

Expander Based Dictionary Data Structures

by

Esben Rune Hansen, Peter Tiedemann, and Mette Berger

31st May 2005

Supervisors: Anna Östlin Pagh and Rasmus Pagh

Abstract/Summary

We consider dictionary data structures based on expander graphs. We show that any one probe scheme with the properties of the previous data structure from [OP02] is indeed space optimal. We then construct four different dictionary data structures for various models of parallel external memory. All of them allows lookups using a single parallel probe. In the following n denotes the number of keys in the dictionary, and u the universe of possible keys. Δ_{opt} denotes the space in bits required to store the n keys and their satellite data without any type of compression and $d = O(\log(u/n))$.

- A static dictionary data structure with error correcting codes using $O(\Delta_{opt})$ bits of space, and one requiring $O(nd \log d + \Delta_{opt})$ bits of space without using error correcting codes.
- A dynamic dictionary data structure for the parallel disk head model using $O(nd \log n + \Delta_{opt})$ bits of space, where updates take $O(1)$ I/O's amortized.
- A dynamic dictionary data structure for the parallel disk model, with the same space usage, and updates in $O(1)$ expected amortized I/O's.

For the data structure in [OP02] and the above dynamic data structures we furthermore discuss an alternative insertion algorithm based on a simple random walk, which we hope can be shown to give a good worst-case insertion time

We also consider two probe schemes where the first probe is randomized and the second is deterministic. Under these circumstances we obtain a static dictionary data structure using $O(nd \log d + \Delta_{opt})$ bits of space. We also give a dynamic variant using $O(nd \log n + \Delta_{opt})$ bits of space.

Finally for the case $u = poly(n)$ we give a construction of lossless bipartite expander graphs, using space $O(\sqrt[b]{n})$ for any constant b , and expanding for sets up to size n , having degree poly-logarithmic in u .

Problem statement

We examine dictionary datastructures that (expected or worst case) circumvent the traditional logarithmic lower bound on number of lookups for various computational models.

We will focus on the expander based dictionary for one probe lookups by Pagh and Östlin, working on the following two main issues:

- How to reduce the space use by relaxing the computational model and/or the number of probes allowed. Specifically we consider the parallel I/O model of computation.
- Proving that the original one probe based datastructure is optimal in terms of space use.

Contents

1	Introduction	6
1.1	Dictionaries and Expanders	6
1.2	Previous work on dictionaries	8
1.3	This Thesis	9
1.3.1	Thesis Overview	9
1.3.2	How we cite results and proofs	10
1.3.3	Definitions and naming standards	10
1.4	One Probe Search	11
1.4.1	Preliminaries	11
1.4.2	Static data structure	12
1.4.3	Dynamic updates	14
1.4.4	Time bounds	19
2	One probe lower bound	21
2.1	Schemes and lower bound models	21
2.2	Lower bound for the space usage of the original OPS	23
2.2.1	Tight lower bound for a guarded one probe scheme	23
2.2.2	Implied bounds on expanders	23
2.2.3	The central role of Property 4	24
2.3	Lower bound for unguarded schemes	24
2.3.1	Lower bound for an unguarded one probe scheme	24
2.3.2	Is the unguarded scheme really less restrictive?	26
2.4	Generality of the lower bound models	27
2.5	Conclusion	28
3	One Probe in the Parallel Disk Models	29
3.1	Introduction	29
3.1.1	Preliminaries	30
3.1.2	Existence of good expander graphs	31
3.2	Simple static PDM dictionary	34
3.3	A dynamic PDHM dictionary	35
3.3.1	Basic structure	35
3.3.2	Performing insertions	35

3.3.3	Performing lookups	36
3.3.4	Performing deletions	36
3.3.5	Requirements for coding	37
3.3.6	Reed Solomon codes	37
3.3.7	Pure erasure codes	38
3.3.8	Time complexity implications of using erasure codes	38
3.3.9	Space consumption	39
3.4	A $O(n(\log u + \beta))$ bits static PDM dictionary	40
3.5	Dynamic PDM dictionary	42
3.5.1	Basic structure	42
3.5.2	Emulation of PDHM by PDM	42
3.5.3	Storing A using the emulator	44
3.5.4	I/O complexity of insertion	44
3.5.5	I/O complexity of deletion	44
3.6	Conclusion	44
4	Two Probe schemes	46
4.1	Static Two Probe scheme using $O(n \log \log u)$ words	46
4.1.1	Versus Perfect Hashing	46
4.1.2	The idea	47
4.1.3	Solving the variable size issue	47
4.1.4	Identifying assignment owners in the dynamic case	48
4.2	Dynamic Two Probe scheme using $O(n \log n)$ words	48
4.2.1	Static data structure with reverse mapping	48
4.2.2	Dynamic data structure	49
4.2.3	Conclusion	50
5	Worst Case Insertion	51
5.1	Introduction	51
5.1.1	Preliminaries	51
5.2	Augmenting paths	52
5.2.1	Introduction	52
5.2.2	Matching and augmenting paths	53
5.2.3	Matchings in the induced assignment subgraph	54
5.2.4	Generality and length of augmenting paths	55
5.3	The Structure of the Induced Assignment subgraph	58
5.3.1	The importance of a larger expansion than assignment constraint	58
5.3.2	Viewing assignments as sets	58
5.3.3	Increased expansion factor	60
5.3.4	A new insertion scheme	61
5.3.5	Finding a worst-case for the new insertion scheme	62
5.3.6	Augmenting paths on parallel disk models	63
5.3.7	Buffering insertions	64

5.4	Conclusion	64
6	Feasible Expander Graph Construction	66
6.1	Introduction	66
6.2	Preliminaries	67
6.2.1	Expander construction	67
6.2.2	Graph notation	68
6.2.3	Expanders as randomness enhancers	68
6.2.4	Expanders	70
6.2.5	Extractors	73
6.3	Expanders and Eigenvalues	74
6.3.1	Lower bounds on graph expansion	74
6.3.2	The adjacency matrix of expanding graphs	74
6.3.3	The spectral gap	74
6.4	The Cartesian Graph Product	76
6.4.1	Properties of the Full Cartesian Graph Product	76
6.4.2	The Right Cartesian product and list expanders	78
6.4.3	Trading space for a smaller degree	82
6.4.4	Conclusion	83
6.5	The Original Zig-Zag product	85
6.5.1	The setup	85
6.5.2	How to follow an edge in the zig-zagged graph	85
6.5.3	Definition of the Zig-Zag product	87
6.5.4	The expansion property	87
6.5.5	Conclusion	89
6.6	The new Zig-Zag product	90
6.6.1	Introduction and preliminaries	90
6.6.2	Performing the new Zig-Zag product	91
6.6.3	Using the Zig-Zag product	95
6.7	Obtaining a feasible construction	97
6.8	Conclusion	101
7	Conclusion	102
A	Notation	103
B	The zig-zag theorem	105

Chapter 1

Introduction

1.1 Dictionaries and Expanders

One of the most fundamental information requests is the question "What, if any, is the information associated with 'x'?" This type of functionality is for example available in a perfectly ordinary dictionary, where each word is associated with an explanation, and the words are ordered alphabetically in order to speed up the search for a specific word.

A *dictionary data structure* is quite similar to such an ordinary dictionary. It also specifies some alphabet of which words, here denoted *keys*, are formed, and the associated information with these keys are called *satellite data*. In a dictionary data structure the essential performance parameters are the space used to store the information and the time required to find keys and retrieve their satellite data. In a dynamic dictionary the time required to insert or remove keys is also essential.

We define such a data structure formally in the following:

Definition 1.1 (Dictionary Data Structure). *Given a set of keys U , a set of values M , and a function: $f : S \rightarrow M$ where $S \subseteq U$ is the set of keys inserted in the data structure. A dictionary data structure provides a single operation which given an element x in U determines if it is present in S and if so provides the value $f(x) \in M$. If the data structure is dynamic it allows S and f to change dynamically.*

A dictionary data structure is a classical basic data structure used as a subcomponent in almost all applications. The central problem in making a dictionary data structure is to somehow map all the individual keys in the universe to some places in the computer memory so that we are able to find the keys and return their satellite data, or answer 'no' if they are not present in the dictionary.

