. Several such projections can be used to project a point p into a low (say k) dimensional space – for example, we can have the function $F(p) = (f_1(p), f_2(p), ..., f_k(p))$ for random choices of projection functions $f_1, ..., f_k$.

The following are well known facts about such random projections (they are direct consequences of the 2-stability of the normal distribution [113]):

Fact 1. *Under a random projection described above, for any point p and q, $F(p) - F(q)$ has the distribution $N^k(0, d(p, q))$ where $d(p, q)$ denotes the distance between p and q. So the distribution of $F(p) - F(q)$ depends only on the distance $d(p, q)$ and not on the positions of p and q.*

Fact 2. *If r is random point on $B(p, r)$, then $F(r) - F(q)$ has the distribution $N(0, \sqrt{(}d(p, q)^2 + x^2)$.*

Guessing the value of a random variable: If a random variable takes one of N discrete values with equal probability then a simple coupon collection based argument shows that if we guess N random values at least one of them should hit the correct value with constant probability. The following lemma states the required number of samples for arbitrary random variables so as to 'hit' a given random value of the variable. It essentially states how many guesses are required to guess the value of a random variable.

Lemma 5.4.1. *Given an random instance x of a discrete random variable with a certain distribution Ω with entropy I, if $O(2^I)$ random samples are chosen from this distribution at least one of them is equal to x with probability at least $\Omega(1/I)$.*

Proof. Let $w_1, w_2, ..., w_N$ denote the probability distribution Ω of the discrete space.

After $s = 4.(2^I + 1)$ samples the probability that x is chosen is $\sum_i w_i[1 - (1 - w_i)^s]$.

If $w_i \geq 1/s$ then the term is at least $w_i(1 - 1/e)$. So if all the $w_i's$ that are at least $1/s$ add up to at least $1/I$ then the above sum is at least $\Omega(1/I)$. Otherwise we have a collection of $w_i's$ each of which is at most $1/s$ and they together add up to more than $1 - 1/I$.

But then by paying attention to these probabilities we see that the entropy $I = \sum_i w_i \log(1/w_i) \geq \sum_i w_i \log s \geq (1-1/I)\log s \geq (1-1/I)(I+2) = I+1-2/I$. For $I \geq 4$, this is strictly greater than I, which is a contradiction. If $I < 4$ then the largest w_i must be at least $1/16$ as otherwise a similar argument shows that $I = \sum_i w_i \log(1/w_i) > w_i \log 16 = 4$, a contradiction; so in this case even one sample guesses x with constant probability. □

Remark 5.4.1. *While the above lemma assumes that the random samples are chosen from the same distribution from which x was derived, it is easy to extend it to the case where random samples are chosen from a distribution slightly different from Ω, where say the probabilities of corresponding events differ at most by a constant factor. For example the random samples could be chosen from a distribution $\Omega' = (w'_1, w'_2, ..., w'_N)$ where the individual probabilities differ from the ones in the distribution $\Omega = (w_1, w_2, ..., w_N)$ by at most a constant multiplicative factor.*

Remark 5.4.2. *The above result is tight to the extent that you cannot get a probability much better than $\Omega(1/I)$ with $O(2^I)$ samples. There is a distribution with entropy I so that even picking $O(I2^I)$ samples will hit x only with probability $O(\log I/I)$. The distribution has one element with probability $4\log I/I$ and all others with equal probability of $O(1/(I^2 2^I))$. The converse of the lemma is not necessarily true. That is there may be a distribution with entropy I, and it may be sufficient to pick much fewer than 2^I samples in fact just one sample – to hit x is significant probability. Think of a distribution where one element has probability $1/2$ there are an exponentially large number of remaining elements with tiny uniform probability.*

5.5 Random Instance in Euclidean Space

We study a random instance of the problem where each point is distributed according to $N^d(0, 1/\sqrt{2})$. The reason we choose this distribution with a deviation of $1/\sqrt{2}$ is because the expected distance between any two pairs of points is 1; in fact, the distance is very close to 1 with high probability for large d. The query point is randomly chosen around a certain point p with distribution $N^d(0, 1/c)$; the query point is at distance close to $1/c$ from its nearest neighbor. The idea is to use the random projections to a real line introduced earlier.

For two points separated by distance x, the distance in the projection is distributed as $N(0, x)$. We use $k = \log n$ such projections. For each point p this gives a vector of real numbers $F(p) = (f_1(p), f_2(p), ..., f_k(p))$. For each projection we produce a bit $h_i(p) = 0$ if $f_i(p) < 0$ and 1 otherwise, giving $H(p) = (h_1(p), h_2(p), ..., h_k(p))$ This hashes each point to an element of $\{0, 1\}^k$. If $k = \log n$, the number of points in any one hash bucket (bin) is at most $\log n$ with high probability.

Unfortunately, the query point q may not hash to the same bucket as its nearest neighbor p. We will try to guess $H(p)$. It can be shown that the hash values $H(p)$ and $H(q)$ are expected to differ in about $O(1/c)$ fraction of the bits. Based on this fact we may need to search a large number of hash buckets, up to $\binom{k}{k/c} \approx n^{I(1/c,1-1/c)} \approx n^{O(\log c/c)}$ for large c.

Our essential observation is that this search space can be pruned significantly by paying attention to the vector $F(q)$ from which $H(q)$ is derived. If a coordinate $f_i(q)$ is far from 0, it is less likely that $h_i(p)$ and $h_i(q)$ will differ. In fact, if the absolute value, $|f_i(q)| = x$ then for $h_i(p)$ and $h_i(q)$ to differ, the projection f_i must map p at least x away from q. This happens with probability at most $e^{-O(x^2c^2)}$. This is exponentially small in c except when x is comparable to $1/c$ which happens only with probability about $1/c$. So the search space of $H(p)$ given $F(q)$ is much smaller than $\binom{k}{k/c}$. To estimate the size of this search space precisely we compute the entropy of $H(p)$ given $F(q)$. If this is M, then by lemma 5.4.1 the search space is about $O(2^M)$.

Now, $I(H(p)|F(q) \leq \sum I(h_i(p)|f_i(q))$. Let us first compute $I(h(q)|f(p))$ for one random projection.

Lemma 5.5.1. *If p is a random point on $B(0, 1/\sqrt{2})$ and q is a random point on $B(p, 1/c)$,* *then for a random projection* $I(h(q)|f(p)) = \frac{1}{c}(1 - o(1))2\alpha/\sqrt{\pi} \approx 1.47/c$

Proof. f(p) has the distribution $N(0, 1)$, and $f(q) - f(p)$ has the distribution $N(0, 1/c)$. So $f(p)$ is at distance x from 0 with probability density $2\eta(\sqrt{2}x)$ and in that case the probability that $f(q)$ is not on the same side as $f(q)$ is $\Phi(xc)$, so the entropy of $h(q)$ is $I(\Phi(cx), 1 - \Phi(cx))$. So

$$\begin{aligned} I(h(q)|f(p)) &= \int_0^\infty 2\eta(\sqrt{2}x) I(\Phi(cx), 1 - \Phi(cx))\, dx \\ &= \frac{2}{\pi}\int_0^\infty e^{-x^2} I(\Phi(cx), 1 - \Phi(cx))\, dx \\ &= \frac{2}{c\pi}\int_0^\infty e^{-(x/c)^2)} I(\Phi(x), 1 - \Phi(x))\, dx \end{aligned}$$

Now $\Phi(x) \leq e^{-x^2/2}/x$ drops exponentially and for large c, $e^{-(x/c)^2}$ drops slowly and is close to 1 until x becomes comparable to c. So $\int_0^\infty e^{-(x/c)^2} I(\Phi(x), 1 - \Phi(x))\, dx = (1 - o(1)) \int_0^\infty I(\Phi(x), 1 - \Phi(x))\, dx$

□

Similarly it can be shown that $I(h(p)|f(q)) = I(h(q)|f(p)) \approx 1.47/c$ (see section 5.8.1). So $I(H(p)|F(q)) \leq 1.47k/c$. But $I(H(p)|F(q))$ is the expected entropy of $H(p)$ given $F(q)$ for random choices of q from $B(p, 1/c)$. We will argue that for large d, even for a fixed random choices of q and f, $I(H(p)|F(q)) \leq (1 + o(1))1.47k/c$ with high probability of $1 - o(1)$: Observe that if $d \geq k$ the tuples $(f_i(q), f_i(p))$ are independent for the k different values of i; so the sum $\sum I(h_i(p)|f_i(q))$ is a sum of independent random variables in the range $[0, 1]$ each with expectation $1.47/c$. By chernoff bounds, with high probability the sum will be close to the mean. Even if $d < k$ the terms are d-wise independent and chernoff bounds may be applied to d-terms at a time; the high probability bound follows if we assume d is large. This means by lemma 5.4.1, with high probability of $1 - o(1)$, the search time is about $2^{(1+o(1))1.47k/c}$ which is $n^{(1+o(1))1.47/c}$.

The algorithm is as follows: For $n^{(1+o(1))1.47/c}$ iterations: Search a random bucket from the distribution of $H(p)$ given $F(q)$. Report the nearest neighbor among all points searched.

Note that $F(p)$ given $F(q)$ has a normal distribution (section 5.8.1) and so sampling with the same distribution as $H(p)$ given $F(q)$ is easy. This gives an algorithm that takes near linear space and $n^{(1+o(1))1.47/c}$ time.

Remark 5.5.1. *In the decision version of the nearest neighbor problem we assumed that we know the exact distance* $1/c$ *to the nearest neighbor whereas in earlier works,* $1/c$ *is only an upper bound on the distance to the nearest neighbor. This can easily be fixed by guessing the exact distance within a factor of* $1+\epsilon$ *where* $\epsilon = O(1/\log n)$*. So* $H(p)$ *has almost the same probability distribution as the nearest neighbor of* q*. Then it follows from remark 5.4.1 that we can still apply lemma 5.4.1 to achieve the same result. The search time only increases by a factor of* $O(\log n)$*.*

Remark 5.5.2. *In our search data structure, we have used a set of* $\log n$ *random hyperplanes to separate the* n *points of the database. It can be shown that if the points can be separated by 'thick' hyperplanes – say* $\log n$ *almost orthogonal hyperplanes of thickness at least* t*, then* $I(h(p)|q,h) = e^{-O(c^2/t^2)}$ *implying a much faster search time of* $n^{e^{-O(c^2/t^2)}}$ *if* t *is not too large. While such thick hyperplanes exist for large dimensions when* $d \geq n$ *(see section 5.8.2), for* $d << n$ *a simple probabilistic calculation shows that such a set of thick separating hyperplanes does not exist.*

Note that for large d, we need not store the entire description of each point in the hash table but only its $O(\log n)$ bit hash value. With high probability, this should be sufficient to distinguish between points that are $1/c$ close to the query point from points that are at least 1 away.

Alternatively, we will show later in section 5.6 how this technique can also be used to search in $\tilde{O}(d)$ time and $n^{(1+o(1))/(1-1.47/c)}$ space.

Although we have assumed that the points are chosen randomly from a normal distribution, our results in this section can be applied to any set of points whose pairwise distances are about the same. This is true when points are chosen randomly from other distributions such as from a cube. In that case we can set the origin to be the centroid of the point set. It can easily be shown that the distance of any point from the centroid is about $1/\sqrt{2}$ of the interpoint distance.

5.6 Generalizing to arbitrary set of points

5.6.1 Proof of Main theorems

We now generalize our techniques to arbitrary set of points. Assume without loss of generality that the nearest neighbor of the query point is at distance $1/c$ from the query point, and we are interested in finding any point at distance at most 1 from the query point. We use locality sensitive hash functions to map database points into a hash table. To find the approximate nearest neighbor of a query point, we hash several randomly chosen points in the $1/c$-neighborhood of the query point and show that a $(1/c, c)$-nearest neighbor is likely to be present in one of these buckets. Let M denote the entropy $I(h(p)|q, h) = M$ where p is a random point in $B(q, r)$. Here $I(h(p)|q, h)$ denotes the entropy of $h(p)$ given the query point and the specific hash function from the hash family in use. Let g denote an upper bound on the probability that two points that are at least distance 1 apart will hash to the same bucket. Pick $k = \log n/\log(1/g)$ random hash functions $h_1, h_2, .., h_k$ (after random rotations and shifts) and store each point p in the database in the bucket compute $H(p) = (h_1(p), h_2(p), .., h_k(p))$. Since many buckets may be empty we use hashing to only store the non-empty buckets.

First observe that after a random rotation and a random shift of the origin, the nearest neighbor of q appears like a random point p on $B(q, 1/c)$ (this rotation and shift may not be required as the hash functions may already perform them implicitly, see section 5.6.2). We will show how to guess $H(p)$ in time $\tilde{O}(n^{M(1+1/\log n)/\log g})$. Since we are only interested in running times where the exponent of n is at most 1, this is $\tilde{O}(n^{M/\log g})$.

Now $I(H(p)|q, H) \leq \sum_1^k I(h_i(p)/q, h_i) = kM$. This means on an average at most kM bits are required to guess $H(p)$ for a given set H of k random hash functions. $I(H(p)|q, H)$ also denotes the expected value of $I(H(p)|q)$ under random choices for fixing the set H of hash functions. So for a fixed H, by Markov inequality with at least probability $1/\log n$, this entropy is at most $kM(1 + 1/\log n)$. Let us assume this is the case. We are now ready to prove theorem 5.3.1.

Proof. **[of theorem 5.3.1]** For a given set H of k hash functions Lemma 5.4.1 implies that by picking $2^{kM(1+1/\log n)}$ random values with the same distribution as $H(p)$, at least one of them is equal to $H(p)$ with at least $O(1/(kM))$ probability. So with one hash table with probability at least $O(1/(kM))$, we can find $H(p)$ in time $2^{kM(1+1/\log n)}$. Also picking random variables with the distribution as $H(p)$ is easy: just compute $H(r)$ where r is a random point from $B(q, 1/c)$. So by lemma 5.4.1, by searching $O(2^{kM(1+1/\log n)})$ buckets obtained by applying H on randomly chosen points v in $B(q, 1/c)$, with probability at least $O(1/(kM))$ we find the nearest neighbor p. Setting $k = \log n/\log(1/g)$ gives us the desired result.

We also need to bound the number of far off points visited over the $O(2^{kM(1+1/\log n)})$ buckets searched. For any point t that is at least distance 1 from q, the probability that it is visited in one bucket is at most g^k. For So out of n such possible points the expected number of such points visited over all buckets is at most $ng^k O(2^{kM(1+1/\log n)}) = O(2^{kM(1+1/\log n)})$. So with probability 1/2 at most twice as many far off points are visited.

□

So in the end the algorithm is simple: Pick $O(2^{kM(1+1/\log n)})$ random points from $B(q, 1/c)$. Search the buckets these points hash to, limiting the total number of points visited at distance more than 1 from q to at most $O(2^{kM(1+1/\log n)})$. Repeat this for *polylogn* hash tables and pick the nearest found neighbor.

Alternatively by storing p in buckets obtained by applying H on $2^{kM(1+\epsilon)}$ randomly chosen points from $B(p, 1/c)$, we can have a small search time with slightly more space. For a fixed random choice of H, by Markov's inequality the probability that $I(H(q)|p)$ exceeds $kM(1+\epsilon)$ is at most ϵ. By lemma 5.4.1 with probability at least $O(\frac{1}{kM(1+\epsilon)})$ the query point will hash to one of these buckets. Again how many far off points can be present in this bucket? A given point t in the database that is at distance at least 1 away from q will be stored in $2^{kM(1+\epsilon)}$ buckets. These buckets are $H(v)$ for $O(2^{kM(1+\epsilon)})$ randomly chosen values v picked from $B(t, 1/c)$. Again if g denotes an upper bound on the probability that one such random point v hashes to the same bucket as q, then the probability that for one such v, $H(v) = H(q)$ is at most g^k.

So over $O(2^{kM(1+\epsilon)})$ choices of v from $B(t, 1/c)$, the probability that any of these hash to the same bucket as q is at most $2^{kM(1+\epsilon)}g^k$. Out of the n points in the database the expected number of points that hash to the same bucket as q is at most $n2^{kM(1+\epsilon)}g^k$. We choose k so that this is at most 1, giving $k = \log n/(\log(1/g) - M(1+\epsilon))$. So by Markov's inequality the probability that more than 2 points distance at least 1 from q hash to the same bucket as q is at most $1/2$. Again by using $O(\log n)$ hash tables with high probability at least for one of them not more than 2 far off points will be searched. We limit the search in each bucket to at most 3 points. Here the size of the hash table is $O(n2^{kM(1+\epsilon)}) = O(n^{\log(1/g)/(\log(1/g)-M(1+\epsilon))}) = O(n^{1/(1-\rho(1+\epsilon))})$. This is $O(n^{1/(1-\rho)})$ if $\epsilon = (1-\rho)^2/\log n$. The total success probability is $O(\frac{\epsilon}{kM(1+\epsilon)}) = \tilde{O}((1-\rho)^3)$

So we have proved the following theorem.

Theorem 5.6.1. *With probability at least $\tilde{O}((1-\rho)^3)$ if we use $k = \log n/(\log(1/g)-M(1+\epsilon))$ projections, using a hash table of size $O(n^{1/(1-\rho)})$ the search algorithm succeeds for one hash table. With constant probability, no more than $\tilde{O}(1)$ points that are at a distance more than 1 from q are searched.*

Again, by using *polylogn* hash tables the algorithm can be made to succeed with high probability.

5.6.2 Choice of Hash functions for Euclidean Space

We now apply our techniques on the locality sensitive hash functions for Euclidean space [82, 69, 42].

Instead of mapping $f(p)$ to a bit we map it to an integer. As in [69, 42], divide the real line into equal sized intervals of size D and add a random shift. Precisely, the point p is hashed to an integer $h(p) = \lfloor (f(p)+\beta)/D \rfloor = \lfloor (p.v+\beta)/D \rfloor$ where v is a random vector from the distribution $N^d(0, \sqrt{d})$ and β is a random number in $[0, D]$. $H(p) = (h_1(p), h_2(p), ..., h_k(p))$. Essentially H divides maps the space R^k into a grid of cubes of side length D.