A naive approach to create such a data structure is simply to make a table with an entry for each possible key. However most of the possible

combinations of letters from the alphabet will not be present in the dictionary. For example the word 'xzyqz' is unlikely to be found in any dictionary. The same applies to the dictionary data structure. We do not wish to use some space for each possible key since the universe of possible keys can be much larger than the actual number of keys present in the dictionary. It seems doubtful anyone would buy a (very thick) dictionary listing all possible words consisting of, say 100 letters, simply listing most of them as non-existent.

There are different ways to solve this problem much more efficiently than the naive approach, the most widely used are hash-functions and comparison based searches of the inserted keys. We will summarize such related techniques in the next section. We adopt a different, fairly recently discovered, approach which is to utilize an *expander graph*, to assist in mapping keys to their data in the dictionary. Informally put an expander graph is a graph where we can take any subset S of the nodes of at most some restricted size and be guaranteed that the set of neighbors to this set is some factor larger than the set itself. Most graphs are in fact good expanders, meaning that if we construct a graph randomly it will be an expander graph with very high probability. General expander graphs have many applications outside dictionary data structures, but in this thesis we will use the expander graph to provide a mapping between the universe of keys and their data, and therefore we restrict our attention to bipartite expander graphs, that is, expander graphs where the nodes can be partitioned in to two sets, left and right, where all the edges have one end in the left and one end in the right set. Below we give a formal definition of such a graph.

Definition 1.2 (Expander graph). *A bipartite graph $G = (U, V, E)$, where the edge set $E \subseteq U \times V$, is left d -regular if the degree of all nodes in U is d . A bipartite left d -regular graph $G = (U, V, E)$ is an (n, d, α) -expander if for each $S \subseteq U$ with $|S| \leq n$ it holds that $|\Gamma(S)| \geq \alpha|S|$.*

A substantial part of this thesis will consider dictionary data structures in a setting where the inserted keys are assumed to be too numerous to be stored in internal memory and must instead be placed in external memory such as a hard disk. Specifically we will consider an interesting special case of this where we are allowed parallel access to external memory, either using a multi-headed hard disk or multiple hard disks in parallel.

An example application As an example application of a dictionary in external memory, we will consider a very simple file system without directories where files can be named using 8 letters from the British alphabet. A single letter can then be represented using $\lceil \log_2 25 \rceil = 5$ bits. Furthermore assume that each file can consist of up to 256 segments. In order to read from such a file system we wish to, given a file name and a segment number, to retrieve the file segment in question as fast as possible. We can view the

file name and the segment number together as a key in the universe of possible requests to a dictionary. In this specific case each key can be represented as 6 bytes, or 48 bits and hence the universe is of size 2^{48} . We will see how it is possible, given parallel access to enough disks, to retrieve a segment of such a file in a *single* read operation, such that only a constant fraction of the combined read data does not belong to the segment in question. As an added benefit this means that the block size of the file system can actually be nearly the number of disks times the block size of a single disk, such that most file segments will fit in a single block.

1.2 Previous work on dictionaries

A significant lower bound regarding dictionaries is due to Yao [Yao81]. Yao's bound states that the worst case time complexity of dictionary lookups on a restricted RAM model allowing words to contain only elements of S or symbols from a fixed set of size $|S|$ (for example pointers) is $\Omega(\log |S|)$. This bound extends to an $\Omega(\log |S|)$ bound on the expected time of randomized Las Vegas lookups. Yao's bound assumes that the space usage is constant as a function of the size of the universe. Thus allowing a small dependency on $|U|$ circumvents this bound and theoretically allows sub-logarithmic lookups.

The most commonly used data structures for supporting dictionary style queries are balanced trees (or sorted lists for the static case) and hash tables. The first of these use $O(\log(|S|))$ for lookups and updates, matching a lower bound for pointer based dictionaries. Standard hash tables provide $O(1)$ time lookups for the static setting by using perfect hashing, or $O(1)$ time if the input is assumed to be uniformly random or the hash function is chosen randomly independent of the input.

More advanced data structures exist for supporting dictionary queries. For dynamic deterministic dictionaries for n keys with lookup time $t = o(\log \log n)$ the best known update time is $n^{O(1/t)}$ obtained in [HMP01]. The best known probabilistic bound for general parameters is that of [DadH90] where each operation takes constant time with probability $1 - O(m^{-c})$, where c can be any constant, and $m = O(n)$.

An interesting result for strictly membership queries, is that of [BMRV00], which by using expander graphs provide a static data structure which uses $O(n \log |U|)$ bits of space, and answers membership queries with a single bit-probe. The answer is correct with probability $1 - \epsilon$ where ϵ can be any constant larger than zero. A variation is given using $O(n^2 \log |U|)$ bits where there is no chance of giving a false negative answer. These results were non-constructive in that they only show the existence of the necessary expander graphs, but do not show how to construct and store them in space less than U .

This result was followed up on in the article "One Probe Search" by

Rasmus Pagh and Anna Östlin [OP02]. Using the same basic technique as in [BMRV00], they instead consider the cell probe model of complexity, and first show the existence of a static dictionary data structure that allows dictionary queries in a single cell-probe, succeeding with probability $1 - \epsilon$. The data structure guards against possible wrong answers, instead answering 'don't know', thus allowing an algorithm to re-query in order to ensure a correct answer. Secondly, a dynamic data structure is given in which a insertions/deletions and b lookups takes time $O(a(\log \frac{2u}{n})^{1+o(1)} + b + t)$ with probability $1 - 2^{-\Omega(a+t/(\log \frac{2u}{n})^{1+o(1)})}$. Both data structures use space $O(n \log \frac{2u}{n})$ words, not counting the space usage of storing the edges of the underlying expanding graph. Again, the results are non-constructive due to the lack of optimal space efficient expander graphs of the type required.

In external memory, B trees offer lookups and updates in $O(\log_B(N/B))$ I/O's, where B is the block size. Furthermore it is possible using hashing to get arbitrarily close to 1 I/O expected for lookups and updates when B is large, space utilization is fixed to a constant smaller than one, and under the usual assumption of uniform hashing [Knu98].

A recent result by Pagh and Skaarup on external memory hashing [JP05] yields probe complexity $1 + O(1/\sqrt{B})$ expected while using $(1 + O(1/\sqrt{B}))N/B$ blocks of space expected, under the assumption that a hash function can be chosen randomly and independent of the input.

1.3 This Thesis

1.3.1 Thesis Overview

Our thesis is based primarily on the results contained in previously mentioned article 'One Probe Search' by Rasmus Pagh and Anna Östlin referenced as [OP02]. We will therefore later in this chapter discuss the major results contained in this article and do a walk-through of the techniques used to obtain it. The second chapter concerns the open problem lower bounding the space consumption of a randomized one probe scheme, which was raised in aforementioned article. Chapter three proceeds to introduce a new dictionary data structure for the parallel disk models, based on an extension of the data structure in [OP02]. Following the parallel results is a small chapter addressing another open problem from the [OP02] article. It is the question of whether or not the use of two probes instead of just one can substantially reduce the space consumption. In the chapter entitled "Worst Case Insertion" we study the possibility of an insertion algorithm with good worst case insertion time. The original OPS data structure as well as our extension uses an insertion algorithm which only gives an amortized guarantee on the time complexity of an insertion. The final chapter concerns construction of the expander graph. All our results as well as those

in [OP02] rely on a small representation of this graph, which is a much studied problem that hasn't yielded entirely satisfactory results yet. The final chapter therefore covers the most common construction methods and a new result combining them to allow a feasible construction of an expander graph for a limited range of problem instances.

1.3.2 How we cite results and proofs

When we restate theorems and lemmas from the literature the original ones are cited in the description. We might have notational differences in our restated versions. When we give a proof of such a lemma or theorem from the literature it can be more or less identical to the original proof. Below is an explanation for what we mean when calling a proof restated, elaborated, and so forth.

Restated A restated version of a proof is one with only or almost only notational difference to the proof given in the literature.

Elaborated An elaborated version of a proof closely follows the argumentation of the proof in literature but we have filled out all argumentational jumps and cleared potential misunderstandings by further explanations.

Following A proof a following a proof b from the literature is using the same general idea as b but has substantial difference in the detail or is a proof of something else and therefore different.

When the theorem or lemma is our own result, it will have no citing in the description. Likewise a proof having neither of the above mentioned labels is our own proof. It could be a proof of one of our own lemmas or theorems or it could be our own proof of some well known fact.

If there are any special cases failing to fall into these categories it will be explicitly mentioned in the text.

1.3.3 Definitions and naming standards

As we deal with dictionaries and expander graphs in all chapters we will here introduce some standard names for concepts related to them.

In describing dictionaries we will denote the universe of keys by U and its size as u . The set of inserted keys is S . All dictionaries in this thesis will be of the kind where we have some maximum size of S , i.e. there is a limit to how many keys can be inserted at the same time. We will denote this limit n . In the dynamic versions of the dictionaries it will often be the case that we can use standard rebuilding techniques if we exceed this limit, by

keeping $\frac{|S|}{2} \leq n \leq |S|$. Without loss of generality we will at times assume that we always have n inserted elements.

As we at all times use an expander graph in the same way, as a mapping from some dictionary universe to places in memory, we will denote nodes of the left set U of an expander graph as keys and nodes of the right set V as elements. Furthermore the left degree will be denoted by d , unless otherwise is stated. A key $x \in U$ has the neighbor set $\Gamma(x) \subseteq V$. We assume that we have an ordering of the Γ -function in such a way that we can ask for a key's neighbor number i and always get the same element in $z \in V$. We define $\Gamma_i(x)$ as the i 'th neighbor of x under this ordering.

In this thesis we will without loss of generality deal with the alphabet $\{0, 1\}$, so that the keys of a universe are bit-strings and we will consider that some universe containing u keys is the universe of bitstrings with length $\lceil \log u \rceil$. Likewise the edges of some key $x \in U$ from some expander graph will be seen as a set D of bit strings with length $\lceil \log d \rceil$.

1.4 One Probe Search

In this section we will cover the results from [OP02] and show how they were obtained as this is the main result on which we base our further studies.

1.4.1 Preliminaries

Definition 1.3 (Assignment). *Let $G = (U, V, E)$ be a bipartite left d -regular graph. An assignment for a set $S \subseteq U$, is a subset $A \subseteq E \cap (S \times \Gamma(S))$ such that for any $v \in \Gamma(S)$, $|A \cap (S \times \{v\})| = 1$*

A balanced assignment is similar to a matching in a left regular bipartite graph with the demand that each left side node is matched at least to some number of right side nodes. We will throughout this thesis adopt the notation that for a key $x \in U$ the subset of $\Gamma(x)$ that is assigned to x is called $\Gamma_{asn}(x)$.

Definition 1.4 (Balanced assignment). *Let $G = (U, V, E)$ be a left d -regular bipartite graph. An $(1 - \epsilon)$ -balanced assignment for $S \subseteq U$ is an assignment $A \subseteq E$, where for each $x \in S$ it holds that $|A \cup (\{x\} \times \Gamma(x))| \geq (1 - \epsilon)d$.*

We will now see that expander graphs allows such balanced assignments, in order to do so we will use Hall's Theorem:

Theorem 1.5 (Hall's Theorem). *In any bipartite graph $G = (U, V, E)$, where for each subset $U' \subseteq U$ it holds that $|U'| \leq |\Gamma(U')|$ there exists a perfect matching.*

A perfect matching is defined in the chapter on worst case insertion in the section 5.2.2.

Lemma 1.6 (Restated [OP02] lemma 10). *If a graph $G = (U, V, E)$ is an $(n, d, (1 - \epsilon)d)$ expander then there exists a $(1 - \epsilon)$ balanced assignment for every set $S \subseteq U$ of size at most n .*

Restated proof: Let $S \subseteq U$ be a subset of size at most n and $G' = (S, \Gamma(S), E')$ be the subgraph induced by the keys of S and $\Gamma(S)$.

Since G is an $(n, d, (1 - \epsilon)d)$ -expander we know that for each subset $S' \subseteq S$ that $|\Gamma(S')| \geq (1 - \epsilon)d|S'|$. Let i be the number of left perfect matchings from S to disjoint subsets of $\Gamma(S)$ and let $M \subseteq V$ denote the elements from $\Gamma(S)$ in the matchings. Observe that due to Hall's theorem we know that $i \geq 1$. For each subset $S' \subseteq S$ it holds that $|\Gamma(S') \setminus M| \geq ((1 - \epsilon)d - i)|S'|$. If $((1 - \epsilon)d - i) < 1$ then we already have the required number of perfect matchings and therefore also the balanced assignment. If on the other hand $((1 - \epsilon)d - i) \geq 1$ the condition in Hall's Theorem is satisfied in the graph G_i were we have removed M from V . From this it is easily seen repeating this argument recursively that there must exist $(1 - \epsilon)d$ perfect matchings that are edge disjoint for each left element, and therefore also an $(1 - \epsilon)$ -balanced assignment. \square

1.4.2 Static data structure

The data structure is build on the concept that each key x from the left side U of an expander graph $G = (U, V, E)$ represents a key from the set of possible insertions.

For keys x not inserted in the data structure we will ensure that an $1 - \epsilon$ fraction of the neighbors of x either contains the symbol $\neg x$ or \perp .

For inserted keys y we will make sure that an $1 - \epsilon$ fraction of the neighboring elements of y will store a symbol representing y appended with some satellite data. That this is possible follows directly from Lemma 1.6 with the appropriate choice of expander. In order to check if a key z is inserted or not we randomly choose one of the d edges leaving z in U , and read the data contained in the neighboring element $\Gamma_i(z)$. If $\Gamma_i(z)$ contains z we will know for sure that z is an inserted. If $\Gamma_i(z)$ contains $\neg z$ or \perp we will know that z is not inserted. Should we encounter any other value we will not be able to decide if z is inserted. Intuitively we would like to avoid using $\neg z$ symbols, since they can only tell us that a single key is not inserted.

It would be better if all not inserted keys had an $1 - \epsilon$ fraction of their neighbors containing \perp , which is a shared negative symbol for all keys that have an edge to it. However there is a possibility that some not inserted keys share more than an ϵ fraction of their neighbors with the neighbors of the inserted set. Such keys are called ϵ -ghosts and are defined below:

Definition 1.7 (ϵ -ghost). *Given a graph $G = (U, V, E)$ a key $x \in U$ is an ϵ -ghost for the set $S \subseteq U$ if $|\Gamma(x) \cap \Gamma(S)| > \epsilon|\Gamma(x)|$ and $x \notin S$.*

ϵ -ghosts needs a special treatment compared to other keys that are not inserted because we cannot allow a $1 - \epsilon$ fraction of their neighbors to contain \perp since that would mean allowing the occurrence of false negatives which we would like to avoid. We will denote by \bar{S} the set of ϵ -ghosts for a set $S \subseteq U$. The following observation was originally stated in [BMRV00], but we give our own simple proof below.

Lemma 1.8. *There are at most n ϵ -ghosts for a set S of size n in a $(2n + 1, d, (1 - \epsilon/2)d)$ expander.*

Proof. Let \bar{S}_a be any set of ϵ -ghosts to the set S of cardinality a where $a = \min\{|\bar{S}|, n + 1\}$, then by the expansion property it must be the case that:

$$|\Gamma(\bar{S}_a) \cup \Gamma(S)| \geq (1 - \epsilon/2)d(n + a)$$

But we also know by the definition of ϵ -ghosts that ϵd of each ghosts neighbors must be in $\Gamma(S)$. This gives us a limit of how large $|\Gamma(\bar{S}_a) \cup \Gamma(S)|$ can be. The neighborhood of S can at most be nd and each ϵ -ghost can at most contribute with $(1 - \epsilon)d$ neighbors outside the neighborhood of S . Hence we get the following inequality :

$$\begin{aligned} (1 - \epsilon/2)d(n + a) &\leq |\Gamma(\bar{S}_a) \cup \Gamma(S)| \leq dn + (1 - \epsilon)da \\ \implies n + a - n\epsilon/2 - a\epsilon/2 &\leq n + a - a\epsilon \\ \implies a &\leq n/2 + a/2 \end{aligned}$$

This equation holds for $a \leq n$ but not for $a = n + 1$ which is the case if $|\bar{S}| \geq n + 1$. \square

Since we need to allow expansion for a set large enough to include both the ϵ -ghosts and the inserted keys in order to limit the number of ghost, we can conveniently also ensure that the ϵ -ghosts can in fact allow themselves to have either \neg or \perp in an $1 - \epsilon$ fraction of their neighbors. There is of course no further problems with ghosts in doing this, as we don't mind \perp symbols in the neighbors of ghosts.