Let $r_i(p) = (f_i(p)+\beta_i) \mod D$. So $R(p) = (r_1(p), r_2(p), ..., r_k(p))$ denotes the relative position of F(p) within its cube. $R(p)$ is uniformly distributed in $[0, D]^k$. We will later set D to be about 3.

Now consider two points p and q that are distance $1/c$ apart. We will try to guess the relative position of p's subcube $H(p)$ from q's subcube $H(q)$, given the position $R(q)$ of q in its subcube; that is we will try to guess $H(p)-H(q)$. Under the k random projections, $F(p)-F(q)$ is randomly distributed according to $N(0, 1/c)^k$ and is independent of the relative position $R(q)$ in its cube as the alignments of the intervals are independent of the projections f_i. Time required to guess $H(p) - H(q)$ depends on the entropy of $H(p) - H(q)$ given $R(q)$.

The following lemma computes $I(h_i(p) - h_i(q)|r_i(q))$

Lemma 5.6.1. *If p and q are distance $1/c$ apart then under a random projection, $I(h(p) - h(q)|r(q)) = \frac{1}{c}(1 + e^{-O(c^2D^2)})2\alpha/D$ where $\alpha = \int_0^\infty I(\Phi(x), 1 - \Phi(x))\,dx$*

Proof. r(p) is a random value in $[0, D]$; so the probability density that it takes value x in $[0, D]$ is $1/D$. $h(p) - h(q)$ takes integral values, however, as c becomes large, in terms of its entropy most of it is concentrated at 1 and -1. Let M_i denote $I(h(p) - h(q) = i|r(p))$. We are interested in the sum $\sum_i M_i$ over all integers i. By symmetry $M_i = M_{-i}$. If $r(p) = D - x$, $Pr[h(p) - h(q) = 1] = \Phi(cx) - \Phi(cx + cD)$

So

$$\begin{aligned} M_1 &= I(h(p) - h(q) = 1|r(p)) \\ &= \frac{1}{D}\int_0^D I(\Phi(cx) - \Phi(cx + cD))\,dx \\ &= \frac{1}{cD}\int_0^{Dc} I(\Phi(x) - \Phi(x + cD))\,dx \end{aligned}$$

Again as $\Phi(x)$ drops exponentially, $\Phi(x + cD)$ is negligible as compared to $\Phi(x)$, and further the integral to ∞ is not much more as than the integral to Dc. So $\int_0^{Dc} I(\Phi(x) - \Phi(x + cD))\,dx = (1 - e^{-O(c^2D^2)})\int_0^\infty I(\Phi(x))\,dx$

We have shown that M_1 (and M_{-1}) $= (1 - e^{-O(c^2D^2)})\frac{1}{cD}\int_0^\infty I(\Phi(x))\,dx$. Also M_i drops exponentially with i since for a given value of $r(p)$, $Pr[h(p) - h(q) = i]$ drops exponentially with a factor of $e^{-(Dc)^2/2}$.

$$\begin{aligned} M_0 &= \frac{2}{D}\int_0^{D/2} I(1 - \Phi(cx) - \Phi(Dc - xc))\,dx \\ &= \frac{2}{cD}\int_0^{Dc/2} I(1 - \Phi(x) - \Phi(Dc - x))\,dx \end{aligned}$$

Again as before we argue that in the range $[0, Dc/4]$, $\Phi(Dc - x)$ is negligible as compared to $\Phi(x)$, and beyond that they are both negligible ($e^{-O(c^2D^2)}$). This gives us,

$$M_0 = (1 + e^{-O(c^2D^2)}).\frac{2}{cD}\int_0^\infty I(1 - \Phi(x))\,dx$$

So, $\sum_i M_i = (1 + e^{-O(c^2D^2)})(M_{-1} + M_0 + M_1) = (1 + e^{-O(c^2D^2)})\frac{2}{cD}\int_0^\infty I(\Phi(x), 1 - \Phi(x))\,dx$ □

The following lemma computes the probability g that a point t at distance at least 1 from q hashes to the same value $h(t)$ as $h(q)$ under one projection.

Lemma 5.6.2. $g = 1 - \frac{1}{D}\sqrt{\frac{2}{\pi}}(1 - e^{-D^2/2})$

Proof. If $f(t)$ and $f(q)$ are x apart then the probability that they are separated by the interval boundaries is x/D. A simple computation shows that the probability $1 - g$ that t and q hash to different values in one projection is $2\int_0^D (x/D)\eta(x)\,dx = \frac{1}{D}\sqrt{\frac{2}{\pi}}(1 - e^{-D^2/2})$ □

Now since the function r is implicit in the description of the function h, $I(h(p)|q, h) \leq I(h(p) - h(q)/r(q))$. So by theorem 5.3.1 we have:

Corollary 5.6.1. *A c-approximate nearest neighbor in the Euclidean space can be found in time $\tilde{O}(n^\rho)$ using a data structure of size $\tilde{O}(n)$ where $\rho = 2\alpha/[Dlog(1 - \frac{1}{D}\sqrt{\frac{2}{\pi}}(1 - e^{-D^2/2}))]$*

Setting $D = 3$ gives the value of $\rho = 2\alpha/1.26 \approx 2.06$

Alternatively, using a data structure of size $\tilde{O}(n^{1/(1-\rho)})$, we can perform the search operation in $\tilde{O}(d)$ time: again, if $d(t, q) > 1$, the upper bound of g still holds on the probability that a random point in $B(t, 1/c)$ hashes to the same value as q. This is because under the random projections in use, $f(r) - f(q)$ has the same distribution as that of a point at distance $\sqrt{(}d(t, q)^2 + 1/c^2)$ from q – clearly this distance is greater than 1.

Remark 5.6.1. *Although the converse of lemma 5.4.1 is not always true – that is, it is not necessary that $2^{I(X)}$ random samples for a required to guess the value of a random variable – it can be shown that for the specific hash functions in consideration this is the case. That is, we need $2^{(1\pm o(1))kM/\log(1/g)}$ random samples to guess the value of $H(p)$ given $F(q)$. The essential idea is to consider different values of $f(q)$ in small increments of ϵ/c and argue* *that the number of projections for which $R(q)$ lie in a small interval is close to the expected value with high probability and then argue that we need close to the corresponding number of guesses for those set of intervals.*

5.7 Implementation Discussion

We may assume that d is at most $O(\log n)$ as for larger d we can use dimension reduction techniques that preserve distances. Alternately we may use $O(\log n)$ locality sensitive hash functions to represent a point in the database. So we need not store the entire description of each point in the hash table but only its $O(\log n)$ size hash value. This makes the size of each hash entry small especially if we know that there is a unique (r, cr)-approximate nearest neighbor. More succinct representations that use close to at most $2 \log n$ bits can be obtained by first embedding the points into a high-dimensional hamming metric and then reducing the number of dimensions to about $O(\log n)$ by XORing suitable sized random subsets of the bits (see lemma 1 in [82]). If the nearest neighbor is unique then in the final representation, each bit of the query and the nearest neighbor will differ with probability at most $1/(2c)$ whereas for other neighbors each bit position will differ with probability at least $\frac{1}{2}(1 - 1/e)$. A simple and tight probability calculation shows that for large enough c, $2 \log n$ bits suffice with high probability to distinguish the nearest neighbor from the other points. Note that the hash key $H(p)$ need not be stored explicitly as it suffices to hash this into an index for the hash array.

While we have included several *polylogn* factors in the space complexity these are unlikely to be required in practice. The first $\log n$ factor comes from Lemma 5.4.1 and is required only for arbitrary random variables. In our case since the entropy is obtained by adding several different independent random variables it is easy to show that this is not required. The second $\log n$ comes from the application of Markov inequality on the entropy distribution. This again can be eliminated by using say Chebyshev or Chernoff bounds. The third one arises by the crude application of Markov's inequality to ensure that not too many far off points are examined in each hash table. Again we expect this will not be really required in practice. So for large enough constant c, if we are searching for a unique (r, cr) nearest neighbor, the total amount of space required in practice is close to

$2n \log n$ bits. Even for n equal to a million, this is the only 5MB which is a tiny fraction of the main memory space available on PC's.

5.8 Micellaneous technical details

5.8.1 $I(h(p)|f(q))$ for Random Instance

We will show that $I(h(p)|f(q)) = I(h(q)|f(p))$ for the random instance of nearest neighbor search in Euclidean space presented in section 5.5. p is a random point distributed as $N^d(0, 1/\sqrt{2})$, and q is distributed as $N^d(p, 1/c)$. We will compute the probability density that $f(q) = y$, and conditioned on this the probability that $h(p) \neq h(q)$.

After the random projection, $f(q)$ is distributed as $N(0, \sqrt{(}1/2 + 1/c^2))$. Also, the probability density function of $f(p)$ conditioned on $f(q) = y$ is $Pr[f(p) = x]Pr[f(q) = y|f(p) = x]/Pr[f(q) = y] = \eta(x\sqrt{2})\eta((x-y)c)/\eta(y/\sqrt{(}1/2+1/c^2))$, which is $\eta(c^2y/(2+c^2), 1/\sqrt{(}2+c^2))$, the normal distribution with mean $c^2y/(2+c^2)$ and deviation $1/\sqrt{(}2+c^2)$.

So given that $f(q) = y$, probability that $h(p) \neq h(q)$ is $\Phi(\sqrt{(}2 + c^2)c^2y/(2 + c^2)) = \Phi(cy/\sqrt{(}1+2/c^2))$. Since the probability density function of $f(q)$ is also given by $\eta(\sqrt{2}y/\sqrt{(}1+2/c^2))$, this results in the same calculation as for $I(h(p)|f(q))$ except that the variables are scaled by a factor of $\sqrt{(}1 + 2/c^2)$. So, $I(h(q)|f(p)) = I(h(p)|f(q))$.

5.8.2 Thick Hyperplanes

Let us consider the case when d is very large say at least $n \log n$. In that case we choose special hyperplanes that better separate the set of points. The hyperplanes are obtained as follows.

If $v_1, .., v_n$ denote the points of the database, choose a_i randomly to be either $+1$ or -1 and set $h = \sum a_i v_i$. Observe that h is a random variable from $N^d(0, \sqrt{d})$. So if p is random point in $B(q, r)$ then $h.p - h.q$ is distributed as $N(0, r)$ We will show that h separates the set of points well. Indeed, look at $h.v_i = a_i|v_i|^2 + \sum a_j v_i.v_j$. Note that $v_i.v_j$ is very small, distributed as $N(0, 1/\sqrt{d})$. So the sum is distributed as $N(0, \sqrt{(}n/d))$. $|v_i|^2$ is concentrated around 1 and at least $1 - \epsilon$ with high probability (at least $1 - exp(-O(d))$). Also the sum term is at most ϵ with probability at least $1 - exp(O(\log n))$. So with high probability over the $\log n$ projections for all such h, $|h.v_i| > (1-\epsilon)$. Now, since the probability that $h.q \neq h.p$ is clearly at most $exp(-O(c^2))$, we have $I(h(q)/f(p))$ is $O(c^2 exp(-O(c^2)))$.

This argument can also be applied if d is as small as n but deriving the appropriate hyperplanes may require solving a system of equations. If a is the column vector with entries as a_i, then h is obtained by solving $Ah = a$, where the rows of A are the point vectors v_i.

Chapter 6

Nearest Neighbor Search using Kd-Trees

6.1 Introduction

In the last chapter we saw some new results on using locality sensitive hashing for the nearest neighbor search problem in a high dimensional space. In this chapter we will see how those techniques can be applied to nearest neighbor search using kd-trees. Recall that we are given a database of n points in a d dimensional space, and our objective is to find the nearest neighbor of a query point.

One of the earliest data structures proposed for this problem that is still in common use is the kd-tree [15] that is essentially a hierarchical decomposition of space a long different dimensions. For low dimensions this structure can be used for answering nearest neighbor queries in logarithmic time and linear space. However the performance seems to degrade as a number of dimensions becomes larger than two. As explained in the last chapter, for high dimensions, the exact problem of nearest neighbor search seems to suffer from the curse of dimensionality that is either the running time or the space requirement grows exponentially in d.

We also saw that one method to overcome the curse of dimensionality is to find an approximate solution whose distance from the query point is at most $1+\epsilon$ times its distance from the nearest neighbor. [7, 78, 69, 82]. Arya et. al. [7] use a variant of kd-trees that they call BDD-trees ((Balanced Box-Decomposition trees) that performs $(1+\epsilon)$-approximate nearest neighbor queries in time $O(d\lceil 1+6d/\epsilon\rceil^d \log n)$ and linear space. We also looked at locality sensitive hashing [69] to perform approximate nearest neighbor search.

However, to the best of our knowledge the most commonly used method in practice is the kd-tree. We show a simple modification to the search algorithm on a kd-tree that can be used to find the nearest neighbor in high dimensions. The modification consists of simply perturbing the query point before traversing the tree, and repeating this for a few iterations. For a certain database of random points, we show that this modified algorithm can find the c-approximate nearest neighbor in $e^{O(d/c)}$ iterations with reasonable probability, while the traditional kd-tree succeeds only with probability $e^{-\Omega(d/c)}$. We then provide empirical evidence through simulations to show that the simple modification results in high probabilities of success in finding a nearest neighbor in high dimensions.

6.2 Results

6.2.1 Trees vs Hashing for NNS

Here are some differences between tree based and hash based metods for nearest neighbor search.

1. Hashing uses one memory access per table vs trees that take $\log n$ memory accesses.
2. Trees have adaptive granularity depending on the density of the points. A locality sensitive hash function typically breaks up space into equal sized regions. So multiple hash functions need to be used for different granularities.
3. In trees the partitioning planes are chosen in a special way instead of randomly.
4. kd-trees are in common use (as far as I have heard).

6.2.2 New Search Algorithm on Kd trees

Although the kd tree is the most widely used their structure for nearest neighbor search in high dimension, as the number of dimensions increase the efficacy of finding the nearest neighbor drops. Consider for instance a random database of points chosen randomly and uniformly from $[0, 1]^d$. We will plant an query point at distance $1/c$ times the distance to the second nearest neighbor. Let r denote the distance from a random point in the database to the nearest other point in the database. We first show that if the query point is chosen to be a random point at distance about r/c then the probability that a kdtree data structure results in the nearest neighbor is about $e^{-\Omega(d/c)}$.

We then propose a minor modification to the search algorithm so as to boost this probability to a large value. The modification is simple. Instead of searching for the query point in kdtree, perturb it randomly by distance r/c (precisely, from $N^d(p, r/c)$) and search for this perturbed point. Repeat this for $e^{O(d/c)}$ random perturbations and report the nearest neighbor found using the kd tree.

6.2.3 Analysis

Lemma 6.2.1. *Expected distance of any point in database to the nearest other point in the database, r, is* $\Theta(\sqrt{d}/n^{1/d})$

Proof. The volume of a sphere of radius r in d dimensions is $\frac{2\pi^{d/2}r^d}{d\Gamma(d/2)} = \Theta(1)\frac{1}{d^{3/2}}(\frac{2e\pi}{d})^{d/2}r^d$. The radius at which the expected number of points that fall within the ball becomes one this fraction is about $1/n$, giving $r = \Theta(\frac{\sqrt{d}}{n^{1/d}})$ Note that even if the point is near the boundary of the cube then at least $1/2^d$ fraction of the sphere is inside the cube which does not change the value of r by more than a constant factor. The value of r is sharply concentrated around this value as the volume of the sphere changes dramatically with a change in the radius. High concentration bounds can be used to show that there must be at least one point for larger radii and almost no point for smaller radii. □

Distribution of kd-tree partition sizes: Consider a random point p in the database and the kd-tree traversal while searching p. The cell of p is specified by $\log n$ choices of left or right while traversing the kd tree. The number of decisions along each dimension is $\log n/d$. Focusing on the first dimension, let l_i denote length of the cell in the i^{th} branch along the first dimension. $l_0 = 1$. l_i can be thought of as to be obtained from the following process: Pick n random points in $[0, 1]$. Take the median point as a partition point and consider the left interval. Randomly sample $1/2^{d-1}$ fraction of the points. Now take the median again and repeat this process i times. The final interval size corresponds to l_i which is sharply concentrated around $1/2^i$ – more precisely, l_i is distributed as the inverse poisson distribution at rate $2^{(d-1)i}/n$ for $n/2^{di}$ arrivals. The distance x_i been p and the dividing plane at the i^{th} level is a random value between 0 and l_i. At the leaf level x_i is distributed by the exponential distribution with rate $2^{(d-1)\log n/d}/n = 1/n^{1/d}$

Lemma 6.2.2. *If we pick a random query point q at distance r/c from a random point in the database then for large dimensions the probability that a kd search returns the nearest neighbor is about $e^{-\Omega(d/c)}$*

Proof. Let us focus on a leaf cell containing the point p . What is the probability that a random point q chosen from $N(p, r/c)$ lies within the same cell? The cell has $2d$ faces. The distance of each face from the point p is approximately given by the exponential distribution with rate $1/n^{1/d}$. The probability that the query point q , lies on the p side of a given face is about $1 - \Omega(1/c)$. The probability that this happens for all the d directions is about $(1 - \Omega(1/c))^d = e^{-\Omega(d/c)}$. □

Theorem 6.2.1. *With probability at least $\Omega(c/d)$, the new search algorithm finds the nearest neighbor in $e^{O(d/c)}$ iterations for the random database and the random query point.*

The proof of this theorem follows from the following two lemmas.

Recall lemma 5.4.1 from the previous chapter.

Guessing the value of a random variable: If a random variable takes one of N discrete values with equal probability then a simple coupon collection based argument shows that if we guess N random values at least one of them should hit the correct value with constant probability. The following lemma states the required number of samples for arbitrary random variables so as to 'hit' a given random value of the variable. It essentially states how many guesses are required to guess the value of a random variable.