Note that if we were allowed to give false negative answers we could disregard the whole concept of ϵ -ghosts and just answer "no" if we read anything different from our own value. This does not affect the inserted keys, and keys not inserted will always give the right answer since they do not appear at all in V . In this form the data structure is nearly identical to the bit-vectors from [BMRV00], just using cell probes instead and allowing the lookup of satellite data.

Constructing the data structure

Constructing the data structure for a set of inserted data S proceeds as follows:

1. Write \perp in all entries not in $\Gamma(S)$
2. Find the set \bar{S} of ϵ -ghosts for S
3. Find a $(1 - \epsilon)$ -balanced assignment for the set $S \cup \bar{S}$
4. For $x \in S$ write x in entries assigned to x .
5. For $x \in \bar{S}$ write $\neg x$ in all entries assigned to x in $\Gamma(S)$

The important thing to notice here, is that step 2 can potentially take time $\Omega(|U \setminus S|)$ time. Ta-Shma manages in [TS02] to construct explicit expander graphs in which this operation can be done in polynomial time in $|V|$. The remaining steps are easily handled in polynomial time in $|V|$.

Also note that in order to perform a probe on this data structure we need to know which expander graph to use for the probe. As long as we know the capacity of the data structure this expander graph can be uniquely defined, and hence we need not waste a probe on discovering which expander graph is used.

1.4.3 Dynamic updates

We consider a sequence of a insertions/deletions and b lookups. We will here denote by n the maximum number of elements the data structure can hold, i.e. we will not necessarily have that $|S| = n$. We will now cover the technique used in [OP02] to get a probe complexity of $O(a \cdot d \cdot \log d + b + t)$ with probability $1 - 2^{-\Omega(a+t/d \log d)}$, where d is the degree of the expander and $t > 0$.

It is shown later in Lemma 3.4 that it is possible to obtain $d = O(\log \frac{u}{n})$ where u is the size of the universe. Furthermore this degree is asymptotically optimal as we will see in section 2.2.2.

ϵ -ghosts in the dynamic setting In the dynamic setting we will not easily be able to trace which keys are ϵ -ghosts, and so the set of keys that are currently identified by the data structure as ϵ -ghosts is not necessarily at all times equivalent to \bar{S} . We will denote the set of keys currently identified as ϵ -ghosts as \tilde{S} . The operations performed on the data structure are divided into stages each containing n insertions and after each such stage we perform some cleaning up to ensure that $\tilde{S} = \bar{S}$.

Insertion and deletion

Insertion When inserting a new key into S we need to assign $(1 - \epsilon)d$ neighbors to it, while ensuring that no other keys in $S \cup \tilde{S}$ are brought below the minimum valid assignment size. The insertion algorithm is a simple 'grab all' algorithm assigning all neighbors to the newly inserted key,

possibly stealing elements of V previously assigned to other keys in $S \cup \tilde{S}$. If any other keys are suffering from a too small assignment after this, each of them grab all of their neighbors, etc. There are three matters of concern with this approach. The first is how we are going to discover when the assignment of some key in $S \cup \tilde{S}$ has gone below the minimum allowed size. This will be done by using a simple linear sized table described below. The second is whether the algorithm terminates at all and if so, how many assignments it performs. The last is how we are going to keep track of which keys in $U \setminus S$ are ϵ -ghosts and therefore need special care of their neighborhood, i.e. we need to maintain the set \tilde{S} .

Deleting When deleting a key we have to know if it has become an ϵ -ghost or not. We do this by in every element of V maintaining a counter saying how many neighbors it has in S . When deleting x we decrease all the counters of $\Gamma(x)$ by one. If the counter of an element has become 0 it is safe to write \perp in that element, otherwise there are still some keys in S that might look it up so we just mark the element as vacant. But if over ϵd of the neighbors would be such marked as vacant, we know that x has become an ϵ -ghost and we must instead write $\neg x$ in all elements previously occupied by x . All this work will only take $O(d)$ time. So the amount of work done for all deletions can never asymptotically exceed the amount of work for the insertions, as we can not delete more elements than we have inserted and we as a minimum need to do $(1 - \epsilon)d$ work in an insertion. Note that after a deletion some keys in \tilde{S} might no longer be ϵ -ghosts. We will see how to handle this later.

Discovering too small assignment sizes In the OPS paper a priority min-queue is used to discover when, after some grab all step, the assignment size of some other inserted keys have gone below the limit. The queue contained all keys of $S \cup \tilde{S}$ with their assignments size as the priority, and so it could after each insertion step easily be checked if any assignment sizes was too small. But a priority queue is not really needed in order to support the grab all algorithm as we will see later. As long as those keys that grab all their neighbors have too small an assignment, it is irrelevant whether or not they are in fact the one with the smallest assignment. This means that we instead of using a priority queue, simply can rely on an array storing in its entry a key and its current assignment size. Discovering which keys are being reduced to an unacceptable assignment size can be discovered simultaneously with changing the counter during the grab all step.

Maintaining \tilde{S}

Optimally we would like that $\tilde{S} = \bar{S}$ at all times. When inserting a new key in S some keys in $U \setminus (S \cup \tilde{S})$ may have become ϵ -ghosts and when

deleting a key from S there might be some keys in \tilde{S} that are no longer ϵ -ghosts. Undetected ϵ -ghosts are then keys of $U \setminus S$ where more than ϵd of its neighbors, when probed, will answer “don’t know”.

Dealing with undetected ϵ -ghosts We deal with undetected ϵ -ghosts in the following way. When lookup of some key takes some fixed amount of time more than what we expected, we examine it, since it is then likely to be an ϵ -ghost. An examination of a key is done by looking all the neighbors up and thereby checking if it is an ϵ -ghost. The time used for lookup of the undetected ghosts and for examination is summed up in section 1.4.4. When we have discovered an ϵ -ghost $x \in U$ by examination we will insert it in \tilde{S} which amounts to the same work done as in an ordinary insertion except that we write $\neg x$ and not x in the assigned neighbors.

We mark a key for examination if a lookup of it answers “don’t know” more than $\log_{1/\epsilon} d$ times. For a key that is not an ϵ -ghost i.e. having at most ϵd neighbors not assigned to it, this will happen with probability at most

$$\epsilon^{\log_{1/\epsilon} d} = \frac{\epsilon^{\log_{1/\epsilon} d} \cdot (1/\epsilon)^{\log_{1/\epsilon} d}}{(1/\epsilon)^{\log_{1/\epsilon} d}} = \frac{1^{\log_{1/\epsilon} d}}{d} = \frac{1}{d} = O(1/d)$$

Likewise for an undetected ϵ -ghosts an examination will happen with probability at least $(\epsilon + \frac{1}{d})^{\log_{1/\epsilon} d} = \Omega(1/d)$, since it has at least $\epsilon + \frac{1}{d}$ of its neighbors saying “don’t know”.

The number of undetected ϵ -ghosts We need a bound on the number of undetected ϵ -ghosts that can emerge during a stage. From lemma 1.8 we know that if we use a $(2x + 1, d, (1 - \epsilon/2)d)$ -expander graph then for some inserted set of size x there can be at most x ϵ -ghosts. If we denote by S the set of keys inserted at the beginning of a stage and S' the set of keys inserted in a stage, we know that $|S| + |S'| \leq 2n$.

If we ensure that the expander graph expands for sets of size $2n + 1$, we will know that at any time we only have n ϵ -ghosts, but the set of key that become ϵ -ghosts in one stage might be much larger than n . During a stage we might have n different sets inserted, many of them differing only on a single key, but even though they might have very different \tilde{S} sets. And so we have no bound on the number of key that become ϵ -ghosts during a stage. But if we ensure that the expander graph expands for sets of size $4n + 1$, we have a bound on this number since we will know that the set $S \cup S'$ only has $2n$ ϵ -ghosts all in all.

An important notion is that during a stage we can then have n insertions in S and $2n$ insertions in \tilde{S} , so we have to perform assignment for $3n$ keys in one stage.

Maintaining the $(1 - \epsilon)$ -balanced assignment

The dominating time factor in an insertion is the time spent on assignments and reassignments. In this section we will by 'assigning' refer both to assigning a vacant element of V to some key and reassigning some element of V from some key to another. When inserting a key our 'grab all' assignment scheme will make a number of assignments depending on how many times some other key suffers from a too small assignment. We would like to show that over a sequence of n insertions the number of assignments will be $O(nd)$ giving us an amortized time bound of $O(d)$ for an insertion.

The proof has two steps. First it is proved that the grab all scheme will only make a constant factor more assignments than any scheme maintaining a $(1 - \frac{\epsilon}{3})$ -balanced assignment, and then we will prove the existence of such a scheme using only $O(d)$ assignments in every update. The trick is that the scheme maintaining the $(1 - \frac{\epsilon}{3})$ -balanced assignment is off-line and therefore easier to analyze.

To ensure that there exists a $(1 - \frac{\epsilon}{3})$ -balanced assignment we need the expander we use to have an expansion factor of $(1 - \frac{\epsilon}{3})d$. Combined with the previously mentioned need for the expanding set to be of size $4n + 1$ we note that a $(4n + 1, d, (1 - \frac{\epsilon}{3})d)$ -expander will suffice for the described data structure.

Lemma 1.9 (Restated [OP02] lemma 13). *Let $G=(U, V, E)$ be a d -regular bipartite graph. Suppose O is a sequence of insert and delete operations on a dynamic set $M \subseteq U$. Let B be an algorithm that maintains a $(1 - \frac{\epsilon}{3})$ -balanced assignment for M , and let C be our 'grab all' scheme for maintaining a $(1 - \epsilon)$ -balanced assignment for M . If B makes at most k reassignments during O , then C assigns all neighbors to a key at most $\frac{3}{\epsilon}(\frac{k}{d} + |M|_{start})$ times, where $|M|_{start}$ is the initial size of M .*

Restated proof: Let $A_C \subset E$ be the assignment of C and $A_B \subset E$ be the assignment of B . We will argue that when C assigns all neighbors of a key to itself A_C and B_C will become "less different".

We will first show that the assignments made by C during O can decrease $|A_B \setminus A_C|$ by at most $k + d|M|_{start}$. At the beginning we have that $|A_B \setminus A_C| \leq d|M|_{start}$, since $|A_B| \leq d|M|_{start}$. Each of the k assignments B performs causes $|A_B \setminus A_C|$ to increase by at most one, since it can increase $|A_B|$ by at most one. This means that the assignments made by C during O can decrease $|A_B \setminus A_C|$ by at most $k + d|M|_{start}$ in total, since this is the maximal size that $|A_B \setminus A_C|$ can have.

Each time C assigns all entries in $\Gamma(x)$ to a key x , at least ϵd assignments are changed, since the assignments neighborhood for x had size less than $(1 - \epsilon)d$ before the assignment.

We know that at least $(1 - \frac{\epsilon}{3})d$ of the edges of x are included in A_B at this point. This means that at least $(1 - \frac{\epsilon}{3})d = \frac{2\epsilon}{3}d$ of the assignments

made by C decrease $|A_B \setminus A_C|$. Intuitively, since C assigns all neighbors to x assigning the exact $(1 - \frac{\epsilon}{3})d$ edges assigned to x in A_B will make A_B and A_C this much more “less different”. We also know that at most $\frac{\epsilon}{3}d$ of the edges of x are not included in A_B at this point. This means that at most $\frac{\epsilon}{3}d$ assignments made by C may increase $|A_B \setminus A_C|$. These are the edges not assigned to x in A_B , but which C assigns anyway because of its assign all strategy.

In total $|A_B \setminus A_C|$ is decreased by at least $\frac{\epsilon}{3}d$ when C assigns all neighbors to a key. The lemma now follows, as $|A_B \setminus A_C|$ can decrease by $\frac{\epsilon}{3}d$ at most $(k + d|M|_{start})/(\frac{\epsilon}{3}d) = \frac{3}{\epsilon}(\frac{k}{d} + |M|_{start})$ times. \square

Now we have shown that the grab all scheme is only a constant factor slower than any scheme B maintaining a $(1 - \frac{\epsilon}{3})$ -balanced assignment. It may even be that B is off-line, which we will use in the following. Here we will describe just such a scheme and prove that it performs $O(d)$ reassignments per update. Remember that we have to account for $3n$ insertions in $S \cup \tilde{S}$ when performing n insertions in S .

Lemma 1.10 (Restated [OP02] lemma 14). *Let $G=(U, V, E)$ be a $(4n, d, (1 - \frac{\epsilon}{3})d)$ -expander. There exists an off-line algorithm maintaining a $(1 - \frac{\epsilon}{3})$ -balanced assignment for a dynamic set $M \subseteq U$, during a stage of $3n$ insertions, by performing at most $4dn$ assignments, where $|M| \leq n$ at the beginning of the stage.*

Restated proof: Let M' be the set of $3n$ keys to insert. We have that $|M \cup M'| \leq 4n$. Let $A_{M \cup M'}$ be a $(1 - \frac{\epsilon}{3})$ -balanced assignment for $M \cup M'$. Such an assignment is shown to exist by Lemma 1.6. Then there exists an off-line algorithm which by knowing M and M' also knows $A_{M \cup M'}$. We do not need to show how it knows $A_{M \cup M'}$ but only note that it is always possible to know $A_{M \cup M'}$ when the graph, M , and M' are known. We will show that the off-line algorithm can fulfill the requirements simply by knowing the assignment $A_{M \cup M'}$ and making assignments according to it.

The off-line algorithm does the following. First it assigns neighbors to the keys in M according to the assignment $A_{M \cup M'}$. This requires at most dn assignments in all. Secondly for each insertion of a key $x \in M'$, it assigns neighbors to x according to $A_{M \cup M'}$. This will require at most d assignments for each key. The assigning of keys in M' will not cause any key already inserted to lose an assigned neighbor since all assignments are strictly following $A_{M \cup M'}$. It follows that the total number of assignments during the $3n$ insertions is at most $4dn$, proving the lemma. \square

We have now shown that if we use an expander with room for a constant factor larger assignment than we strictly need, we will in a sequence of n updates to the data structure spend $O(nd)$ time to perform the updates, i.e. to maintain an $(1 - \epsilon)$ -balanced assignment.

1.4.4 Time bounds

We know that assigning takes $O(ad)$ time and we know that examining the ϵ -ghosts also takes at most $O(ad)$ since we know there are at most $2a$ ϵ -ghosts all in all. We need to bound the time spent on lookup and on examining the keys that are not ϵ -ghosts. Remember that when looking up a key that is not an ϵ -ghost we will chose it for examination with probability $O(1/d)$ and when looking up an undetected ϵ -ghosts it will be chosen with probability $\Omega(1/d)$.

Looking up keys that are not ϵ -ghosts Each of these lookups has probability $(1-\epsilon)$ of only needing one iteration and all iterations are independent of all other events. By using Chernoff bounds we get that the probability of the number of iterations being more than $\frac{2}{(1-\epsilon)}b + t$ is less than $e^{-\frac{(1-\epsilon)}{4}t}$. Since ϵ and e are constants we can say that the time spent is $O(b+t)$ with probability $1 - 2^{-\Omega(t)}$.

Examining keys that are not ϵ -ghosts We might during a lookup falsely come to think that some key is an ϵ -ghost and then spend some additionally time examining it. The probability of selecting a non- ϵ -ghost to be examined is $O(1/d)$. The total number of these examinations will be at most $\frac{b}{d} + k$ with probability $1 - (\frac{e}{1+kd/b})^{b/d+k}$, for $k > 0$, using Chernoff bounds. Setting $k = (2e-1)b/d + t/d$ we have that the probability of at most $2eb/d + t/d$ examinations is at least $1 - 2^{-t/d}$. We know that each examination takes d time and so the probability of using $O(b+t)$ time on examining these keys is $1 - 2^{-\Omega(t/d)}$.

Looking up ϵ -ghosts before they are detected We know there are at most $2a$ ϵ -ghosts all in all since one stage can have at most $2n$.

Recall that the probability of selecting an undetected ϵ -ghost for examination is $\Omega(1/d)$. The probability that more than $4ad+k$ lookups are made on undetected ϵ -ghosts, for $k > 0$ is by using Chernoff bounds at most $e^{-k/4d}$. Each lookup costs $O(d)$ time and so the probability of using $O(ad \log d + t)$ time is at least $1 - e^{-t/4d \log d}$. Since e is a constant the probability of using $O(ad \log d + t)$ time is $1 - 2^{-\Omega(t/d \log d)}$.