Lemma 6.2.3. *(5.4.1 from previous chapter) Given an random instance x of a discrete random variable with a certain distribution D with entropy I, if $O(2^I)$ random samples are chosen from this distribution at least one of them is equal to x with probability at least $\Omega(1/I)$.*

Let p be a random datebase point and q is a random point with distribution $N^d(p, r/c)$. Let $L(p)$ denote the leaf of the kd-tree where p resides. We will estimate $I[L(p)|q, T]$.

Lemma 6.2.4. $I[L(p)|q, T] = O(d/c)$

Proof. The cell of p is specified by $\log n$ choices of left or right while traversing the kd tree. The number of decisions along each dimension is $\log n/d$. Let b_{ij} ($i \in 1..\log n/d$, $j \in 0..d-1$) denote this choice in the i^{th} branch along dimension j.

Then $I[L(p)|q, T] \leq \sum_{i,j} I[b_{ij}|q, T]$

Focusing on the first dimension for simplicty, for the i^{th} choice in the first dimension, the distance x_i between p and the deciding plane is close to uniform in the range $[0, 1/2^i]$.

The distance between q and p along the dimension is given by $N(0, O(\frac{1}{cn^{1/d}})$.

Since, $I[b_{i0}|x_i = x] = I(\Phi(x), 1-\Phi(x))$, $I[b_{i0}|d_i(p) = x]$ is negligible unless x is $O(1/c)$. This happens with probability $O(\frac{1}{c2^i})$. So $I[b_{i0}|q, T] = O(\frac{1}{c2^i})$.

Similarly bounding and summing over all dimensions, $I[L(p)|q, T] \leq \sum_{i,j} I[b_{ij}|q, T] = O(d/c)$ □

6.2.4 Experiments

We perform experiments with our modified algorithm for different dimensions d, with different number of iterations k for different values of c. The results are summarized in table 6.2.4. We use of the GNU implementation of kd trees. We vary the number of dimensions from 3 to 10 and the approximation constant c from $4/3$ to 4. In 3 dimensions for $c = 4$ kd-trees report the nearest neighbor 84% of the time whereas even with 5 iterations of the new algorithm this goes up to 96%. In 10 dimensions for $c = 4$, 15 iterations raises the success rate from 60% to about 95%.

Table 6.1: Simulation Results: The entries indicate the percentage of times the nearest neighbor os found. As the number of iterations k is increased the success rate increases.

d	c	Kd tree	5 iter	15 iter	20 iter	25 iter	30 iter
3	4	84	96.1	98.8	99.3	99.3	99.8
3	2	73.9	89.5	97.4	98.4	99.0	98.7
3	4/3	73	88.5	96	96.6	98.7	98.7
5	4	73.6	91	97.5	98.1	98.5	99.3
5	2	54	78	92.1	94.9	94.4	96.2
5	4/3	50.7	71.3	87	91.2	92.3	94
10	4	60.7	80.5	94.8	96.6	96.7	96.8
10	2	36	56.4	77.6	84.3	86.6	88.4
10	4/3	25	43.7	61	70	73.4	75.6
20	4/3	13	25	28	41	42	46
20	2	22	42	67	68	70	72

Chapter 7

Lower bounds on Locality Sensitive Hashing

7.1 Introduction

So far we have seen how locality sensitive hashing can be used to find nearest neighbors. In this chapter we study tight lower bounds on the complexity of finding the nearest neighbor of a query point in certain high dimensional spaces using *Locality Sensitive Hashing* (LSH). Recall that the nearest neighbor problem is formulated as follows: Given a database of n points in a metric space, preprocess it so that given a new query point it is possible to quickly find the point closest to it in the data set.

Also recall that while the exact nearest neighbor problem seems to suffer from the "curse of dimensionality", many efficient techniques have been devised for finding an approximate solution whose distance from the query point is at most c times its distance from the nearest neighbor. One of the most versatile and efficient methods for approximate nearest neighbor search is based on Locality Sensitive Hashing, as introduced by Indyk and Motwani in 1998 [71]. This method has been refined and improved in several papers- the most recent algorithm can be found in [42]. Recall that the LSH approach to the approximate nearest neighbor problem is based on the following concept.

Definition 3. *Let (X, d_X) be a metric space, $r, R > 0$ and $p, q \in [0, 1]$. A distribution over* *mappings* $\mathscr{H} : X \to \mathbb{N}$ *is called a* (r, R, p, q)*-sensitive hash family if for any* $x, y \in X$*,*

- $d_X(x, y) \le r \implies \mathbf{Pr}_{\mathscr{H}}[\mathscr{H}(x) = \mathscr{H}(y)] \ge p$.
- $d_X(x, y) > R \implies \mathbf{Pr}_{\mathscr{H}}[\mathscr{H}(x) = \mathscr{H}(y)] \le q$.

Given $c \ge 1$ and $q \in (0, 1)$ we define $\rho_X(c, q)$ to be the smallest constant $\rho > 0$ such that for every $r > 0$ there exists $p \in (0, 1)$ and a (r, cr, p, q)- sensitive hash family $\mathscr{H} : X \to \mathbb{N}$ with $\frac{\log(1/p)}{\log(1/q)} \le \rho$. In other words

$$\rho_X(c, q) = \sup_{r>0} \inf \left\{ \frac{\log(1/p)}{\log(1/q)} : \exists (r, cr, p, q) - \text{sensitive hash family } \mathscr{H} : X \to \mathbb{N} \right\} . \tag{7.1}$$

Of particular interest is the case $X = \ell_s^d$, for some $s > 0$ and $d \in \mathbb{N}$. Here, and in what follows, ℓ_s^d denotes the space $\mathbb{R}^d$ equipped with the ℓ_s norm $\|(x_1, \ldots, x_d)\|_s = (|x_1|^s + \cdots + |x_d|^s)^{1/s}$ (this is only a quasi-norm when $0 < s < 1$). In this case we define

$$\rho_s(c) = \sup_{0<q<1} \limsup_{d\to\infty} \rho_{\ell_s^d}(c, q) .$$

The importance of these parameters stems from the following application to approximate nearest neighbor search. It will be convenient to discuss it in the framework of the following decision version of the c-approximate nearest neighbor problem: Given a query point, find any element of the data set which is at distance at most cr from it, provided that there is a data point at distance at most r from the query point. This decision version is known as the (r, cr)- near neighbor problem. It is well known that the reduction to the decision version adds only a logarithmic factor in the time and space complexity [71, 64]. The following theorem was proved in [71]; the exact formulation presented here is taken from [42].

Theorem 7.1.1. *Let* (X, d_X) *be a metric on a subset of* $\mathbb{R}^d$*. Suppose that* (X, d_X) *admits a* (r, cr, p, q)*-sensitive hash family* $\mathscr{H}$*, and write* $\rho = \frac{\log(1/p)}{\log(1/q)}$*. Then for any* $n \ge \frac{1}{q}$ *there exists a randomized algorithm for* (r, cr)*- near neighbor on* n*-point subsets of* X *which uses* $O\left(dn + n^{1+\rho}\right)$ *space, with query time dominated by* $O(n^\rho)$ *distance computations and* $O\left(n^\rho \log_{1/q} n\right)$ *evaluations of hash functions from* $\mathscr{H}$*.*

Thus, obtaining bounds on $\rho_X(c)$ is of great algorithmic interest. It is proved in [71] that $\rho_1(c) \le 1/c$, and for small values of c, namely $c \in [1, 10]$, is was shown in [42] that this inequality is strict. We refer to [42] for numerical data on the best known estimates for $\rho_1(c)$ for small c. For $s = 2$ a recent result of Andoni and Indyk [5] shows that $\rho_2(c) \le 1/c^2$, and for general $s \in (0, 2]$ the best known bounds [42] are $\rho_s(c) \le \max\{1/c, 1/c^s\}$.

Our main purpose in this chapter is to obtain lower bounds on $\rho_1(c)$ and $\rho_2(c)$ which nearly match the bounds obtained from the constructions in [71, 42, 5]. Our main result is:

Theorem 7.1.2. *For every $c, s \ge 1$,*

$$\rho_s(c) \ge \frac{e^{\frac{1}{c^s}} - 1}{e^{\frac{1}{c^s}} + 1} \ge \frac{e-1}{e+1} \cdot \frac{1}{c^s} \ge \frac{0.462}{c^s} \quad . \tag{7.2}$$

The second to last inequality in (7.2) follows from concavity of the function $t \mapsto \frac{e^t-1}{e^t+1}$ on $[0, \infty)$. Observe also that as $c \to \infty$, $\frac{e^{1/c}-1}{e^{1/c}+1} \sim \frac{1}{2c}$. It would be very interesting to determine $\limsup_{c\to\infty} c \cdot \rho_1(c)$ exactly- due to Theorem 7.1.2 and the results of [71] we currently know that this number is in the interval $[1/2, 1]$.

7.2 Proof of Theorem 7.1.2

The basic idea in the proof of Theorem 7.1.2 is simple. Choose a random point $x \in \{0, 1\}^d$ and consider the random subset A of the cube $\{0, 1\}^d$ consisting of points u for which $\mathscr{H}(u) = \mathscr{H}(x)$. The second condition in Definition 3 forces A to be small in expectation. But, when A is small we can bound from above the probability that after r steps, the random walk starting at a random point in A will end up in A. We obtain this upper bound using a Fourier analytic argument, and in combination with the first condition in Definition 3 we deduce the desired bound on $\rho_1(c)$.

Theorem 7.1.2 follows from the following result:

Proposition 7.2.1. *Let $\mathscr{H}$ be a (r, R, p, q)- sensitive hash family on the Hamming cube $(\{0, 1\}^d, \|\cdot\|_1)$. Assume that r is an odd integer and that $R < \frac{d}{2}$. Then*

$$p \le \left(q + e^{-\frac{1}{d}\left(\frac{d}{2}-R\right)^2}\right)^{\frac{e^{2r/d}-1}{e^{2r/d}+1}} \quad .$$

Choosing $R \approx \frac{d}{2} - \sqrt{d \log d}$ and $r \approx R/c$ in Proposition 7.2.1, and letting $d \to \infty$, yields Theorem 7.1.2 in the case $s = 1$. The case of general $s \geq 1$ follows from the fact that for $x, y \in \{0,1\}^d$, $\|x - y\|_s = \|x - y\|_1^{1/s}$.

Remark 7.2.1. Proposition 7.2.1 implies non-trivial lower bounds on $\frac{\log(1/p)}{\log(1/q)}$ for any (r, cr, p, q)-sensitive hash family on $(\{0,1\}^d, \|\cdot\|_1)$ even if q is allowed to depend on d. Observe that with the definition given in (7.1), Theorem 7.1.2 implies such a lower bound only for constant q. But, Proposition 7.2.1 is much stronger, and implies a bound which asymptotically coincides with the lower bound in 7.1.2 for every $q \geq 2^{-o(d)}$.

The proof of Proposition 7.2.1 will be broken into a few lemmas.

Lemma 7.2.1. *Let $\mathscr{H}$ be a (r, R, p, q)-sensitive hash family on the Hamming cube $(\{0,1\}^d, \|\cdot\|_1)$, and fix $x \in \{0,1\}^d$. Then*

$$\mathbb{E}\left|\mathscr{H}^{-1}\left(\mathscr{H}(x)\right)\right| \leq \sum_{k=0}^{\lfloor R \rfloor} \binom{d}{k} + q \cdot \sum_{k=\lfloor R \rfloor+1}^{d} \binom{d}{k} .$$

Proof. We simply write

$$\begin{aligned} \mathbb{E}\left|\mathscr{H}^{-1}\left(\mathscr{H}(x)\right)\right| &= \sum_{u \in \{0,1\}^d} \mathbf{Pr}[\mathscr{H}(u) = \mathscr{H}(x)] \\ &\leq \left|\{u \in \{0,1\}^d : \|u - x\|_1 \leq R\}\right| + q \cdot \left|\{u \in \{0,1\}^d : \|u - x\|_1 > R\}\right| \\ &= \sum_{k=0}^{\lfloor R \rfloor} \binom{d}{k} + q \cdot \sum_{k=\lfloor R \rfloor+1}^{d} \binom{d}{k} . \end{aligned}$$

□

Corollary 7.2.1. *Assume that $R < \frac{d}{2}$. Then, using the notation of Lemma 7.2.1, we have that*

$$\mathbb{E}\left|\mathscr{H}^{-1}\left(\mathscr{H}(x)\right)\right| \leq 2^d \left(q + e^{-\frac{1}{d}\left(\frac{d}{2} - R\right)^2}\right) .$$

Proof. This follows from Lemma 7.2.1 and the standard estimate $\sum_{k \leq \frac{d}{2} - a} \binom{d}{k} \leq 2^d \cdot e^{-\frac{a^2}{d}}$. □

Lemma 7.2.2 (Random walk lemma)**.** *Let r be an odd integer. Given $\emptyset \neq B \subseteq \{0,1\}^d$, consider the random variable $Q_B \in \{0,1\}^d$ defined as follows: Choose a point $z \in B$*

uniformly at random, and perform r-steps of the standard random walk on the Hamming cube starting from z. The point thus obtained will be denoted Q_B. Then

$$\mathbf{Pr}[Q_B \in B] \leq \left(\frac{|B|}{2^d}\right)^{\frac{e^{2r/d}-1}{e^{2r/d}+1}} .$$

Proof. We begin by recalling some background and notation on Fourier analysis on the Hamming cube. Given $S \subseteq \{1, \dots d\}$, the Walsh function $W_S : \{0,1\}^d \to \{-1,1\}$ is defined by

$$W_S(u) = (-1)^{\sum_{j\in S} u_j} .$$

For $f : \{0,1\}^d \to \mathbb{R}$ we set

$$\widehat{f}(S) = \frac{1}{2^d} \sum_{u\in\{0,1\}^d} f(u) W_S(u) ,$$

so that f can be decomposed as follows:

$$f = \sum_{S\subseteq\{1,\dots,d\}} \widehat{f}(S) W_S .$$

For every $f, g : \{0,1\}^d \to \mathbb{R}$ we write

$$\langle f, g\rangle = \frac{1}{2^d} \sum_{u\in\{0,1\}^d} f(u) g(u) .$$

By Parseval's identity,

$$\langle f, g\rangle = \sum_{S\subseteq\{1,\dots,d\}} \widehat{f}(S)\widehat{g}(S) .$$

For $\varepsilon \in [0,1]$ the Bonami-Beckner operator T_ε is defined as

$$T_\varepsilon f = \sum_{S\subseteq\{1,\dots,d\}} \varepsilon^{|S|} \widehat{f}(S) W_S .$$

The Bonami-Beckner inequality [21, 13] states that for every $f : \{0,1\}^d \to \mathbb{R}$,

$$\sum_{S\subseteq\{1,\dots,d\}} \varepsilon^{2|S|}\widehat{f}(S)^2 = \|T_\varepsilon f\|_2^2 = \frac{1}{2^d}\sum_{u\in\{0,1\}^d}(T_\varepsilon f(u))^2 \le \|f\|_{1+\varepsilon^2}^2 = \left(\frac{1}{2^d}\sum_{u\in\{0,1\}^d} f(u)^{1+\varepsilon^2}\right)^{\frac{2}{1+\varepsilon^2}} .$$

Specializing to the indicator of $B \subseteq \{0,1\}^d$ we get that

$$\sum_{S\subseteq\{1,\dots,d\}} \varepsilon^{2|S|}\widehat{\mathbf{1}_B}(S)^2 \le \left(\frac{|B|}{2^d}\right)^{\frac{2}{1+\varepsilon^2}} . \tag{7.3}$$

Now, let P be the transition matrix of the standard random walk on $\{0,1\}^d$, i.e. $P_{uv} = 1/d$ if u and v differ in exactly one coordinate, $P_{uv} = 0$ otherwise. By a direct computation we have that for every $S \subseteq \{1,\dots,d\}$,

$$PW_S = \left(1 - \frac{2|S|}{d}\right)W_S ,$$

i.e. W_S is an eigenvector of P with eigenvalue $1 - \frac{2|S|}{d}$. The probability that the random walk starting form a random point in B ends up in B after r steps equals

$$\begin{aligned}
\mathbf{Pr}[Q_B \in B] &= \frac{1}{|B|}\sum_{a,b\in B}(P^r)_{ab} \\
&= \frac{2^d}{|B|}\langle P^r\mathbf{1}_B, \mathbf{1}_B\rangle \\
&= \frac{2^d}{|B|}\sum_{S\subseteq\{1,\dots,d\}} \widehat{\mathbf{1}_B}(S)^2\left(1-\frac{2|S|}{d}\right)^r \\
&\le \frac{2^d}{|B|}\sum_{\substack{S\subseteq\{1,\dots,d\}\\ |S|\le d/2}} \widehat{\mathbf{1}_B}(S)^2\left(1-\frac{2|S|}{d}\right)^r ,
\end{aligned}$$

where we used the fact that r is odd (i.e. we dropped negative terms).

Thus, using (7.3) we see that

$$\mathbf{Pr}[Q_B \in B] \le \frac{2^d}{|B|}\sum_{S\subseteq\{1,\dots,d\}} \widehat{\mathbf{1}_B}(S)^2 \cdot e^{-2r|S|/d} \le \frac{2^d}{|B|}\cdot\left(\frac{|B|}{2^d}\right)^{\frac{2}{1+e^{-2r/d}}} = \left(\frac{|B|}{2^d}\right)^{\frac{1-e^{-2r/d}}{1+e^{-2r/d}}} .$$

□

Proof of Proposition 7.2.1. Assume that r is an odd integer and $R < \frac{d}{2}$. For $x \in \{0,1\}^d$ let $W_r(x) \in \{0,1\}^d$ be the random point obtained by preforming a random walk for r steps starting at x. Since $\|x - W_r(x)\|_1 \le r$ we know that $\mathbf{Pr}[\mathscr{H}(W_r(x)) = \mathscr{H}(x)] \ge p$. Taking expectation with respect to the uniform probability measure on $\{0,1\}^d$ we deduce that

$$\begin{aligned}
p &\le \mathbb{E}_{x\in\{0,1\}^n} \mathbf{Pr}\left[\mathscr{H}(W_r(x)) = \mathscr{H}(x)\right] \\
&= \mathbb{E}_{\mathscr{H}} \mathbf{Pr}\left[x \in \{0,1\}^n :\ W_r(x) \in \mathscr{H}^{-1}(\mathscr{H}(x))\right] \\
&= \mathbb{E}_{\mathscr{H}} \sum_{k\in\mathbb{N}} \mathbf{Pr}\left[x \in \{0,1\}^n :\ W_r(x) \in \mathscr{H}^{-1}(\mathscr{H}(x)) \ \wedge\ \mathscr{H}(x) = k\right] \\
&= \mathbb{E}_{\mathscr{H}} \sum_{k\in\mathbb{N}} \frac{\left|\mathscr{H}^{-1}(k)\right|}{2^d} \mathbf{Pr}\left[Q_{\mathscr{H}^{-1}(k)} \in \mathscr{H}^{-1}(k)\right] \\
&\le \mathbb{E}_{\mathscr{H}} \sum_{k\in\mathbb{N}} \frac{\left|\mathscr{H}^{-1}(k)\right|}{2^d} \cdot \left(\frac{\left|\mathscr{H}^{-1}(k)\right|}{2^d}\right)^{\frac{e^{2r/d}-1}{e^{2r/d}+1}} &(7.4)\\
&= \mathbb{E}_{\mathscr{H}} \mathbb{E}_{x\in\{0,1\}^d} \left(\frac{\left|\mathscr{H}^{-1}(\mathscr{H}(x))\right|}{2^d}\right)^{\frac{e^{2r/d}-1}{e^{2r/d}+1}} \\
&\le \mathbb{E}_{x\in\{0,1\}^d} \left(\frac{\mathbb{E}_{\mathscr{H}}\left|\mathscr{H}^{-1}(\mathscr{H}(x))\right|}{2^d}\right)^{\frac{e^{2r/d}-1}{e^{2r/d}+1}} &(7.5)\\
&\le \left(q + e^{-\frac{1}{d}\left(\frac{d}{2}-R\right)^2}\right)^{\frac{e^{2r/d}-1}{e^{2r/d}+1}}, &(7.6)
\end{aligned}$$

where in (7.4) we used Lemma 7.2.2, in (7.5) we used Jensen's inequality, and in (7.6) we used Corollary 7.2.1. □

Chapter 8

Space Lower Bounds for finding Frequent Elements in a Stream

8.1 Introduction

The data stream model of computation has evolved into an indispensable tool to cope with processing massive data sets. In this model, the input is processed as and when it arrives by a computational device that has only a limited storage space. The complexity in this model is typically measured in terms of the space required by the algorithm; this is often quantified as a function of the length of the input (n) and the size of the domain from which the elements of the inputs are drawn (m). Data stream algorithms are deemed interesting and usable only if their space requirement is sub-linear in n; this is especially so since n is assumed to be extremely large. Efficient data stream algorithms are out of question for most problems if neither randomization nor approximation is permitted. Sub-linear space data stream algorithms have been developed for several statistical data analysis primitives, ranging from frequency moments and L_p differences to histograms. For an excellent account of the important results in the area, see the recent monograph by Muthukrishnan [93] and the survey by Babcock et al [9].

Motivation. Even though many problems are shown to have efficient data stream algorithms, it is provably impossible to obtain $o(n)$ space algorithms for many other interesting data processing problems, even with the allowances of approximation and randomization. This is a consequence of both the limitations of the data stream model itself and the inherent difficulty of the underlying problem. Two important problems that fall in this doomed class are the following.

(1) *MostFrequent* problem. Given an input sequence $X = x_1, \ldots, x_n$, output the element(s) that occur the most in the sequence X. This is an important and recurring problem in databases, information retrieval, and web analysis; for instance, search engines would like to quickly distill the frequently-issued queries from their daily query logs. Unfortunately, Alon, Matias, and Szegedy [8] showed an $\Omega(n)$ lower bound for estimating the frequency of the largest item given an arbitrary data stream. Despite this seemingly insurmountable lower bound, a wide variety of heuristics that work very well in practice have been proposed [59, 60, 56].

(2) *InnerProduct* problem. Given two vectors $X = x_1, \ldots, x_n$ and $Y = y_1, \ldots, y_n$ arriving in an arbitrary interleaved order, compute their inner product $\langle X, Y \rangle$. This is a central problem in database join size estimation [4, 93] and plays a role in information retrieval as a measure of document similarity. Inner product is a notoriously difficult quantity to approximate because of the potential "cancellations": intuitively, it is very hard to obtain a relative-error approximation to the inner product value when it is very close to zero. By an application of the well-known communication complexity lower bound for the inner product [29, 79], it is easy to obtain an $\Omega(n)$ space lower bound to approximate the inner product.

Instance-specific bounds. Given that the above two problems have discouraging worst-case lower bounds of $\Omega(n)$, can anything be salvaged at all? We note the following two results. Charikar, Chen, and Colton [27] obtained an algorithm along with a performance guarantee for a relaxed version of the *MostFrequent* problem — given a parameter k, it suffices to output a list of k elements such that each element in the list occurs with frequency almost more than the k-th largest element — that runs in sub-linear space if the input elements were distributed according to a power law. Likewise, Gilbert et al [4] obtained an $O(1/\epsilon^2)$ algorithm to approximate the *InnerProduct* to within an additive factor of $\pm\epsilon \|X\|_2 \|Y\|_2$; subsequently, Cormode and Muthukrishnan [32] improved the space bound to $O(1/\epsilon)$ to obtain an additive error approximation of $\pm\epsilon\|X\|_1\|Y\|_1$. These results suggest that the worst-case lower bounds for these problems are perhaps way too pessimistic and stronger and better bounds could be stated if one were to take the properties of inputs themselves into account. To quote [93]: "...the goal would be to find workable approximations under suitable assumptions on input data distributions."

This is precisely the goal of this chapter. We propose the study of data stream algorithms through an *instance-specific* lens. Roughly speaking, the plan is to obtain optimal space bounds, both upper and lower, for data stream algorithms in terms of the *actual* properties of the input sequence than just its length, n. We call these the *instance-specific data stream bounds*.

Our contributions. As a concrete instantiation of our framework, we obtain the following results for the two above mentioned problems.

(1) For the *MostFrequent* problem, we show a tight space bound (up to logarithmic factors) of $\tilde{\Theta}(F_2/F_\infty^2)$, where F_2 is the second frequency moment of the input sequence and F_∞ is the frequency of the most frequent input element. The upper bound can be inferred from the algorithm of Charikar, Chen, and Colton [27]; see also [75]. We show a matching lower bound for one-pass data stream algorithms. The lower bound is established by information-theoretic arguments as in Bar-Yossef et al [18] and consists of two steps. The first step involves obtaining an information complexity lower bound for a functional version of set disjointness. The second step constructs separate instances of the set disjointness problem, carefully interleaves them, and uses the lower bound from the first step together with additional information-theoretic arguments to obtain the final lower bound. In particular, our bounds show the precise behavior of the space requirements as a function of the second and the infinity frequency moments of the input. In particular, it shows that the algorithm in [27] for Zipf distributions is essentially optimal.

(2) For the *InnerProduct* problem, we show a tight space bound of $1/\langle X,Y\rangle^2$, where $\langle X,Y\rangle$ is the inner product of X and Y. The upper bound essentially follows from the algorithm based on moments estimation [4]. We show the matching lower bound for one-pass data stream algorithms by a reduction to a communication complexity problem based on a gap version of the Hamming distance. Our bound not only formalizes the intuition that

the inner product becomes easy to approximate if it is bounded away from zero, but also quantifies the exact relationship between the space complexity and the value of the inner product. Our method also shows that the algorithms in [4] and [32] are essentially optimal.

Notation. Let $X = x_1, \ldots, x_n$ denote the input sequence of length n. For the *MostFrequent* problem, we assume that the input sequence X arrives in an arbitrary order. For the *InnerProduct* problem, we assume that the input sequences X and Y arrives in an arbitrary order with arbitrary interleaving of elements in X and Y.

We work with randomization approximation algorithms, i.e., algorithms that given an error parameter $\epsilon > 0$ and a confidence parameter $\delta > 0$, output, output a value that is an $(1 \pm \epsilon)$ multiplicative approximation to the correct value, with probability at least $1 - \delta$.

We use $\tilde{O}, \tilde{\Omega}, \tilde{\Theta}$ to suppress factors that are logarithmic in n.

8.2 Bounds for the *MostFrequent* problem

First, we formally state the *MostFrequent* problem. Let $F_\infty = F_\infty(X)$ denote the frequency of the most frequent element in the input sequence X. Let $F_2 = F_2(X)$ denote the second frequency moment of the input sequence X.

Problem 8.2.1 (*MostFrequent*)**.** *In the MostFrequent problem, given an n element sequence X, the objective is to find the (or one of the) most frequent element with frequency $\|F\|_\infty$ in one pass over the input.*

8.2.1 Upper bound

For sake of completeness, we briefly discuss the known two-pass algorithm for the *MostFrequent* problem.

The algorithm maintains h hash tables of size b buckets each. In the first pass, each input element is hashed into the h tables using independent hash functions, keeping a counter per bucket. In the second pass, input elements that hash to a large number buckets with a high counter value — above a threshold τ — are reported as frequent elements.

Here is a glimpse of the analysis. Focusing on one hash table, if there are b buckets, then the expected count in a bucket cell is n/b. The variance of this count can be computed to be at most F_2/b. But the expected count of the bucket with the most frequent element is at least $n/b + F_\infty(1 - 1/b)$. So, as long as the gap between the expectations $F_\infty(1 - 1/b)$ is at least $c \cdot \sqrt{F_2/b}$ for sufficiently chosen constant $c > 0$, we can set the threshold τ so that there is a constant gap in the probability of a count exceeding the threshold for the frequent element versus any other element. This gives a bound on the number of buckets as $b > \Omega(F_2/F_\infty^2)$. By using several (constant) such tables and using the median count, the probability gap can be amplified arbitrarily.

Theorem 8.2.2 ([27])**.** *There is a randomized data stream algorithm for approximating the MostFrequent problem and uses space $\tilde{O}(F_2/F_\infty^2)$.*

8.2.2 Tools for the lower bound

We assume the reader is familiar with communication complexity and information theory. See the book by Kushilevitz and Nisan [79] for the former and the book by Cover and Thomas [38] for the latter.

In our usage, $H(\cdot)$ will denote the entropy and $I(\cdot;\cdot)$ will denote the mutual information between two random variables. Some basic properties of these information-theoretic quantities can be found in [38].

First, we need the following simple lemma.

Lemma 8.2.1. *For random variables A, B, X, Y such that $X \perp Y$, we have*

$$H(A) + H(B) \geq I(AB; XY) + I(A; B) \geq I(AB; X) + I(A; Y) + I(B; Y).$$

Proof. The first inequality follows easily since

$$H(A) + H(B) \;=\; H(AB) + I(A; B) \;\geq\; I(AB; XY) + I(A; B).$$

To show the second inequality, we first show

$$\begin{aligned} I(AB;XY) &= H(XY) - H(XY|AB) \\ &= H(X) + H(Y) - H(XY|AB) \quad (\text{by } X \perp Y) \\ &\geq H(X) + H(Y) - H(X|AB) - H(Y|AB) \quad (\text{by subadditivity}) \\ &= I(AB;X) + I(AB;Y). \end{aligned} \tag{8.1}$$

Next, we show that

$$\begin{aligned} I(B;Y|A) + I(A;B) &= H(B|A) - H(B|A,Y) + H(B) - H(B|A) \\ &= H(B) - H(B|A,Y) \\ &\geq H(B) - H(B|Y) \quad (\text{by dropping condition}) \\ &= I(B;Y). \end{aligned} \tag{8.2}$$

Finally, the second inequality follows as

$$\begin{aligned} I(AB;XY) + I(A;B) &\geq I(AB;X) + I(AB;Y) + I(A;B) \quad \text{by (8.1)} \\ &= I(AB;X) + I(A;Y) + I(B;Y|A) + I(A;B) \\ &\geq I(AB;X) + I(A;Y) + I(B;Y). \quad \text{by (8.2)} \end{aligned}$$

□

Theorem 8.2.3. *Given random variables $U_0, \ldots, U_{2^k-1}$ and $k+1$ random variables $X_0, \ldots, X_k$ that are independent. Consider a binary tree with $U_0, \ldots, U_{2^k-1}$ at the root and each left (right) child labeled with the left (right) half of the parents label. That is, the j-th node at the i-th level is labeled by V_{ij}, $i \in \{0, \ldots, k\}$, $j \in j(k,i) = \{0, \ldots, 2^{(k-i)} - 1\}$ denoted by $U_{j2^i} U_{j2^i+1} \cdots U_{(j+1)2^i-1}$. Then*

$$\sum_{j=0}^{2^k-1} H(U_j) \geq \sum_{i\in[k], j\in j(k,i)} I(V_{ij}; X_i).$$

Proof. We prove by induction on k. The base case for $k = 1$ is given by the Lemma 8.2.1.

For notational simplicity, we first illustrate the proof for $k = 2$.

$$\begin{aligned}
&H(U_0) + H(U_1) + H(U_2) + H(U_3) \\
&\quad = H(U_0U_1) + H(U_2U_3) + I(U_0; U_1) + I(U_2; U_3) \\
&\quad \geq I(U_0U_1U_2U_3; X_0) + I(U_0U_1; X_1X_2) + I(U_2U_3; X_1X_2) + I(U_0; U_1) + I(U_2; U_3) \\
&\qquad\quad \text{(by Lemma 8.2.1)} \\
&\quad \geq I(U_0U_1U_2U_3; X_0) + I(U_0U_1; X_1) + I(U_0; X_2) + I(U_1; X_2) + I(U_2U_3; X_1) + I(U_2; X_2) + I(U_3; X_2). \\
&\qquad\quad \text{(by Lemma 8.2.1 applied twice)}
\end{aligned}$$

This completes the proof for $k = 2$. For general $k > 2$, we proceed by induction. Let X' be a sequence of k independent random variables such that $X'_i = X_i$ for $i = 0, \ldots, k-1$ and $X'_{k-1} = X_{k-1}X_k$.

$$\begin{aligned}
\sum_{j=0}^{2^k-1} H(U_j) &= \sum_{j'=0}^{2^{k-1}-1} H(U_{2j'}U_{2j'+1}) + \sum_{j'=0}^{k} I(U_{2j'}; U_{2j'+1}) \\
&\geq \sum_{i\in[k-1], j\in j(k-1,i)} I(V_{ij}; X'_i) + \sum_{j'=0}^{k} I(U_{2j'}; U_{2j'+1}) \\
&\qquad \text{(by induction hypothesis and using } X'\text{)} \\
&= \sum_{i\in[k-2], j\in j(k-2,i)} I(V_{ij}; X_i) + \sum_{j'=0}^{k} I(U_{2j'}U_{2j'+1}; X_{k-1}X_k) + \sum_{j'=0}^{k} I(U_{2j'}; U_{2j'+1}) \\
&= \sum_{i\in[k-2], j\in j(k-2,i)} I(V_{ij}; X_i) + \sum_{j'=0}^{k} I(U_{2j'}U_{2j'+1}; X_{k-1}) + \sum_{j'=0}^{k} I(U_{2j'}; X_k) + I(U_{2j'+1}; X_k) \\
&\qquad \text{(by Lemma 8.2.1 applied } k \text{ times)} \\
&= \sum_{i\in[k], j\in j(k,i)} I(V_{ij}; X_i).
\end{aligned}$$

□

Note that the above theorem easily extends to a conditional version where all entropies and mutual informations are conditioned on some variable. Such a conditional version will be used later.

We consider both two-party and multi-party communication protocols. A protocol specifies the rules of interaction between the players and a transcript $\Pi(X)$ contains the messages exchanged between the players upon the input X. If A is a random variable corresponding to a distribution on the inputs, then $I(A;\Pi(A))$ measures the mutual information between the input and the transcript; this is called *information cost*. The *information complexity* of a function is the minimum information cost of a protocol that correctly computes the function. Conditional version of these quantities are defined in an analogous manner; the reader is referred to [18] for more details.

8.2.3 The *IndexedAnd* problem

We will reduce the problem to the following problem that involves t players each with a bit-vector. The objective is to find an index in the vector so that all players have a '1' in that index.

Problem 8.2.4 (*IndexedAnd*)**.** *In the IndexedAnd problem, there are t players each with a binary vector of length n. Viewing the input to the players as a $n \times t$ binary matrix, the goal of the players is to output an index i, if any, so that AND of that row is 1; here a i-th row corresponds to the i-th bit of all players. If there is no such row with all 1's, then any output is valid.*

Note that *IndexedAnd* is thus a functional version of the t-player set disjointness problem. Despite this close relationship, it is not clear if the communication lower bounds for set disjointness directly translate to the *IndexedAnd* problem. We consider the following instances for the problem.

(1) NO instance. In the inputs corresponding to the NO instance, each row has one 1 with probability 1/2 at a random column. As stated before, any output is considered valid if the input is a NO instance.

(2) YES instance. The inputs corresponding to the YES instance is the same as above except that one row as all 1's; the correct answer in this case would be the index of that row.

We also need the one-bit version of the *IndexedAnd* problem.

Problem 8.2.5 (*And*)**.** *In the And problem, there are two players with one bit each. The objective is to compute the conjunction of these two bits.*

A random NO instance can be generates as follows. Let $D = (D_1, \ldots, D_n)$ where D_i denotes the column number in row i that is set to 1 with probability 1/2; so each D_i takes a random value in the range $1..t$. Let $X \in \{0,1\}^{n \times t}$ be the random variable denoting the input and let $X_1, \ldots X_n$ denote the rows of X; so $X_{ij} = 1$ with probability $1/2$ if $D_i = j$ and always 0 otherwise. Note that $X_1, \ldots, X_n$ are independent random variables, even conditioned on D. Let $\mathrm{IC} = \min_\Pi I(X; \Pi(X)|D)$ over all protocols that solve the above promise instances of *IndexedAnd*. Let Π be the (randomized) protocol that achieves the minimum condition information cost.