Putting it together We have that

- assigning takes $O(ad)$ with probability 1
- looking up non- ϵ -ghosts takes $O(b+t)$ with probability $1 - 2^{-\Omega(t)}$
- examining non- ϵ -ghosts takes $O(b+t)$ with probability $1 - 2^{-\Omega(t/d)}$

- looking up undetected ϵ -ghosts takes $O(ad \log d + t)$ with probability $1 - 2^{-\Omega(t/d \log d)}$

The time complexity will then be $O(ad) + 2 \cdot O(b + t) + O(ad \log d + t) = O(ad \log d + b + t)$ with probability at least $1 - 2^{-\Omega(t/d \log d)}$. Furthermore, by setting $t = ad \log d + t'$ we get complexity

$$O(ad \log d + b + t')$$

with probability at least

$$1 - 2^{-\Omega(a+t'/d \log d)}$$

Chapter 2

One probe lower bound

In the One Probe Search paper [OP02], it was left as an open question whether or not the space usage was optimal. In this chapter we will first define lower bound models for two one probe schemes, the guarded and unguarded scheme, both applicable to the original OPS data structure. We then proceed to give a lower bound for the guarded scheme of $\Omega\left(n \log \frac{u}{n}\right)$ words, asymptotically matching the space usage in [OP02] and thus proving that the space usage in [OP02] is optimal. Finally we give a somewhat poorer bound of $\Omega\left(n \frac{\log u}{\log n}\right)$ words for the less restrictive unguarded scheme.

2.1 Schemes and lower bound models

In this section we formally define the unguarded and guarded one probe schemes and matching lower bound models. In section 2.4 we will briefly discuss the generality of the models, i.e. how well the models encompass all unguarded and guarded schemes respectively.

Definition 2.1 (Unguarded one probe scheme). *An unguarded one probe scheme is a dictionary with universe U , and an inserted set S , where the lookup function has the following characteristics:*

- *It only probes one cell of data.*
- *It returns either, 'no', 'don't know' or some satellite data (i.e. 'yes').*
- *It answers correctly with probability at least $1 - \epsilon$, for any constant $0 < \epsilon < 1$.*
- *It never gives a false positive answer.*

Definition 2.2 (Guarded one probe scheme). *A guarded one probe scheme is an unguarded one probe scheme with the additional characteristic that the lookup function never gives a false negative answer.*

Notice that the original OPS scheme is a guarded scheme. We will now define the two models for which the lower bounds are proved. We will later discuss to what degree the models encompasses the defined schemes.

Definition 2.3 (Lower bound model for unguarded schemes). *This model encompasses unguarded one probe schemes consisting exclusively of the following components:*

- An ordered table T_V with an entry for each element in a set V , where each entry in T_V consist of $\log(2u + 1)$ bits, where u is the size of the universe.
- A deterministic 'neighbor' function $\Gamma(x) : U \rightarrow V_x$, where $V_x \subseteq V$.
- The probe function P of the scheme, mapping a key x in U to a single element in $\Gamma(x)$ randomly according to some distribution over $\Gamma(x)$, independently of S .
- An implicit assignment function Γ_{asn} that for a given $S \subseteq U, |S| \leq n$ maps a key x in U to a set $\Gamma_{asn}(x) \subseteq \Gamma(x)$. (This function is not available to the scheme)

and having the following properties:

1. The assignment function Γ_{asn} must be so that for all distinct keys $x, y \in S$, $\Gamma_{asn}(x) \cap \Gamma_{asn}(y) = \emptyset$.
2. For all elements z in V , $T_V(z)$ stores the symbol ' x ' if and only if $z \in \Gamma_{asn}(x)$.
3. For all keys $x \in S$: $\Pr[P(x) \in \Gamma_{asn}(x)] \geq 1 - \epsilon$

Put informally, we demand that, for all inserted keys, the probe has more than probability $1 - \epsilon$ of reaching an element assigned to it, because otherwise it can not give a correct positive answer with probability at least $1 - \epsilon$. Notice that we have no bound on the size of $\Gamma(x)$, for any $x \in U$ and that the Γ -function could be biased, choosing to probe some element in $\Gamma(x)$ more likely than another. Furthermore observe that the space usage of such a scheme must be $|V| \cdot \log(2u + 1)$ bits as the $|V|$ entries in T_V each at least use $\log(2u + 1)$ bits.

Definition 2.4 (Lower bound model for guarded schemes). *Any guarded one probe scheme fits this model if it fits the model for unguarded schemes (def. 2.3) and have the additional property:*

4. For all keys $x \notin S$ $\Pr [P(x) \in \bigcup_{z \in S} \Gamma_{asn}(z)] \leq \epsilon$

In the model for guarded schemes we in addition demand that elements that are not inserted have a probability of probing an element assigned to an inserted key less than ϵ . If this was not the case the scheme would be either incapable of giving the correct negative answer with probability $1 - \epsilon$ or it would base its negative response on reading a neighbor assigned to some other key. If the latter is the case, it would not be possible to discern between a possible false negative and real negative when we perform a probe.

2.2 Lower bound for the space usage of the original OPS

We will in this section prove a tight space lower bound of $\Omega(n \log(u/n))$ for the guarded one probe scheme. This will imply that the original OPS data structure has optimal space usage since the original OPS data structure consists of an expander graph with a right side V of cardinality $O(n \log(u/n))$.

2.2.1 Tight lower bound for a guarded one probe scheme

Theorem 2.5 (Guarded one probe scheme lower bound). *For any guarded one probe scheme fitting the model from definition 2.4, with $\epsilon < 1/2$, it is the case that $|V| = \Omega(n \log(u/n))$.*

Proof. Let T_V be an ordered table of the elements in V and let B be a bit-vector of length $|V|$ such that $B(i)$ is 1 if $\exists x \in S$ s.t. $T_V(i) \in \Gamma_{asn}(x)$ and 0 otherwise. In other words, B is a bit-mask telling us which elements from V are used in the assignment for the given set S .

Consider the two bit-vectors B_1 and B_2 corresponding to two distinct sets S_1 and S_2 . We will show that B_1 is not identical to B_2 . Consider an element $z \in S_1, z \notin S_2$. By property 4, there is a set of elements Q in B_2 with value 0, such that $\Pr[P(z) \in Q] > 1/2$. If B_1 is identical to B_2 then $P(z)$ also has more than $1/2$ probability of reaching a 0 element in B_1 which violates property 3 for the set S_1 . By this contradiction we now have that there must exist a distinct bit-vector for each possible insertion set. There are $\sum_{1 \leq i \leq n} \binom{u}{i}$ such sets. Just counting the sets of size n the bit-vector will need to be able to assume $\binom{u}{n}$ values requiring at least $\lceil \log \binom{u}{n} \rceil = \Omega(n \log \frac{u}{n})$ bits. Since the bit-vector has the same number of bits as there are elements in V this implies that $|V| = \Omega(n \log \frac{u}{n})$ \square

2.2.2 Implied bounds on expanders

Through the OPS result this lower bound also lower bounds the right side of an expander with $\epsilon < 1/2$, that expands for sets of size n . Furthermore, consider an expander that expands for sets of size $n = O(|V|)/d$. For such an expander we also get an $\Omega(\log(u/n))$ bound on the the degree d , since

$$n = O(|V|)/d \Leftrightarrow d = O(|V|)/n = \Omega(n \log(u/n))/n = \Omega(\log \frac{u}{n})$$

2.2.3 The central role of Property 4

An interesting point to note about the above proof, is the central role that property 4 plays. If we give up this property it becomes much harder to prove a lower bound on the space usage. As mentioned earlier property 4 is tied to the notion that the one probe scheme must not be allowed to answer 'no' incorrectly, so it becomes natural to wonder whether a similar lower bound applies in a scenario similar to that of section 1.4, where we just allow false negatives. We will investigate this in the next section.

2.3 Lower bound for unguarded schemes

In this section we consider the case of a one probe scheme that doesn't require a guard against false negatives. We can therefore no longer rely on property 4 to give us a simple information theoretic bound.

2.3.1 Lower bound for an unguarded one probe scheme

Assumptions In the following we assume $\epsilon < 1/2$. We also assume that V has is large enough to contain any n elements from U and that $|V|/n < n$. Both assumptions will be the case in all reasonable instances of a one probe scheme. Most of the lemmas and theorems in this section uses these assumptions in their proof. In our proof, we will regard the probe function P of the one probe scheme in question as a directed graph with weighted edges. The only edges will be those from a key $x \in U$ to an element $v \in V$ where $\Pr[P(x) = v] > 0$, and such an edge will have weight $\Pr[P(x) = v]$.