8.2.4 Lower bound for the *IndexedAnd* problem

We show the main theorem in this section.

Theorem 8.2.6. $\mathrm{IC}_{IndexedAnd} \geq \Omega(n \cdot \mathrm{IC}_{And})$.

To show this theorem, we first argue that there is a row i about which the protocol doesn't reveal much information. First let $\mathrm{IC} = \mathrm{IC}_{IndexedAnd}$.

Lemma 8.2.2. *There is a row $i \in [n]$ such that* $\mathbf{Pr}$[*i is output in NO instance*] $\leq 2/n$ *and* $I(X_i; \Pi \mid D) \leq (2/n) \cdot \mathrm{IC}$.

Proof. Since $X_1, \ldots, X_n$ are independent given D, it follows (à la [18]) that

$$\begin{aligned}
\mathrm{IC} &= I(X; \Pi(X) \mid D) \\
&= H(X_1, \ldots, X_n \mid D) - H(X_1, \ldots, X_n \mid \Pi(X), D) \\
&\geq \left(\sum_{i=1}^{n} H(X_i \mid D) \right) - \left(\sum_{i=1}^{n} H(X_i \mid \Pi(X), D) \right) \\
&= \sum_{i=1}^{n} I(X_i; \Pi(X) \mid D).
\end{aligned}$$

Now, there are at least $n/2$ rows i so that $\mathbf{Pr}$[i is output in NO instance] $\leq 2/n$. Also, by averaging, for at least one of these, $I(X_i; \Pi \mid D) \leq 2 \cdot \mathrm{IC}/n$. $\square$

Let D_{-i} denote the entries of D except for the row i.

Lemma 8.2.3. *There exist values for D_{-i}, so that with those values when row i is chosen randomly as in NO case* $\mathbf{Pr}[$*i is output in NO case*$] \leq 6/n$ *and* $I(X_i; \Pi | D_i, D_{-i} = d_{-i}) \leq 6 \cdot \mathrm{IC}/n$.

Proof. From the Lemma 8.2.2, $\mathbf{Pr}[i$ is output in NO instance$] \leq 2/n$. Let us set the remaining rows (other than i) randomly using D_{-i} and private coin tosses. By averaging, for a random value d_{-i} of D_{-i}, with probability at least 2/3, $\mathbf{Pr}[i$ is output in NO instance$] \leq 6/n$. Once again from Lemma 8.2.2, $I(X_i; \Pi \mid D_i D_{-i}) \leq 2 \cdot \mathrm{IC}/n$. This is equal to $E_{d_{-i}}[I(X_i; \Pi | D_i, D_{-i} = d_{-i}] \leq 2 \cdot \mathrm{IC}/n$ over different fixed values d_{-i} of D_{-i}. So with probability at least 2/3 for a random fixed value d_{-i} of D_{-i}, we have $I(X_i; \Pi | D_i, D_{-i} = d_{-i}) \leq 6 \cdot \mathrm{IC}/n$. So with at least 1/3 probability both conditions are satisfied. □

Proof of Theorem 8.2.6. To complete the proof, we provide a reduction. Given an instance of the one-bit *And* problem, we embed the problem in row i given by Lemma 8.2.2 and set hard coded values for D_{-i} other rows given by Lemma 8.2.3 . The actual values of the rows other than i can be set using private randomness based on the fixed value of D_{-i}. Answer YES if output equals i. In the NO instance of one-bit AND, the probability of outputting YES is at most $6/n$. In the YES instance it is 1. Therefore, $6 * \mathrm{IC}/n \geq \mathrm{IC}_{And}$ and thus $\mathrm{IC} \geq \Omega(n) * \mathrm{IC}_{And}$. □

A lower bound on the information complexity of the one-bit *And* problem was proved in [18, 31].

Theorem 8.2.7 ([18, 31])**.** $\mathrm{IC}_{And} \geq \tilde{\Omega}(1/t)$.

8.2.5 Lower bound for the *MostFrequent* problem

We now prove a lower bound on the space required for the *MostFrequent* problem. The set up is that we are given a vector $f = \{(n_1, k_1), (n_2, k_2), \ldots, n_s, k_s), (1, t)\}$ where n_i elements occur with frequency k_i ($i \in [s]$) and the most frequent element occurs with frequency t that is the greater than all other frequencies k_i.

For the purposes of proving a lower bound we will make the following relaxations on the frequency vectors and the input instance. We assume that all the frequencies k_i and t are powers of 2; this will in fact show that it is even hard to approximate the most frequent element to within factor 2.

Allowing randomness in frequency counts. Next, instead of producing a stream with frequency vector exactly f, we will allow some randomness in the number of elements with the different frequencies – the number of elements with frequency k_i will be sharply concentrated around n_i. Precisely, frequency counts are obtained by drawing elements from a universe U that is a disjoint union $U_1 \cup U_2 \cup \cdots \cup U_s$ where $|U_i| = 2n_i$. Each element of U_i occurs independently and randomly in the stream either 0 or k_i times with probability $1/2$. So the expected count of number of elements with frequency k_i is n_i; if n_i is large, this count is sharply concentrated around its n_i. Each element of the universe U occurs in the stream independently and randomly in this way, except that one element may occur t times.

If there is such an element in the stream with frequency t the algorithm is required to identify that element; if no such element occurs any output is allowed.

Theorem 8.2.8. *The space complexity for the above problem is* $\tilde{\Omega}(\sum_i n_i k_i^2/t^2)$.

We prove the lower bound by reduction from the *IndexedAnd* problem.

Reduction idea. If all the frequencies k_i were the same, say k (i.e., $s = 1$), then the reduction is easy: given an instance of the *IndexedAnd* problem with $2n$ rows and t/k players we can construct a sub-stream for each player where the element is present or absent based on the bit in the input. Further each element is replicated k times. The final stream is the union of the sub-streams for each player, where each element occurs k times and the element corresponding to the all 1's row occurs t times. The count of the number of elements with frequency k is sharply concentrated around n. By Theorem 8.2.6, the total communication complexity is at least $\tilde{\Omega}(\frac{n}{(t/k)})$. So the average communication between successive players among the t/k players is $\tilde{\Omega}(\frac{n}{(t/k)^2}) = \tilde{\Omega}(nk^2/t^2)$

Reduction for multiple frequencies. When the number of frequencies s is more than 1 the reduction is more complicated. Instead of working with one instance of the *IndexedAnd* problem we will look at s instances each with t/k_i players. We will then interleave the substreams corresponding the different players over all the instances in such as way as to be able to add up their individual communication complexities.

Formally, we prove the lower bound by contradiction. We assume there is a data stream algorithm to solve the *MostFrequent* problem and use it to obtain a communication protocol for *IndexedAnd*. Consider a collection of s random NO instances of the *IndexedAnd* problem, where the i-th instance has $2n_i$ rows and t/k_i players. We will show how a solution to *MostFrequent* can be used to solve a random YES instance of *IndexedAnd* implanted on any of these instances. Construct an instance of *MostFrequent* as follows.

By assumption $t = 2^{t'}$. Also assume without loss of generality that all the frequencies between 1 and t that are powers of 2 are present — this is easily achieved by setting the n_i corresponding to the absent ones to 0. So, $s = t', k_i = 2^{i-1}, i = 1, \dots, s$.

Interleaving the different inputs. A block in the stream is constructed for each player; each player in the i-th instance replicates his object k_i times. The i-th instance has $t/k_i = 2^{t'-i+1}$ blocks (equivalently, players). Thus, there are a total of $t/k_1 + \cdots + t/k_s = 2t-2$ blocks. Let b_{ij} denote the j-th block in the i-th instance. The stream is obtained by arranging these blocks in a specific order. The order is obtained recursively by arranging the 2, 4, 8, ..., t blocks of the different instances in a binary tree under a dummy root, such that 2 blocks appear just below the dummy root, 4 blocks appear one level down, and so on. The blocks are then laid down in the depth-first search traversal order. The first block in the first instance appears first; the second block in the first instance appears as the first block in the second half of the ordering. The other blocks are placed recursively in this fashion according to the depth-first search ordering.

Observe that if a row in any instance i has all entries set to 1's then that element occurs t times in the stream. And since the problem on the stream reports the identity of the most frequent element this would reveal the row identity, solving the corresponding YES instance of the *IndexedAnd* problem.

Let Y_i denote a random NO input for the i-th instance. Let $Y = (Y_1, \dots, Y_s)$. Let D denote the union of all the conditioning variables for all the instances. The first instance has t blocks. Let Π_r denote the output of the r-th block from the last instance, in the streaming algorithm. We will lower bound $\sum_r H(\Pi_r \mid D)$.

Define $V_{t'-1,0} = \Pi_0, \ldots, \Pi_{t-1}$.

Lemma 8.2.4. *$I(V_{t'-1,0}; Y_{t'-1} \mid D) \geq \tilde{\Omega}(n_{t'-1}/t)$.*

Proof. Assume that $I(V_{t'-1,0}; Y_{t'-1} \mid D)$ is not $\tilde{\Omega}(n_{t'-1}/t)$. Then we have a protocol for the first instance of the indexed-AND problem with better conditional information complexity which violates the earlier lower bound (Theorem 8.2.6). Given a *IndexedAnd* instance with $n_{t'-1}$ elements and t players, it can be implanted in the first instance of the reduction. The other instances can be generated as private random coin tosses based on the conditioning variable D. □

Similarly define $V_{t'-2,0} = \Pi_0\Pi_2 \ldots \Pi_{t-2}$ and $V_{t'-2,1} = \Pi_1\Pi_3 \ldots \Pi_{t-1}$.

Then again $I(V_{t'-2,0}; Y_{t'-2} \mid D) > \tilde{\Omega}(n_{t'-2}/(t/k_{t'-2}))$ and $I(V_{t'-2,1}; Y_{t'-2} \mid D) > \tilde{\Omega}(n_{t'-2}/(t/k_{t'-2}))$ since if either of these is not true, we get an improved conditional information complexity bound for the second instance.

Similarly we get $2k_{t'-1-i} = 2^i$ inequalities for $V_{t'-1-i,0}, \ldots, V_{t'-1-i,2^i-1}$. The V's are defined recursively as before by taking alternate Π_i's from the parents.

The idea is to use Theorem 8.2.3. The binary tree of depth t' has $V_{t'-1,0} = \Pi_0, \ldots, \Pi_{t-1}$ at its root. Each left (right) child has the even (odd) alternate variables of its parent. Although this is a little different from the setting in Theorem 8.2.3, the theorem still applies as it is independent of the order in which the variables appear. Plugging into Theorem 8.2.3 we get

$$\sum_i H(\Pi_i \mid D) \geq \sum 2 \cdot k_{t'-1-i} \cdot \tilde{\Omega}(n_{t'-i-1}/(t/k_{t'-i-1})).$$

So the total information complexity is at least $\tilde{\Omega}(\sum n_i k_i^2/t)$. So the space complexity is at least $\tilde{\Omega}(\sum n_i k_i^2/t^2) = \tilde{\Omega}(F_2/F_\infty^2)$.

8.3 Bounds for the *InnerProduct* problem

We consider the problem of computing the (real) inner product $\langle X, Y\rangle$ of two n-dimensional ± 1 vectors X and Y. We claim that the space bound for this problem is $\Theta(1/\langle X, Y\rangle^2)$.

Problem 8.3.1 (*InnerProduct*)**.** *The InnerProduct problem involves two players with unit vectors in n dimensions and a number δ so that the inner product value is either δ or $-\delta$. The objective is distinguish between the two inner products in the data stream model.*

8.3.1 Upper bound

We outline the known algorithm for approximating inner product of two unit vectors in the data stream model using space $O(1/\langle X, Y\rangle^2)$.

Lemma 8.3.1. *There is an algorithm for approximating the InnerProduct of any two unit vectors X, Y using space $O(1/\langle X, Y\rangle^2)$.*

Proof. From [4], we know that *InnerProduct* can approximated in space $\tilde{O}(1/\epsilon'^2)$ to within an additive factor $\pm\epsilon'||X||_2||Y||_2$. Since the vectors are unit norm in our case, to get a multiplicative approximation to within ϵ, we need to set $\epsilon' = \epsilon \cdot \langle X, Y\rangle$, proving the lemma. □

8.3.2 Lower bound

Next, we show a matching lower bound.

Theorem 8.3.2. *For every value δ and for sufficiently large n, any data stream algorithm to approximate InnerProduct of two vectors X and Y such that $\langle X, Y\rangle = \pm\delta$ will require $\Omega(1/\delta^2)$ space.*

Proof. We use the following one-way space lower bound for the gap Hamming distance problem. Consider d-dimensional bit vectors A, B with the promise that $H(A, B) \geq (1/2 - c_1)d$ or $H(A, B) \leq (1/2 - c_1)d - c_2\sqrt{d}$. Then, the communication complexity of deciding this problem is $\Omega(d)$ [74, 17]. Given a d-dimensional gap Hamming distance problem as above, we first pad it with extra dimensions so that the bounds are symmetric around the length of the vector. The number of dimensions is still $O(d)$ and the gap between the YES and NO are still $\Theta(\sqrt{d})$.

Interpreting them as ± 1 vectors and normalizing them to have two-norm one, the inner products for this promise problem is either $+\sqrt{1/d}$ or $-\sqrt{1/d}$. Let $\delta = \sqrt{1/d}$. So, if we can solve the inner product problem in $o(1/\delta^2)$ space, then we can solve the gap Hamming distance in $o(d)$ space, which is a contradiction. □

Chapter 9

Sketching algorithms for trees

9.1 Introduction

The proliferation of information on the internet creates a huge amount of data in different formats. In the absence of rigid standards, *ad hoc* methods are used by different entities to represent data. For instance, a book may be represented hierarchically as a collection of chapters and these in turn as a collection of sections. Such an abstraction may be represented as a tree in the emerging XML standard. Similarly directory structures may be viewed as hierarchical entities that can be communicated in different formats. It then becomes a challenge to compare two such information corpora. For example, the same information can be represented under different isomorphic permutations of the underlying directory structure. A database record could be presented with attributes in different order. In such cases, rather than looking at the exact tree structure representing the objects, it may be more appropriate to to check the similarity of the two tree structures under all possible isomorphic permutations. Again, to measure the similarity between two directory structures, it may be more appropriate to ignore the directory names and check if there is an ordering of the directory structure that minimizes mismatch between the file contents. However, computing the ordering that minimizes the mismatch on the tree data becomes intractable as the degree and height of the tree increase.

Sketching is a powerful tool for concise representation and comparison of complex and large data. For instance, strings and sets can be compressed into points in hamming space.

This typically yields sketches represented by bit-vectors. Such sketches, although are not identical to the original data, preserve many of their properties. For instance sketches can be used to measure containment and distance [10]. Given the plethora of applications for sketching, there have been several attempts to compute sketches of more complex data such as trees and graphs. Some examples include file system directories, XML DOM trees, and phylogenetic trees [115, 89, 1, 24, 117]. Other works of research have presented sketch based measurement of similarity between trees [1, 35]. In this work, we propose a sketch based algorithm via an Earth Mover's Distance metric for trees with provable guarantees.

9.1.1 Related Work

Similarity among trees has been widely studied in the context of edit distance between trees, i.e., the number of edits required to transform one tree into the other [81, 114, 117] while others have approached this problem of matching via alignment of trees and effectively computing the best alignment that minimized the objective function [77]. The distance function adopted in this work is similar in spirit and is based on the optimal superimposition that minimizes the number of mismatches in the set of leaf labels between the two trees. A more recent work of Augesten et al [1] uses *pq*-grams to match hierarchical data. A sketch of the tree is composed of a set of *pq*-gram profiles and sketch similarity is computed using well known set similarity measures. This approach, however, is not resistant to permutations in the leaf labels when $q = 1$. The other case of $q > 1$ generates mismatches that are amplified by the shingling approach adopted to generate the *pq*-grams, thus resulting in a poor match between trees even when they are similar.

9.1.2 Contributions of this study

We introduce the distance between arbitrary weighted trees as the best possible superimposition that results in the maximum number of matches between the trees. Specifically, we look at data objects that are represented using leaf-labeled trees denoting a set of elements at the leaves organized in a hierarchy. Our distance measure is equivalent to an Earth Mover's Distance measure since the leaf-labeled trees of height one can be viewed as sets and can be recursively extended to multi-level by viewing them as set of sets. For weighted trees, this is equivalent to recursive fractional weighted matching between the leaves according to the tree hierarchy. We propose sketching algorithms that generalize the concept of min-wise independent permutations to trees. We analyze the sketching algorithms in the framework of locality-sensitive hashing and show how our sketch functions can be used to perform nearest-neighbor search among trees.

Specifically, we use sketching algorithms on sets to compute their min-hashes and then propagate the min-hashes up the hierarchy. We show that propagating only one min-hash can result in poor similarity guarantees. For example, two trees that are far apart can produce very similar sketches using this scheme. Finally, we show that by propagating two min-hashes for each set of labels, we can compute similarity accurately between two trees in the context of locality-sensitive hashing.

9.2 Models and Definitions

In this section, we present the preliminaries for hierarchies as well as introduce notions of similarities between hierarchies. We classify trees as either labeled or unlabeled and ordered or unordered. In the labeled case, all nodes in the tree are labeled. A common example of such a hierarchy is a file system directory structure. Leaf-labeled trees where only the leaves are labeled have no labels on the internal nodes. For example, we could consider only the names and content of all files under a directory to compute the similarity between two directories. In such a case, we ignore the names of the directories themselves. An ordered tree has all leaves in a defined order. For example, a string can be considered an ordered tree of level one.

Observe that in practice two trees are considered similar even if one is obtained by a isomorphic reordering of nodes. In this case, we assume the trees to be unordered. We also assume that leaves are uniquely labeled.

Before we analyze the similarity measures for leaf-labeled trees, we introduce well-studied notions of similarities between unordered entities such as sets and how these notions have been successfully used in defining similarity measures between ordered sequences such as strings. Set similarity is computed using measures such as intersection and (symmetric) difference. String similarity is measured using edit distances. While there are many efficient algorithms for sketching sets such as min-hash computation [11], obtaining efficient sketching algorithms for strings with strong guarantees is much harder. The best known algorithm for sketching strings work by embedding strings into L_1 and the best known embedding has distortion $2^{O(\sqrt{\log n \log\log n})}$ [97] which is far less efficient compared to sketching algorithms for sets which produce almost no distortion.

Many of these algorithms are based on shingling [10], which essentially transform an ordered sequence of characters into a set of substrings known as *shingles*. Then by applying standard sketching algorithms on these sets the similarity between the original strings is estimated.

In this work, we present algorithms for unordered leaf-labeled weighted trees. We believe our algorithms may be extended to ordered trees just as shingling based methods extend algorithms on sets to algorithms on strings. Furthermore, we consider general trees with arbitrary shape with non-uniform degrees but of the same height. More generally, we allow the nodes to be weighted where the total weight of all children under a given node is 1.

Before we analyze similarity measures for trees, we introduce some of the basic concepts we employ in our sketching algorithms. First is the concept of sketching sets using the technique of min-wise independent permutations which are often referred to as min-hash [11]. The commonly used similarity measure for sets A and B is $sim(A, B) = \frac{|A \cap B|}{|A \cup B|}$ [11].

A weighted one-level tree can be viewed as a weighted set (a bag) instead of a set. In this case, the similarity measure is computed via min-hash computations on bags. This extension can be easily introduced by replacing a bag $A = \{(a_1, \alpha_1), (a_2, \alpha_2), \ldots, (a_n, \alpha_n)\}$ with integral weights by a set $\tilde{A} = \{a_{1,1}, a_{1,2}, \ldots, a_{1,\alpha_1}, \ldots, a_{n,1}, a_{n,2}, \ldots, a_{n,\alpha_n}\}$ where $a_{i,j}$ is an element obtained by concatenating the bag element a_i with a frequency j. Note that this transformation can be extended to bags with real weights as well. Under this transformation, we can easily show $|A \cap B| = |\tilde{A} \cap \tilde{B}|$ and $|A \cup B| = |\tilde{A} \cup \tilde{B}|$ [65, 62] where intersections(unions) over bags is obtained by assigning to each element a weight equal to the minimum(maximum) of its weights in the two bags $\tilde{A}$ and $\tilde{B}$. Thus, under a given permutation $\pi : [|U|] \to [|U|]$, we have $\mathbf{Pr}[MH_\pi(A) = MH_\pi(B)] = \frac{|A \cap B|}{|A \cup B|} = \mathbf{Pr}[MH_\pi(\tilde{A}) = MH_\pi(\tilde{B})] = \frac{|\tilde{A} \cap \tilde{B}|}{|\tilde{A} \cup \tilde{B}|}$ [65, 62].

Definition 4. *Let U denote the universal set. Given a set $A \subseteq U$ and a permutation $\pi : [|U|] \to [|U|]$, a min-hash denoted by $MH_\pi(A)$, is equal to $argmin_x\{\pi(x)|x \in A\}$. Essentially, $MH_\pi(A)$ is the element in A whose value in the permutation is the minimum. Alternately, let $f(x), x \in U$ be a real valued (hash) function that maps elements from the universe U to a real number randomly and uniformly in the interval* $[0, 1]$. *Then $MH_f(A) = argmin_x\{f(x)|x \in A\}$. Therefore, $MH_f(A)$ is the element in A whose hash value into the interval* $[0, 1]$ *is minimum. Note that this definition of min-hash can be applied to weighted sets or bags as well.*

A common measure used to compute similarity between sets of entities is the *Earth Mover's Distance*. We now define the Earth Mover's Distance in the presence of fractional weights as follows.

Definition 5 ([25])**.** *Let (X, δ) be a metric on a set X of weighted elements (x_i, w_i), $1 \le i \le n$, where $\delta(.,.)$ is the underlying distance measure between elements. Let $P(X)$ denote the a distribution of non-negative weights $u_1, u_2, \ldots, u_n$ on X such that $\sum_i u_i = 1$. The Earth Mover's Distance is a measure between two such distribution $P(X) = u_1, u_2, \ldots, u_m$ and $Q(X) = v_1, v_2, \ldots, v_m$. Specifically, the Earth Mover's Distance defines the optimal cost of transforming $P(X)$ into $Q(X)$ and can be formally stated as*

$$\begin{aligned} EMD(P(X), Q(X)) = &\quad min \sum_{ij} f_{ij}\delta(i, j) \\ \forall i \textstyle\sum_j f_{ij} = u_i & \\ \forall j \textstyle\sum_i f_{ij} = v_j, &\qquad and \\ \forall i, j\; f_{ij} \ge 0 & \end{aligned}$$

In fact the EMD corresponds to a fractional weighted matching in a bi-partite graph with the nodes on either side of an edge corresponding to elements of X with weights u_i and u_j respectively. Thus, the total weight of the matching edges at a node equals the weight of the point. This distance measure can be generalized to a distance measure on leaf-labeled trees of same height. Observe that a tree of height one can be viewed as a weighted set $(X, P(X))$ where X is the set of leaf labels and $P(X)$ denotes the weights on the leaves. The above EMD measure defines the distance between two such trees $(X, P(X))$

and $(X, Q(X))$ of height one. This can be extended to multi-level trees easily by treating X as the set of subtrees at height $h - 1$ and the weights being assigned to subtrees instead of leaves in one-level trees.

Sketching functions can be used to estimate similarity between two objects by comparing their sketches. Given a sketching function that results in a low matching probability when the underlying objects are dissimilar and a higher probability when the objects are similar, we can construct efficient data structures for approximate nearest neighbor search on a database of objects. Such sketching functions are also called *locality sensitive hash*(LSH) functions[42]. For a domain X of points with distance measure d, an LSH family of functions is defined as follows.

Definition 6. *A family $\mathcal{H} = h : X \rightarrow U$ is called $(r_1, r_2, p_1, p_2)-$ sensitive for d if for any $v, q \in X$*

- *if $d(v, q) \leq r_1$, then $\mathbf{Pr}_{\mathcal{H}}[h(q) = h(v)] \geq p_1$.*
- *if $d(v, q) > r_2$, then $\mathbf{Pr}_{\mathcal{H}}[h(q) = h(v)] \leq p_2$.*

Indyk and Motwani [69] show how LSH can be used for nearest neighbor searches in high dimensions.

Theorem 9.2.1 ([42]). *Suppose there is a (R, cR, p_1, p_2)-sensitive family $\mathcal{H}$ for a distance measure d. Given a query point with a nearest neighbor within distance R, the algorithm return a c-approximate nearest neighbor, i.e., a neighbor within distance cR. The algorithm uses an index of $O(N^{1+\rho})$ space, with query time dominated by $O(N^{\rho})$ distance computations where N is the size of the database of points and $\rho = \frac{\ln 1/p_1}{\ln 1/p_2}$ for $N > 1/p_2$.*

An information theoretic formulation of LSH resulting in linear size data structures has been studied in [100].

9.3 Similarity measures for trees

In this study we focus on weighted leaf-labeled trees of arbitrary shape with a given height h. The notion of similarity between two such trees is measured by the best possible superimposition that minimizes the number of mismatched leaf labels. Given two trees of height one, we use the EMD measure to compare the sets of leaves. We extend this recursively to trees of arbitrary height. The total weight of all children under any given node is at most 1. This can recursively be defined as follows.

Definition 7 (distance). *For two trees T_1 and T_2 of height one, we look at their weighted set of leaves given by $(X, P(X))$ and $(X, Q(X))$ where X is the universe of all leaf labels. Then we define the distance between the trees T_1 and T_2 to be equal to $EMD(P(X), Q(X))$. Using this base case, we can define the distance between any two trees of height h by viewing such trees as a weighted set of trees of height $h-1$. Further, if X denotes the space of all leaf-labeled trees of height $h-1$, then any such weighted set can be represented by a distribution. So, given two trees T_1 and T_2, we can obtain the corresponding distributions $P(X)$ and $Q(X)$ over trees of height $h-1$. We then define $d(T_1, T_2) = EMD(P(X), Q(X))$ where the underlying EMD measure uses the distance between sub-trees of height $h-1$ recursively.*

We define a δ-near neighbor T_2 of a query tree T_1 such that $d(T_1, T_2) < \delta$.

Definition 8 (similarity). *For trees T_1 and T_2 of height h, the similarity between them is defined as $sim(T_1, T_2) = 1 - d(T_1, T_2)$.*

Given two bags $\tilde{A}$ and $\tilde{B}$ with total weight 1 each, the size of their intersection $|\tilde{A} \cap \tilde{B}|$ is equal to $1 - d(a_i, b_j)$. The size of the bag union $|\tilde{A} \cup \tilde{B}|$ is equal to $1 + d(a_i, b_j)$. We note that for complete unweighted trees of uniform degree, the distance measure $d(.,.) \in [0, 1]$ is indeed the fraction of unmatched leaf labels under the best possible super-imposition. For two unweighted trees of height 1, with the same number of leaves, the EMD measure is the same as the fraction of unmatched leaf labels. This is because for such trees, the best fractional matching becomes an integral matching. The same reasoning applies recursively for larger heights for two complete trees of uniform degree.

Since the object of interest in this study are trees, our goal is to study tree sketching algorithms in the LSH framework. We will refer to such a sketch of tree T as *tree-hash* $TH(T)$. The framework studies the probability of the sketches matching under a gap in the distance between the trees. Using theorem 9.2.1, this framework can be used to find the δ-near neighbor of query tree. If there is a unique δ-near neighbor and all other trees in the database have similarity of at most ϵ with the query tree, then we will show that the query time of our algorithm is $N^{O(\delta/\log(1/\epsilon))}$, where N is the number of trees in the database. Formally, we will lower bound the probability p_1 of the sketches matching when the $d(T_1, T_2) < \delta$ and upper bound the probability p_2 when the $d(T_1, T_2) > 1 - \epsilon$ (correspondingly, $sim(T_1, T_2) \leq \epsilon$). Given two isomorphic trees T_1 and T_2, let s_{ij} denote $sim(T_{1i}, T_{2j})$. We have

Lemma 9.3.1. $\sum_i s_{ij} \leq 1$ *and* $\sum_j s_{ij} \leq 1$.

Proof. We use induction where the base case is obvious. The inductive step follows from the recursive definition of similarity. Consider the optimal super-imposition of the trees which result in the similarity measure. Since the similarity is defined in terms of the minimum weight of the matching, we arrange the subtrees according to the minimum weight permutation so that corresponding positions are matched. Since, the lemma holds inductively for each position, it holds for the sum of average similarity over all positions. □

9.4 Sketching algorithms for two-level trees

One-level trees are same as sets and therefore, sketching such trees are equivalent to sketching a set. We note that two-level trees are good abstractions for recursing the similarity computations from a set to set of sets. Therefore, they present a good starting point for experimenting with different approaches. In the following section, we contrast different algorithms starting from naive extensions of sketching algorithms on sets to algorithms on set of sets. We generalize many of the concepts introduced for two-level trees to multi-level trees in Section 9.5. We show how algorithms for sketching sets can be extended recursively to such trees. We start with a naive approach and show that it does not admit accurate similarity computations. We then show an approach that does produce good sketches resulting in effective similarity computations.

9.4.1 Propagating one min-hash does not work

A naive recursive extension is to compute the min-hash for each sub-tree and then compute the min-hash of min-hashes ar level one. Furthermore, we use a different min-hash permutation at each level. We will show that this method does not work as two completely different trees can result in the same min-hash with high probability.

Let π_1 and π_2 denote the min-hash permutation used on the leaves and level one nodes respectively. The following two propositions show that a random assignment of a given set of labels results in very different trees, but with a same tree-hash with significant probability. For this we will consider two-level trees that are unweighted and have degree n with n^2 leaves.

Proposition 1. *For given permutations π_1 and π_2 on the two levels of the tree and a set of n^2 labels, if the labels are randomly assigned without replacement to the n^2 leaves, there is some fixed label among these so that with at least constant probability $\Omega(1)$, the min-hash of the tree will result in that label.*

Proof. Let $l_1, l_2, \ldots, l_{n^2}$ denote the labels according to the min-hash permutation order π_1. We will show that with constant probability the tree-hash will be equal to $MH_{\pi_2}(l_1, l_2, \ldots, l_{n^2})$. Essentially, we will show that the set of min-hashes at level one is close to the set $H = l_1, l_2, \ldots, l_{n^2}$. Precisely, if A is the set of min-hashes at level one, then $E[|A \cap H|] \geq n/2$. This is because the probability of l_1 being present in H is 1; l_2 is present in H if l_1 is not in the same subtree which happens with probability at least $1 - 1/n$. Similarly, l_i ($i \leq n$) is present in H with probability at least $1 - (i-1)/n$. By linearity of expectations, the $E[|A \cap H|] \geq 1 + (1 - 1/n) + \ldots + 0 \geq n/2$.

From set similarity, it follows that $\mathbf{Pr}[MH_{\pi_2}(A) = MH_{\pi_2}(H)] = |A \cap H|/|A \cup H| > \frac{n/2}{2n} = 1/4$. □

Proposition 2. *Given two different random assignments (without replacement) of a given set of n^2 labels, then w.h.p., the distance between the resulting trees is $1 - o(1)$.*

Proof. Consider one set of leaves in a subtree. The expected number of leaves from this set present in any given subtree of the other tree is 1. By Chernoff bounds, w.h.p, the overlap between any two subtrees across the trees is $O(\log n)$. So, even under the best super-imposition, the distance is $1 - \frac{O(\log n)}{n}$. □

9.4.2 Propagating multiple min-hashes at each level

One method that uses multiple min-hashes is one in which we propagate a single min-hash from all leaves in a subtree to the parent and then the root node considers c permutations of the min-hashes computed at level one and concatenates them to get a tree-hash. However, this approach suffers from the same weakness of the naive approach wherein two different trees obtained from the same set of labels have a high matching probability of $(1/4)^c$ which can be derived as follows. If the random variable X denotes $|A \cap H|/|A \cup H|$, then the probability of a resulting fixed tree-hash is $E[X^c] \geq (E[X])^c > 1/4^c$.

We consider the following modification to the naive algorithm wherein we compute more than one min-hash at all nodes in the tree. Specifically, each level one node computes c min-hashes of the set of leaves in its subtree. The compound min-hash of the tree rooted at the level one node is computed by concatenating all the c min-hashes. Then the root node computes one min-hash from the compound min-hashes at level one nodes as shown in Algorithm 1. We note that we keep the number of permutations c constant and can hence be treated as an implicit parameter in all our analysis.

Algorithm 1 TH(T)

$\Pi \leftarrow \{\pi_{1,1}, \pi_{2,1}, \ldots, \pi_{c,1}\}$ be the permutations used by level one nodes
$\pi_{r,1}$ be the permutation used by the root node; $n \leftarrow$ degree of T
for all subtree T_i **do**
  $L_i \leftarrow leafs(T_i)$
  $CMH(T_i) \leftarrow MH_{\pi_{1,1}}(L_i) \bullet MH_{\pi_{2,1}}(L_i) \bullet \ldots \bullet MH_{\pi_{c,1}}(L_i)$
end for
return $MH_{\pi_{r,1}}\{CMH(T_1), CMH(T_2), \ldots, CMH(T_n)\}$

We now present the bounds on the probability of sketches matching for this case.

Lower bound p_1 on the matching probability

We now prove a lower bound on $\mathbf{Pr}[TH(T_1) = TH(T_2)]$ given $d(T_1, T_2) \leq \delta$. Here $T_1 = \{(a_1, \alpha_1), (a_2, \alpha_2), \ldots, (a_n, \alpha_n)\}$ where a_i is a one-level subtree with weight α_i. T_2 is defined similarly. Using the definition of $EMD(T_1, T_2)$ from above, it follows that there are values there is a fractional matching f_{ij} so that the distance between T_1 and T_2 is equal to $\sum_{ij} f_{ij} d(a_i, b_j) \le \delta$. Using this bound on the distance, we need to lower bound the probability that the two sketches match.

For a one-level subtree a_i from T_1 and b_j from T_2, the probability that a min-hash computation will result in the same hash value for the two trees is $|a_i \cap b_j|/|a_i \cup b_j| = (1 - d(a_i, b_j))/(1 + d(a_i, b_j)) \ge (1 - d(a_i, b_j))/2$. So the probability that all c min-hashes match for the two subtrees is $\ge 2^{-c}(1 - d(a_i, b_j))^c$. Let A and B denote the weighted sets of compound min-hashes in the tree-hash computation of T_1 and T_2 respectively. Then the probability that the final sketches match is $E[\frac{|A\cap B|}{|A\cup B|}] \ge E[|A \cap B|/2] \ge 1/2 \sum_{ij} f_{ij} 2^{-c}(1 - d(a_i, b_j))^c = 2^{c+1} \sum_{ij} f_{ij}(1 - d(a_i, b_j))^c$. By concavity of $(1-x)^c$ and Jensen's inequality, this is at least $2^{-(c+1)}(1 - \sum_{ij} f_{ij} d(a_i, b_j))^c \ge 2^{-(c+1)}(1 - \delta)^c$. Therefore the probability that the sketches match is $2^{-(c+1)}(1 - \delta)^c$.

Computing the upper bound is dependent on whether we consider the overlap between *every* pair of subtrees to bounded by ϵ or we use the more general case wherein the average overlap is bounded by ϵ. We consider both the cases next.