As a shorthand we will define the weight function w taking an edge and producing a real number r , where $0 < r \leq 1$, where r is the probability function that this edge will be chosen by the probe function.

Definition 2.6. (x, Y) :

$$(x, Y) \stackrel{\text{def}}{=} \{(x, y) | y \in Y\}$$

$$w(x, Y) \stackrel{\text{def}}{=} \sum_{y \in Y} w(x, y)$$

Definition 2.7. $a_{\min}(x) : \min\{|E| \mid E \subseteq (x, \Gamma(x)) \text{ and } w(E) \geq (1 - \epsilon)\}$

Definition 2.8. $A_i : \{x \in U \mid a_{\min}(x) \leq i\}$

So the $a_{\min}(x)$ heaviest edges in the the neighborhood of x has accumulated probability of at least $1 - \epsilon$ of being chosen while the $a_{\min}(x) - 1$ heaviest

does not. A_i is the set of keys in U whose i 'th heaviest edges has a total probability of being probed at least $1 - \epsilon$.

Definition 2.9. $b_{min}(x) : \min\{|E| \mid E \subseteq (x, \Gamma(x)) \text{ and } w(E) > \epsilon\}$

Definition 2.10. $B_i : \{x \in R \mid b_{min}(x) \leq i\}$

B_i is the set of keys in U whose i 'th heaviest edges has a total probability of being probed strictly larger than ϵ . Such keys are significant, when inserted, because they have to be able to assign themselves on at least one of the i edges, as they otherwise cannot obtain a total weight on their assignment of at least $1 - \epsilon$.

Lemma 2.11. $a_{min}(x) \geq b_{min}(x)$

Proof. The inequality holds because $(1 - \epsilon) > \epsilon$ which follows from the assumption that $\epsilon < \frac{1}{2}$. \square

Lemma 2.12. $A_i \subseteq B_i$

Proof. $x \in A_i \stackrel{def}{\Leftrightarrow} a_{min}(x) \leq i \stackrel{*}{\Rightarrow} b_{min}(x) \leq i \stackrel{def}{\Leftrightarrow} x \in B_i$
(Step “*” is due to lemma 2.11) \square

Lemma 2.13. $|A_{|V|/n}| > u - n$

Proof. Assume towards contradiction the opposite i.e $|A_{|V|/n}| \leq u - n$. Then the n keys $x \in U$ for which $a_{min}(x) > |V|/n$ will each need to have more than $|V|/n$ elements in V assigned to them, if they are inserted in the data structure. Hence if we insert exactly these n elements more than $|V|$ entries in V are needed. We have arrived at a contradiction. \square

Lemma 2.14. $\frac{|B_{|V|/n}|}{\binom{|V|}{|V|/n}} \leq |V|/n$

Proof. $B_{|V|/n}$ is the set of keys whose $|V|/n$ 'th heaviest edges has a total probability of being probed of strictly more than ϵ . Recall that if such a key is unable to assign itself on at least one of its $|V|/n$ heaviest edges it will be unable to achieve an assignment with weight at least $1 - \epsilon$. Therefore the end elements of its $|V|/n$ heaviest edges can be shared with at most $|V|/n$ other inserted keys from $B_{|V|/n}$, as otherwise at least one inserted key will not be able to achieve an assignment of weight $\geq 1 - \epsilon$.

Assume towards contradiction that $\frac{|B_{|V|/n}|}{\binom{|V|}{|V|/n}} > |V|/n$. $\binom{|V|}{|V|/n}$ is the number of subsets of V containing exactly $|V|/n$ elements and hence $\frac{|B_{|V|/n}|}{\binom{|V|}{|V|/n}}$ is the average number of keys sharing a set of elements with more than ϵ weight for each of them. By the pigeon hole principle we know that at least one set must be shared by at least the average number of keys. Hence, if

$\frac{|B_{|V|/n}|}{\binom{|V|}{|V|/n}} > |V|/n$, then more than $|V|/n$ keys are sharing a set of size $|V|/n$ of neighbors of which all of them needs at least one, meaning that at least one inserted key will be unable to achieve its required assignment if all the keys sharing its $|V|/n$ most likely neighbors are inserted along with itself. Note that its possible to insert all these keys since $|V|/n < n$. \square

Lemma 2.15. $\frac{u-n}{(en)^{|V|/n}} \leq |V|/n$

Proof. By lemma 2.14 we have that $\frac{|B_{|V|/n}|}{\binom{|V|}{|V|/n}} \leq |V|/n$. Since $\frac{|B_{|V|/n}|}{\binom{|V|}{|V|/n}} > \frac{|B_{|V|/n}|}{\binom{|V|}{|V|/n}}$ and $A_i \subseteq B_i$ we have that $\frac{|A_{|V|/n}|}{(en)^{|V|/n}} \leq |V|/n$. Then by lemma 2.13 we get the result. \square

Theorem 2.16. $|V| > \frac{n \log u}{4 \log n}$ for $u > \max\{n^2, 64\}$ and $n > e$

Proof. Assume towards contradiction that $|V| = (1/4)n \frac{\log u}{\log n}$. By inserting $|V|$ in lemma 2.15 we get

$$\frac{u-n}{(ne)^{|(1/4)\frac{\log u}{\log n}|}} \leq (1/4) \frac{\log u}{\log n}$$

which by the assumption that $n > e$ implies

$$\frac{u-n}{n^{(1/2)\frac{\log u}{\log n}}} \leq (1/4) \frac{\log u}{\log n}$$

and since $u > n^2$ we have that

$$\frac{u}{(n)^{|(1/2)\frac{\log u}{\log n}|}} \leq (1/4) \frac{\log u}{\log n} + 1$$

by $\frac{u}{(2^{\log n})^{(1/2)\frac{\log u}{\log n}}} = \frac{u}{2^{(1/2)\log n \frac{\log u}{\log n}}} = \frac{u}{2^{(1/2)\log u}} = \frac{u}{(2^{\log u})^{1/2}} = \sqrt{u}$ we have

$$\sqrt{u} \leq ((1/4) \frac{\log u}{\log n}) + 1 \implies \sqrt{u} \leq \log u$$

Which contradicts our assumption that $u > 64$. \square

2.3.2 Is the unguarded scheme really less restrictive?

The bound attained for the unguarded scheme is significantly lower than the bound for the guarded scheme. The unguarded bound itself is only non-trivial for super-polynomial relations between u and n . We do think it is possible to achieve a bound that is asymptotically identical to the bound for the guarded scenario. To explore the differences between the guarded

and the unguarded scheme consider a restricted version of the unguarded scheme where the number of possible probe positions for each key in U is the same and all probe positions of a key has equal probability of being probed. This is equivalent to an expander graph where we have already have a lower bound of $\Omega(n \log \frac{u}{n})$ on V . If we remove the restriction on the number of probe positions for each key, we are left with a 'graph' with expander like properties but with no bound on the left degree. Specifically in any set of keys S of cardinality n or less, each key x_i must be able to achieve at least a $(1 - \epsilon)d_i$ matching in V , where d_i is the left degree of x_i . It is not hard to see that the irregularity of such a graph does not make it substantially easier to construct. In fact, it can easily be shown that a substantial number (nearly u) of keys must have degree $c|V|/n$, for some constant c , indicating that the graph actually contains a slightly smaller left regular expander graph inside it. So removing this restriction can not give any asymptotic space reduction. This allows us to conclude that any improvement over the guarded scheme, can only be due to allowing weighted 'edges' in the probe function. While we have not shown that this cannot make a difference, it is worth noting that for the guarded scheme it did not make a difference for the lower bound, whether or not we allowed weighted edges, as seen from the fact that expanders exist that asymptotically matches the guarded lower bound.

2.4 Generality of the lower bound models

We will here discuss how general the guarded and unguarded lower bound models (def. 2.3 and 2.4) and are compared to the defined schemes (def. 2.1 and 2.2). The questions are: could there exist some guarded or unguarded one probe scheme circumventing the lower bounds by not fulfilling the definition of the lower bound models and if such schemes exist could their space consumption be asymptotically less than the bound we proved in section 2.2 and 2.3.

Lookup function of an unguarded scheme The lookup function will probe a location in T_V by using the probe function and then it will make some calculations to determine whether this probed element is assigned to x , and only if it is, the lookup function will answer 'yes'.