Upper bound p_2 on the matching probability

The upper bound is on $\mathbf{Pr}[TH(T_1) = TH(T_2)]$ given $d(T_1, T_2) > 1 - \epsilon$. Let d_{ij} denote the distance $d(a_i, b_j)$. Then for all f_{ij} satisfying $\sum_j f_{ij} \le \alpha_i$ and $\sum_i f_{ij} \le \beta_j$, we have $\sum_{ij} f_{ij} d_{ij} \ge 1 - \epsilon$. Let A and B denote the weighted sets of level one min-hashes in the tree-hash computation of T_1 and T_2 respectively. Let $s_{ij} = 1 - d_{ij}$. Then, $E[|A \cap B|] \le \sum_{ij} \min(\alpha_i, \beta_j)\, \mathbf{Pr}[CMH(A_i) = CMH(B_j)] = \sum_{ij} \min(\alpha_i, \beta_j)(s_{ij}/(2 - s_{ij}))^c \le \sum_{ij} \theta_{ij} s_{ij}^c$, where $\theta_{ij} = \min(\alpha_i, \beta_j)$. We have $p_2 = E[\frac{|A\cap B|}{|A\cup B|}] \le E[|A \cap B|] \le \sum_{ij} \theta_{ij} s_{ij}^c$.

Special case - To simplify we assume that the similarity s_{ij} between *every* pair of subtrees is bounded by ϵ and root nodes have degree n.

We know that s_{ij} is at most ϵ and $\sum_i s_{ij} \le 1$ and $\sum_j s_{ij} \le 1$. The sum $\sum_{ij} s_{ij}^c$ is maximized when each s_{ij} is either ϵ or 0. Exactly, n/ϵ will be non-zero giving a maximum value of $\frac{n}{\epsilon}\epsilon^c$ for the sum. So the probability is bounded above by ϵ^{c-1}. This shows that for $c = 1$, the algorithm works poorly to distinguish between very different trees. In fact, as an example, it can be shown that if we take two trees with the same set of leaves appearing in random order, for $c = 1$ they will result in the same sketch with high probability.

Next, we compute the upper bound p_2 in the general case.

General case - We eliminate the assumption in the special case.

In this case, all that we are given is that the distance between the two trees is at least $1-\epsilon$. By our definition of tree distance, this means for all fractional matchings $\sum_{ij} f_{ij} s_{i\pi(i)} \leq \epsilon$. Thus, the maximum weight fractional matching between the two sets of subtrees is at most ϵ where the weight is set to be the similarity measure. Now, we need to bound $\sum_{ij} \theta_{ij} s_{ij}^c$.

To prove the existence of this bound, we consider each level-one subtree in the trees T_1 and T_2 to represent a node in a bi-bipartite graph and the weights on the edges between the nodes to denote the similarity between the sub-trees. This naturally leads to an optimization problem on a complete bipartite graph with a given maximum weight matching formulation.

Problem 9.4.1. *Given a complete bipartite graph with edge weights w_{ij} and weights α_i, β_j on the left and right vertices, what is the maximum value of $\sum_{ij} \theta_{ij} w_{ij}^c$ where $\theta_{ij} = \min(\alpha_i, \beta_j)$ given*

- *the weighted degree of any node in G is bounded by 1, i.e., $\sum_i w_{ij} \leq 1$, and $\sum_j w_{ij} \leq 1$,*
- *$\sum_i \alpha_i = 1$ and $\sum_j \beta_j = 1$,*
- *the maximum weight fractional matching is at most ϵ, that is if $\sum_j f_{ij} \leq \alpha_i$ and $\sum_i f_{ij} \leq \beta_j$ then $\sum_{ij} f_{ij} w_{ij} \leq \epsilon$.*

The following lemma shows $p_2 \leq O(\epsilon \log(1/\epsilon))$.

Lemma 9.4.1. *Given a complete weighted bipartite graph $G = (V, E)$ on m left vertices and n right vertices with weight $w_{ij} \in [0, 1]$ on each edge and weights α_i, β_j on the left and right vertices such that $\sum_i \alpha_i = 1$ and $\sum_j \beta_j = 1$. Given that the maximum fractional weight matching of G is bounded by ϵ where $\epsilon > 0$, then for any $c \geq 2$, $\sum_{ij} \theta_{ij} w_{ij}^c \leq O(\epsilon \log(1/\epsilon))$.*

Proof. We round up all weights to the nearest power of $1/2$ and group the edges by their weights. We consider the bound as a sum of two parts. First, from those edges with weights less than ϵ and second, those greater than ϵ. Let η_{ik} denote the number of edge weights in the range $[\gamma_k, \frac{\gamma_k}{2})$ coming out of node i.

Part 1 - The range of interest is $[0, \epsilon]$. We round up weights to the nearest powers of $1/2$, i.e., $\epsilon/2^k$. If we denote $\gamma_k = \epsilon/2^k$, then the contribution of these weights to $\sum_{ij} \theta_{ij} w_{ij}^c$

is $\sum_i \theta_{ij} \frac{\eta_{ik}}{\gamma_k} \gamma_k^c = \sum_i \theta_{ij} \eta_{ik} \gamma_k^{c-1}$ giving us $\sum_{ij} \theta_{ij} w_{ij}^c \leq \sum_i \sum_k \alpha_i \eta_{ik} \gamma_k^{c-1}$. Clearly, $\sum_{ij} w_{ij}^c \leq \epsilon^{c-1} \sum_i \sum_k \eta_{ik}$. Furthermore, since all the weights are bounded by ϵ, $\sum_i \sum_k \alpha_i \eta_{ik} \leq 1$. This gives us $\sum_{ij} \theta_{ij} w_{ij}^c \leq \epsilon^{c-1}$.

Part 2 - Here the weights are in the range $[\epsilon, 1]$. Let γ denote a particular weight $\geq \epsilon$ that is a power of 2. Note that for the k-th bucket, $\gamma_k = (1/2)^k$. We have,

$$\sum_{ij} \theta_{ij} w_{ij}^c \leq \sum_{k=0}^{\log(1/\epsilon)} \sum_i \theta_{ij} \eta_{ik} \gamma_k^{c-1}. \tag{9.1}$$

We will compute the contribution to the sum from one value of k with edge weights $\gamma_k = 1/2^k$. We make use of the fact that for a graph with a maximum degree of d, there exists a decomposition into at most d matchings (the matchings need not be perfect). Since the total edge weight at any given node is at most 1, the maximum degree d is $\frac{1}{\gamma_k}$. Furthermore, the number of edges with weight γ_k, η_{ik}, is decomposed into d components – ω_{ikl}, $1 \leq j \leq d$ – one for each matching. The weight of each decomposed matching is bounded by ϵ. To prove this, let us consider any one matching. Let $f_{ij} = \theta_{ij}$ if (i, j) is in the matching and 0 otherwise. These f_{ij} satisfy the constraints in our similarity definition. Using the bound of ϵ on each matching and adding this bound over all $1/\gamma_k$ matchings gives us $\sum_i \theta_{ij} \eta_{ik} \leq \frac{\epsilon}{\gamma_k}$. Substituting this value in Equation 9.1 and using $\gamma_k^{c-1} < \gamma_k$, we have for $c \geq 2$,

$$\sum_{ij} \theta_{ij} w_{ij}^c \leq \epsilon \log(1/\epsilon).$$

Adding the bounds on the two parts completes the proof. □

Observe that choosing any value of $c > 2$ does not improve the bound on p_2. Therefore $c = 2$ is good choice for our algorithm. We summarize the result with the following theorem.

Theorem 9.4.2. *Given two two-level isomorphic trees T_1 and T_2 with similarity ϵ, the probability their sketches match is at most $\epsilon \log(1/\epsilon)$.*

9.4.3 Connections to Locality-Sensitive Hashing

Given that $p_1 = 2^{-(c+1)}e^{-c\log(1-\delta)}$ and $p_2 = \epsilon log(\frac{1}{\epsilon})$, for $c = 2$, the above probabilities give $\rho = O(\frac{\delta}{\log(1/\epsilon)})$. From theorem 9.2.1, we obtain the following result.

Theorem 9.4.3. *Given a database of N trees in which all trees are* $(1-\epsilon)$*-far, then there is a data structure of size* $N^{1+\delta/\log(1/\epsilon)}$ *to find the* δ*-near neighbor in time* $N^{\delta/\log(1/\epsilon)}$*. Alternately, such a data structure can be used to find an* $(1-\epsilon)$*-near neighbor assuming the nearest neighbor is within distance* δ*.*

9.5 Sketching algorithms for multilevel trees

We continue the analysis for multilevel weighted trees of arbirary shape and present a generalization to compute the bounds for the matching probabilities between such trees.

We now generalize the algorithm for computing the tree-hash, $TH(T^h)$, of a multilevel tree, T^h, of level h. If $h = 1$, then tree-hash is equal to the min-hash of the leaf set. For $h > 1$, we consider c random instances of the tree-hash function with independent coin tosses. We use these functions to compute the compound tree-hash for every subtree at level $h - 1$. Thus, a compound tree-hash itself is a concatenation of all the c tree-hash values obtained using the c functions. The tree-hash for a subtree at level h is computed from the n compound tree-hashes at level $h - 1$. In practice, to avoid min-hashes that are concatenation of many labels, we can simply use a hash value of the concatenation instead. However, for the analysis, we will conceptually use the concatenated value.

Remark 9.5.1. *In the current definition of Algorithm 2, the number of min-hash computations at the leaf level becomes* 2^h *for* $c = 2$ *because of the recursion. This can be avoided by using the same set of c min-hashes for each level resulting in* $2h$ *min-hash computations for* $c = 2$*. Under this assumption, the analysis of the lower bound can be easily extended, while the analysis of the upper bound remains an open question.*

Using the above recursive formulation for computing the tree-hash of a multilevel tree, we compute the probability bounds $p_{1,h}$ and $p_{2,h}$, where $p_{.,h}$ denotes the matching probability for two subtrees between T_1 and T_2 at level h. We show inductively that the bounds hold for all levels in the tree.

Algorithm 2 TH(T^h)

$\pi^r \leftarrow$ permutation used by the root node; $h \leftarrow height(T^h)$; $n \leftarrow degree(root(T^h))$
if $h > 1$ **then**
  use c random instances of tree-hash functions, $TH_1, TH_2, \ldots, TH_c$ with independent coin tosses
  for subtree T_i^{h-1} at level $h-1$ **do**
    $CTH(T_i^{h-1}) \leftarrow TH_1(T_i^{h-1}) \bullet TH_2(T_i^{h-1}) \bullet \ldots \bullet TH_c(T_i^{h-1})$
  end for
  return $MH_{\pi^r}\{CTH(T_1^{h-1}), CTH(T_2^{h-1}), \ldots, CTH(T_n^{h-1})\}$
else if $h = 1$ **then**
  $L \leftarrow leafs(T^h)$
  return $MH_{\pi^r}(L)$
end if

Theorem 9.5.1 (Lower bound p_1). *Given two isomorphic trees T_1 and T_2 of level h with distance δ and the probability that their tree-hashes match at level h is $p_{1,h}$, then for $h \geq 1$ and $c = 2$,*

$$p_{1,h} \geq \frac{(1-\delta)^{2^{h-1}}}{2^{2^h-1}}$$

Proof. We want to lower bound $\mathbf{Pr}[TH(T_1) = TH(T_2)]$ given the distance bound $d(T_1, T_2) \leq \delta$.

We present a proof similar to the one presented to compute the lower bound in Section 9.4. Let $p_{1,h}$ denote the matching probability for any subtree at level h. Therefore, the probability that the compound tree-hash for a subtree at level h is the same is $p_{1,h}^c$. Let A and B be the weighted sets of compound min-hashes of the subtrees in T_1 and T_2 respectively. We have $p_{1,2} = 2^{-(c+1)}(1-\delta)^c$ from the analysis for the two-level trees. Given $p_{1,1}$ is concave, we can inductively show that $p_{1,l}$, $l > i$ is also concave. This gives us a matching probability for the compound tree-hashes of subtrees at level $h+1$ equal to $p_{1,h+1} = E[|A \cap B|/|A \cup B|] \geq \sum_{ij} f_{ij} p_{1,h}^c/2 = p_{1,h}^c/2$, where the last inequality follows from the concavity of $p_{1,h}$. Given this recurrence, for $h > 0$ and $c = 2$, we have

$$p_{1,h} \geq \frac{p_{1,h-1}^2}{2} = \frac{(1-\delta)^{2^{h-1}}}{2^{2^h-1}}$$

□

Theorem 9.5.2 (Upper bound p_2). *Given two isomorphic trees T_1 and T_2 of level h with similarity ϵ and the probability that their tree-hashes match is $p_{2,h}$, for $h \geq 2$ and $c = 2$, there exists some constants c_1 and c_2 for which*

$$p_{2,h} \leq c_1 \epsilon \log(\frac{1}{\epsilon}) c_2^{3^h} (\log(\frac{n}{\epsilon}))^{2^{h-1}-2}$$

Proof. Let A and B denote the set of compound tree-hashes for the child subtrees of T_1 and T_2 respectively. The probability that the sketches match is $E[|A \cap B|/|A \cup B|] \leq E[|A \cap B|]$. Let $p_{ij} = \mathbf{Pr}[TH(T_i^h) = TH(T_j^h)]$, i.e., probability that the tree-hashes of two given subtrees match. Now, given p_{ij}, we want to bound $p_{2,h} = \mathbf{Pr}[TH(T_1^h) = TH(T_2^h)]$ recursively.

Similar to two-level trees, $p_{2,h} = E[|A \cap B|/|A \cup B|] \leq E[|A \cap B|]$. Thus, we have

$$\begin{aligned} E[|A \cap B|] &\leq \sum_{ij} \theta_{ij} \mathbf{Pr}[CTH(A_i) = CTH(B_j)] = \sum_{ij} \theta_{ij} (\mathbf{Pr}[TH(A_i) = TH(B_j)])^2 \leq \sum_{ij} \theta_{ij} p_{ij}^2 \\ &\leq \sum_{ij} \theta_{ij} (c_1 s_{ij} \log(\frac{1}{s_{ij}}) c_2^{3^h} (\log(\frac{n}{s_{ij}}))^{2^{h-1}-2})^2 \quad \text{by induction.} \end{aligned}$$

By concavity and Jensen's inequality, we have,

$$E[|A \cap B|] \leq \sum_{ij} \theta_{ij} c_1^2 R \log^2(\frac{1}{R}) c_2^{3^h 2} (\log^2(\frac{n}{R}))^{2^h-4}$$

where, $R = \sum_{ij} \theta_{ij} s_{ij}^2 / \sum_{ij} \theta_{ij}$. Viewing s_{ij} as weights in a weighted bipartite graph which satisfy conditions of Lemma 9.4.1, we get

$$\sum_{ij} \theta_{ij} s_{ij}^2 \leq O(\epsilon \log(1/\epsilon)),$$

since ϵ is the normalized maximum weight matching. Using $m = \sum_{ij} \theta_{ij} < n$ gives us

$$\begin{aligned} p_{2,h+1} &\leq c_1^2 m \frac{\epsilon \log(1/\epsilon)}{m} \log^2(\frac{m}{\epsilon \log(1/\epsilon)}) c_2^{3^h 2} (\log(\frac{nm}{\epsilon \log(1/\epsilon)}))^{2^h-4} \\ &\leq c_1^2 \epsilon \log(1/\epsilon) \log^2(\frac{n}{\epsilon \log(1/\epsilon)}) (4c_2)^{3^h 2} \log(\frac{n}{\epsilon \log(1/\epsilon)}))^{2^h-4} \\ &= c_1^2 \epsilon \log(1/\epsilon) 2^{2^h-4} c_2^{3^h 2} \log(\frac{n}{\epsilon \log(1/\epsilon)}))^{2^h-2} \end{aligned}$$

It follows that $c_1 2^{2^h-4} c_2^{3^h 2} \leq c_2^{3^{h+1} 2}$ for small h and $c_2 > c_1$ and $c_2 > 4$. For the specified values of h, c_2 and c_1, we have,

$$p_{2,h+1} \leq c_1 \epsilon \log(1/\epsilon) c_2^{3^h} (\log(n/\epsilon))^{2^h-2}$$

Thus, we have shown inductively that the bound for $p_{2,h+1}$ holds given $p_{2,h}$. □

Given the upper and lower bounds for matching probabilities, we have for trees of small height, $\rho = \frac{\log 1/p_{1,h}}{\log 1/p_{2,h}}$. Clearly for small values of h, $1/p_{2,h}$ is dominated by $O(\epsilon \log(1/\epsilon))$. This gives us a value of $\rho = \frac{-2^{h-1}\log(1-\delta)}{\log(\frac{1}{\epsilon \log(1/\epsilon)})} \approx \frac{2^{h-1}\delta}{\log(1/\epsilon)}$ which gives us the following result as a consequence of theorem 9.2.1.

Theorem 9.5.3. *Given a database of N trees in which all trees are $(1-\epsilon)$-far, then there is a data structure of size $N^{1+O(\delta/\log(1/\epsilon))}$ to find the δ-near neighbor in time $N^{O(\delta/\log(1/\epsilon))}$.*

9.6 Conclusions

In this chapter, we analyzed sketching algorithms to compute similarities between trees. Specifically, we studied our algorithms using the framework of locality-sensitive hashing introduced in [58]. This allowed us to find an approximate nearest neighbor among trees in time $N^{O(\delta/\log(1/\epsilon))}$, where N is the size of the database and δ and $1-\epsilon$ are well separated upper and lower bounds of distance in our algorithms.

Chapter 10

Conclusion

In this thesis we have seen a collection of results on searching based on hashing and sketching. We saw how variants of balls and bins processes can be used to derive space efficient methods for maintaining hash tables. By viewing items and buckets as balls and bins respectively and performing moves during inserts we can maintain a space utilization of about 83%. We also saw some new results on balls-and-bins processes when bins were chosen according to a given underlying graph and studied the distribution of balls into bins in terms of graph properties. Balls-and-bins processes also led to a more compact version of Bloom filters for approximate set operations.

We also studied the similarity search problem by viewing it as a nearest neighbor search in a high dimensional space. Since this problem suffers from the curse of dimensionality, our objective was to find an approximate nearest neighbor. We saw a variant of locality sensitive hashing that uses linear space by searching for points in the neighborhood of the query point. This method was also applicable to the kd-tree data structure and led to a much improved rate of success in finding an approximate nearest neighbor. We also saw tight lower bounds on the performance of the locality sensitive hashing technique for nearest neighbor search. In the area of streaming algorithms, we saw tight lower bounds on the space required for finding frequent elements in a stream. Finally, we saw a sketching algorithm for similarity search on hierarchical data represented as trees.

10.1 Future Work

There are several possible future directions and open problems. For exact search using hashing, we may ask what is the best possible space utilization achievable while requiring only a few memory accesses for search; there are several possible trade-offs here – is it possible do have a high space utilization while performing close to one memory access for search; what if we allow the number of memory accesses per lookup to vary but insist that the average number of lookups is close to one, or allow additional accesses to a much smaller segment of memory that could fit in a cache?

In the variant of balls and bins processes based on an underlying graph, we only study the case where two bin choices are made to per ball but leave the case of larger number of bin choices open. An interesting open question is whether the results continue to extend for larger number of bin choices where the d bin choices are constrained by an underlying hypergraph instead of a graph.

For similarity search using nearest neighbor, although we show tight lower bounds on what can be achieved using the locality sensitive hashing, a wide open question remains – can we do better by using other methods? Are there faster algorithms for nearest neighbor search in other important metrics such as the edit distance metric, or earth mover distance, or L_p distance metrics? In the area of streaming algorithms for finding frequent elements, can we do better by exploiting the arrival turn of the elements in the stream; does it make a difference if stream elements arrive in bursts or in random order? As for sketching on hierarchical data, are there good sketching algorithms more complicated structures such as labeled trees, DAGs, or graphs?

In conclusion, it seems that the area of sketching and hashing for similarity search is a relatively young and fertile ground for research with huge potential impact.

Bibliography

[1] N. Augsten, M. Bohlen, and J. Gamper. Approximate matching of hierarchical data using *pq*-grams. In *Proc. of the 31st VLDB Conference*, pages 301–312, 2005.

[2] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. SIAM Journal on Computing, 29:180-200, 1999. *A preliminary version of this paper appeared in Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing, 1994.*

[3] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. Rasmussen. Parallel randomized load balancing. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 238-247, May 1995.

[4] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems*, pages 10–20, 1999.

[5] A. Andoni and P. Indyk, Near-Optimal Hashing Algorithms for Near Neighbor Problem in High Dimensions. In *Proceedings of the Symposium on Foundations of Computer Science* , 2006.

[6] P.K. Agarwal and J. Matousek. Ray shooting and parametric search. In *Proc. of 24th STOC*, pp. 517–526, 1992.

[7] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.

[8] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. PODS*, 2002.

[10] A. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of SEQUENCES SEQS: Sequences '91*, 1998.

[11] A.Z. Broder, M. Charikar, A.M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[12] P. Berenbrink, A. Czumaj, A. Steger, and B. Vöcking. Balanced allocations: the heavily loaded case. In *Proc. of the 32nd Annual ACM STOC*, pp. 745–754, 2000.

[13] W. Beckner. Inequalities in Fourier analysis. *Ann. of Math. (2)*, 102(1):159–182, 1975.

[14] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond Bloom filters: From approximate membership checks to approximate state machines. In Proc. of *SIGCOMM*, 2006.

[15] J. L. Bentley, J. H. Friedman, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, 1977.

[16] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485-509, 2004.

[17] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An easy $\omega(n)$ lower bound for a gap Hamming distance problem. *Manuscript*, 2004.

[18] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004.

[19] B. Bloom. Space/time tradeoffs in in hash coding with allowable errors. *Communications of the ACM*, 13(7):422-426, 1970.

[20] A. Broder and M. Mitzenmacher. Using multiple hash functions to improve IP Lookups. In *Proceedings of IEEE INFOCOM*, pp. 1454-1463, 2001.

[21] A. Bonami. Étude des coefficients de Fourier des fonctions de $L^p(G)$. *Ann. Inst. Fourier (Grenoble)*, 20(fasc. 2):335–402 (1971), 1970.

[22] A. Borodin, R. Ostrovsky, and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 312–321, 1999.

[23] M. Sharma and J. Byers. Scalable Coordination Techniques for Distributed Network Monitoring. *6th International Workshop on Passive and Active Network Measurement (PAM)*, pp. 349-352, 2005.

[24] W. Chen. New algorithms for ordered tree-to-tree correction problem. *Journal of Algorithms*, 40(2):135–158, August 2001.

[25] M. Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. 34th Annual ACM Symposium on Theory of Computing*, pages 380–388, 2002.

[26] K. L. Clarkson. Nearest neighbor queries in metric spaces. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing* (May 1997), pp. 609–617.

[27] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming*, pages 693–703, 2002.

[28] R. Cole, A. Frieze, B.M. Maggs, M. Mitzenmacher, A. W. Richa, R. K. Sitaraman, and E. Upfal. On balls and bins with deletions. In *Second International Workshop on Randomization and Approximation Techniques in Computer Science* (RANDOM), number 1518 in Lecture Notes in Computer Science, pages 145-158. Springer-Verlag, October 1998.

[29] B. Chor and O. Goldreich. Unbiased bits from sources of weak randomness and probabilistic communication complexity. *SIAM Journal on Computing*, 17(2):230–261, 1988.

[30] T. Cover, P. Hart. Nearest neighbor pattern classification.*IEEE Trans. Information Theory* IT-13(1967), pp. 21-27.

[31] A. Chakrabarti, S. Khot, and X. Sun. Near-optimal lower bounds on the multiparty communication complexity of set-disjointness. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity*, pages 107–117, 2003.

[32] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[33] R. Cole, B. M. Maggs, F. Meyer auf der Heide, M. Mitzenmacher, A. W. Richa, K. Schröder, R. K. Sitaraman, and B. Vöcking. Randomized protocols for low-congestion circuit routing in multistage interconnection networks. In *Proc. of the 30th ACM Symp. on Theory of Computing* (STOC), pages 378-388, 1998.

[34] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Shared memory simulations with triple-logarithmic delay. *Lecture Notes in Computer Science*, 979:46-59, 1995.

[35] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 493–504, 1996.

[36] A. Czumaj, C. Riley, and C. Scheideler. Perfectly Balanced Allocation In *Proceedings of the 7th International Workshop on Randomization and Approximation Techniques in Computer Science* (RANDOM'03), pages 240 - 251.

[37] A. Czumaj and V. Stemann. Randomized allocation processes. Random Struct. Algorithms 18(4): 297-331 (2001). A preliminary version appeared in *the proceedings of the Thirty-Eighth Annual Symposium on Foundations of Computer Science*, pages 194-203, October 1997.

[38] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., 1991.

[39] S. Deerwester, S. T. Dumais, T.K. Landauer, G.W. Furnas, and R.A. Harshman. Indexing by latent semantic analysis, *Journal of the Society for Information Science*, 41(6), 391-407, 1990.

[40] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using Bloom filters. *Proceedings of the ACM SIGCOMM 2003*, pp. 201-212.

[41] Danny Dolev, Yuval Harari, and Michael Parnas. Finding the neighborhood of a query in a dictionary. In *Proc. 2nd Israel Symposium on Theory of Computing and Systems*, pages 33–42, 1993.

[42] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04: Proceedings of the Twentieth Annual Symposium on Computational Geometry (Brooklyn, NY, 2004)*, pages 253–262. ACM Press, New York, NY, 2004.

[43] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738-761, 1994.

[44] D. Dobkin and R. Lipton. Multidimensional search problems. *SIAM J. Comput.*, 5:181-186, 1976.

[45] L. Devroye and P. Morin. Cuckoo hashing: further analysis. *Information Processing Letters*, vol. 86, pp. 215-219, 2003

[46] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming* (ICALP '90), volume 443 of Lecture Notes in Computer Science, pages 6-19. Springer-Verlag, Berlin, 1990.

[47] M. Dietzfelbinger and F. Meyer auf der Heide. Simple Efficient Shared Memory Simulations.*Proc. of the 5th SPAA* (1993); pp. 110-119.

[48] M. Dietzfelbinger and P. Woelfel. Almost random graphs with simple hash functions. In the *proceedings of STOC 2003*, pp. 629-638

[49] M. Dietzfelbinger and C. Weidling. Balanced Allocation and Dictionaries with Tightly Packed Constant Size Bins. In the *proceedings of ICALP 2005*.

[50] L. Devroye and T.J. Wagner. Nearest neighbor methods in discrimination. Handbook of Statistics, volume 2, P.R. Krishnaiah and L.N. Kanal, editors, North-Holland, 1982.

[51] R. Fagin, Fuzzy Queries in Multimedia Database Systems, *Proc. ACM Symposium on Principles of Database Systems* (1998), pp. 1–10

[52] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Trans. on Networking*, 8(3):281-293, 2000.

[53] E. Friedgut and G. Kalai. Every monotone graph property has a sharp threshold. *Proceedings of the American Mathematical Society* 124 (1996), 2993-3002.

[54] M. L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with O(1) worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538-544, 1984.

[55] D. Fotakis, R. Pagh, P. Sanders and P. Spirakis. Space Efficient Hash Tables With Worst Case Constant Access Time. *20th Annual Symposium on Theoretical Aspects of Computer Science*, 2003.

[56] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. In *Proceedings of 22nd International Conference on Very Large Data Bases*, pages 307–317, 1996.

[57] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, P. Yanker. Query by image and video content: The QBIC system. *Computer 28* (1995) 23–32

[58] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of 25th International Conference on Very Large Data Bases, VLDB*, pages 518–529, 1999.

[59] P. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 331–342, 1998.

[60] P. Gibbons and Y. Matias. Synposis data structures for massive data sets. In *Proceedings of 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 909–910, 1999.

[61] G.H. Gonnet. Expected length of the longest probe sequence in hash code searching. *J. ACM*, 28(2):289-304, 1981.

[62] S. Gollapudi and R. Panigrahy. Exploiting asymmetry in hierarchical topic extraction. *Manuscript*, 2006.

[63] S. Har-Peled. A replacement for voronoi diagrams of near linear size. *Proceedings of the Symposium on Foundations of Computer Science*, 2001.

[64] S. Har-Peled. A replacement for Voronoi diagrams of near linear size. In *42nd IEEE Symposium on Foundations of Computer Science (Las Vegas, NV, 2001)*, pages 94–103. IEEE Computer Soc., Los Alamitos, CA, 2001.

[65] T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *WebDB (Informal Proceedings)*, pages 129–134, 2000.

[66] P. Indyk. High-dimensional computational geometry. Dept. of Comput. Sci., Stanford Univ., 2001.

[67] P. Indyk, Approximate Nearest Neighbor under Frechet Distance via Product Metrics, *ACM Symposium on Computational Geometry*, 2002

[68] P. Indyk. Nearest neighbors in high-dimensional spaces. In Jacob E. Goodman and Joseph O'Rourke, editors, Handbook of Discrete and Computational Geometry, chapter 39. CRC Press, 2rd edition, 2004.

[69] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th Symposium on Theory of Computing*, 1998, pp. 604–613.

[70] P. Indyk, R. Motwani, P. Raghavan, and S. Vempala. Locality-preserving hashing in multidimensional spaces. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 618–625, 1997.

[71] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC '98: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (Dallas, TX, 1998)*, pages 604–613. ACM Press, New York, NY, 1998.

[72] P. Indyk and N. Thaper, Embedding Earth-Mover Distance into the Euclidean space, *manuscript*, 2001.

[73] P. Indyk and N. Thaper, Fast color image retrieval via embeddings. Workshop on Statistical and Computational Theories of Vision (at ICCV), 2003.

[74] P. Indyk and D. P. Woodruff. Tight lower bounds for the distinct elements problem. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science*, pages 282–289, 2003.

[75] P. Indyk and D. P. Woodruff. Optimal approximations of the frequency moments of data sets. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 202–208, 2005.

[76] T. S. Jayram, S. Khot, R. Kumar, and Y. Rabani. Cell-probe lower bounds for the partial match problem. In *Proc. 35th Annu. ACM Symp. Theory Comput.*, pages 667–672, 2003.

[77] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. In *Proc. Intl Conference on Combinatorial Pattern Matching*, pages 75–86, 1994.

[78] J. Kleinberg. Two algorithms for nearest-neighbor search in high dimension. *Proc. 29th Annu. ACM Sympos. Theory Comput.*, pp. 599–608. 1997.

[79] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[80] D. E. Knuth. The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 3rd ed. Reading, MA: Addison-Wesley, 1997.

[81] K. Kailing, H-P. Kriegel, S. Schonauer, and T. Seidl. Efficient similarity search for hierarchical data in large databases. In *Proc. 9th Intl Conference on Extending Database Technology*, pages 676–693, 2004.

[82] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proc. of 30th STOC*, pp. 614–623, 1998.

[83] K. Kenthapadi and R. Panigrahy. Balanced allocation on graphs. In *Proc. of the Seventeenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 434-443, 2006.

[84] V. F Kolchin and B. A. Sevast'yanov and V. P. Chistyakov. Random Allocations. V H Winston & Sons, 1978.

[85] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. In *IEEE Hot Interconnects 12*, 2003.

[86] N. Linial and O. Sasson, Non-Expansive Hashing, In *Proc. 28th STOC* (1996), pp. 509–517.

[87] S. Meiser. Point location in arrangements of hyperplanes. *Information and Computation*, 106(2):286–303, 1993.

[88] M. Mitzenmacher, The Power of Two Choices in Randomized Load Balancing, Ph.D. thesis, University of California, Berkeley, 1996.

[89] T. Margush and F. R. McMorris. Consensus *n*-trees. *Bulletin of Mathematical Biology*, 3:239–244, 1981.

[90] M. Mitzenmacher, A. Richa, and R. Sitaraman The Power of Two Random Choices: A Survey of Techniques and Results. Book chapter, in Handbook of Randomized Computing: volume 1, edited by P. Pardalos, S. Rajasekaran, and J. Rolim, pp. 255-312, 2001.

[91] F. Meyer auf der Heide, C. Scheideler and V. Stemann. Exploiting Storage Redundancy to Speed Up Randomized Shared Memory Simulations. *Theoretical Computer Science*, 162(2):245-281, 1996. Preliminary version in Proc. of the 12th STACS (1995); pp. 267-278.

[92] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[93] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers Inc., 2005.

[94] M. Mitzenmacher and B. Vöcking. The asymptotics of selecting the shortest of two, improved. In *Analytic Methods in Applied Probability: In Memory of Fridrikh Karpelevich*, edited by Y. Suhov, American Mathematical Society, 2003.

[95] R. Motwani, A. Naor, and R. Panigrahy. Lower Bounds on Locality Sensitive Hashing. In *Proceedings of the 22nd Annual ACM Symposium on Computational Geometry*, 2006

[96] A. Östlin and R. Pagh. Uniform Hashing in Constant Time and Linear Space. In the *proceedings of STOC 2003*, pp. 622-628.

[97] R. Ostrovsky and Y. Rabani. Low distortion embeddings for edit distance. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 218–224, 2005.

[98] A. Pagh, R. Pagh, and S. Srinivas Rao. An Optimal Bloom Filter Replacement. In *Proc. of the Sixteenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 823-829, 2005.

[99] R. Panigrahy. Efficient hashing with lookups in two memory accesses. In *Proc. of the Sixteenth Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 830-839, 2005.

[100] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm (Miami, FL, 2006)*, pages 1186–1195. ACM Press, New York, NY, 2006.

[101] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory Probab. Appl.*, 16:264–280, 1971.

[102] A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Tools for content-based manipulation of image databases. In *Proceedings of the SPIE Conference On Storage and Retrieval of Video and Image Databases* (February 1994), vol. 2185, pp. 34–47.

[103] R. Pagh and F. Rodler. Cuckoo Hashing. Journal of Algorithms 51 (2004), p. 122-144. A preliminary version appeared in *proceedings of the 9th Annual European Symposium on Algorithms.* pp. 121-133, 2001.

[104] B. Pittel, S. Spencer and N. Wormald. Sudden emergence of a giant k-core in a random graph. *J. Combin. Theory Ser.* B 67 (1996), no. 1, 111-151.

[105] C. J. van Rijsbergen. Information Retrieval. Butterworths, London, United Kingdom, 1990.

[106] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*, pp. 494–505, 2005.

[107] M. Raab and A. Steger. "Balls into bins" – a simple and fight analysis. In the *proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science*, LNCS volume 1518, pages 159-170, 1998.

[108] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

[109] G. Salton. Automatic Text Processing. Reading, MA: Addison-Wesley, 1989

[110] P. Sanders, S. Egner, Jan H. M. Korst. Fast concurrent access to parallel disks. Algorithmica, 35(1):21-55,2003. A Preliminary version appeared in *SODA 2000*: 849-858.

[111] S. Cohen and Y. Matias. Spectral Bloom Filters. *Proceedings of the 2003 ACM SIGMOD Conference*, pp. 241-252.

[112] B. Vöcking. How asymmetry helps load balancing. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. pp. 131-140, 1999.

[113] V.M. Zolotarev, One-dimensional Stable Distribution. *Translations of Mathematical Monographs* 65, American Mathematical Society, Providence, RI, 1986.

[114] K. Zhang. A constrained editing distance between unordered labeled trees. *Algorithmica*, 15:205–222, 1996.

[115] Li Zhang. On matching nodes between trees. Technical Report HPL-2003-67, HP Laboratories, Palo Alto, CA, April 2003.

[116] Y. Zhang, X. Lin, J. Xu, F. Korn, and W. Wang. Space-efficient relative error order sketch over data streams. In *Proceedings of the 22nd International Conference on Data Engineering*, page 51, 2006.

[117] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.