Now consider what information the lookup function has available to help it decide whether the probed element is assigned to x . It has the following information:

- the key $x \in U$ to lookup
- the random choice r it gives to the probe function
- the assignment value p returned by the probe function

- the index number i in T_V probed by the probe function
- some information from the neighbor function about the neighborhood of x , i.e. its size, all of its indexes in T_V , and the individual probabilities of choosing them.

But notice that the only values that can vary on account of if x is in the inserted set or not, is p and i , all the others must be indifferent of this and so the lookup function can not use them to determine whether the element is assigned to x or not. One could think that we could use r as well, as the assignment function could use r like it could use i , by somehow interpreting the p value in the light of r . This can not help any scheme to determine some assignment problem since seen from some key, which is all the probe function will ever be able to do, the values i and r are fixed on each other.

It can be shown that the $\log |V|$ bits of i can at best be used to reduce the cell size in T_V by $\log |V|$ bits, which makes no asymptotic difference compared to our lower bounds, if $u = \Omega(n^k)$ for any $k > 1$.

2.5 Conclusion

We achieved a lower bound on the space usage for one probe schemes that answers correctly with probability at least $1 - \epsilon$ and otherwise answers “don’t know”. This lower bound matches the space usage of the original OPS data structure thereby showing that it is indeed optimal.

Further more we have shown somewhat weaker lower bound for the less restrictive scenario where the scheme is allowed to give a false negative answer.

Chapter 3

One Probe in the Parallel Disk Models

3.1 Introduction

An interesting theoretical question is how much more powerful an adaptive algorithm is as compared to a parallel one. The original OPS data structure, suggests a very simple data structure for parallel lookups: Instead of probing a single neighbor in the expander graph, we simply probe all of them. This means that we by $O(\log \frac{u}{n})$ probes done in parallel can do the same work that takes $O(\log n)$ adaptive probes in for example a balanced search tree or a sorted list. In this section we introduce such a data structure, and using error correcting codes¹ we efficiently distribute large satellite data so that it is stored with arbitrarily small redundancy.

The first part covers a very simple static data structure which does not require error correcting codes, while the next part discusses the dynamic version of the data structure in the powerful parallel disk head model. We then make a few changes to this data structure in order to obtain an even better static data structure that requires some fairly powerful error correcting codes. Finally we end this chapter by analyzing the data structure in the less powerful, and much more realistic parallel disk model.

The results obtained. We construct a static dictionary in the Parallel Disk Model using $O(n \cdot d \cdot \log d + n\beta)$ bits of space without the use of error correcting codes and one requiring $O(n \log u + n\beta)$ using some powerful error correcting codes, where β is the satellite data size of a single key. We also give dynamic dictionaries in the Parallel Disk Head Model and Parallel Disk Model both using $O(n \log n \log u + n\beta)$ bits. The dynamic dictionary for the Parallel Disk Head Model requires $O(1)$ I/Os amortized for insertions and $O(1)$ worst-case for deletions. The dynamic dictionary in the Parallel Disk

¹This approach was suggested by our supervisors

Model has the same deletion complexity while requiring expected $O(1)$ I/Os amortized for insertions. All dictionaries require only one parallel probe in order to perform a lookup.

3.1.1 Preliminaries

The parallel models of computation

In the following sections we will refer to the following two models of computation:

Definition 3.1 (Parallel k -Disk Head Model (PDHM)). *We define the PDHM as the standard external memory model, with the added ability to perform writes or reads to k blocks in one operation.*

Definition 3.2 (k -Parallel Disk Model (PDM)). *We define the PDM as the standard external memory model, equipped with k separate disks, where each operation allows one operation on each disk.*

We note that the PDM is by far the most realistic of the two models, as it is easily achievable in practice by means of hardware RAID controllers.

Block size The block size of a single disk is denoted B . We assume that $B \geq \log u$.

Satellite data The satellite data for a key x is denoted $\Delta(x)$ and has size β , that is, all satellite data is assumed to be of equal size.

The basic data structure. As previously we use an expander graph to map keys in U to locations in V . The basic idea in all the data structures in this chapter is for a key $x \in U$ to distribute its satellite data among $\Gamma_{asn}(x)$, and in a lookup retrieve all $\Gamma(x)$ in one probe. (It is assumed that we have internal memory for such d elements.)

ϵ in the parallel models When we retrieve all neighbors in every probe there is no dependence on ϵ in the time for lookup. In some of the data structures in this chapter it will however be necessary to set $\epsilon < \frac{1}{2}$ and requiring an expansion factor of $(1 - \epsilon/3)d$.

ϵ -ghosts in the parallel models An important observation is that in this parallel settings the concept of ϵ -ghost has no immediate relevance. The ghosts caused problems in the original OPS scheme because we by reading just one neighbor had to ensure both that we never gave a false negative answer and that we had $(1 - \epsilon)$ probability of answering correctly. In the parallel setting we will use two different approaches. For the static

data structures we will maintain that $(1 - \epsilon)d$ neighbors of a key that is not inserted contains \perp . In order to do this we simply utilize a $(2n + 1, d, (1 - \epsilon/2)d)$ expander exactly as in the original OPS scheme. Since we use a $(2n + 1, d, (1 - \epsilon/2)d)$ expander we know from Lemma 1.8 that there are at most n ϵ -ghosts, meaning that there will be expansion capacity enough for both the ϵ -ghosts and the inserted set, and hence for all keys $x \notin S$ at least an $(1 - \epsilon)$ fraction of the neighbors of x can be set to \perp . In the dynamic versions of the data structures we will not require any kind of assignment for keys that are not inserted.

3.1.2 Existence of good expander graphs

In previous sections we have assumed the existence of an expander graph with degree $O(\log \frac{u}{n})$, and we have also noted that a high degree adversely affected the space usage in the original OPS scheme. When we distribute satellite data over a key's neighbors, the scenario is slightly different as we wish for each part of the satellite data to fill approximately one disk block of space. A higher degree yields more neighbors, and so for very large satellite data it might be interesting to have a slightly higher degree.

We will now prove that expanders with degree $O(\log \frac{u}{n})$ and higher exist, and that they are very easy to construct probabilistically. We will furthermore prove that, at the same time, it is possible to achieve that the edges of any key can reside on different disks without incurring any further space cost. We require this property when we consider the PDM.

Definition 3.3. *An (n, d, α) -striped expander is an (n, d, α) -expander where for all $1 \leq i, j \leq d, i \neq j$ we have*

$$\left(\bigcup_{\forall x \in U} \Gamma_i(x) \right) \cap \left(\bigcup_{\forall x \in U} \Gamma_j(x) \right) = \emptyset$$

The Lemma below is a generalization of Lemma 4 in [OP02] and the proof is based on the same basic concept, extended to give a randomized construction that succeeds with very high probability and guarantee the striping property.

Lemma 3.4. *For $0 < \epsilon < 1$ and $d \geq 1$, $r^{\epsilon d} > 1$, if $|V| \geq (1 - \epsilon) dnr(2u/n)^{1/(\epsilon d)} e^{1/\epsilon}$ then there exists a striped $(n, d, (1 - \epsilon)d)$ expander as well as a corresponding randomized construction using time and space $O(ud)$ which has probability $1 - 1/r^{\epsilon d}$ of constructing an $(n, d, (1 - \epsilon)d)$ expander.*

Proof. The proof is a standard application of the probabilistic method. We will create a random graph in the following way: we partition V into d parts each containing $|V|/d$ elements. Each node in U then randomly chooses its k 'th edge into the k 'th part of V , $1 \leq k \leq d$.

We will now bound the probability that some subset of U of size $i \leq n$ has less than $(1 - \epsilon) di$ neighbors. A subset $S \subseteq U$ of size i can be chosen in $\binom{u}{i}$ ways. A set $V' \subseteq V$ of size $(1 - \epsilon) di$ can be chosen in $\binom{|V|}{(1-\epsilon)di}$ ways. Denote the maximum probability that a node chooses all its edges within such a set V' as p .

This gives us a bound on the probability that some subset of size $i \leq n$ has less than $(1 - \epsilon) di$ neighbors:

$$\sum_{i=1}^n \binom{u}{i} \binom{|V|}{(1-\epsilon)di} p^i$$

The partitioning of V' that gives the highest probability of a key's neighbors being contained in it, is one that has $|V'|/d$ nodes in each part of the d parts of V (if $|V'| < d$ it is impossible for the edges of a key to be contained in the set V'). Thus we have $p < \left(\frac{(1-\epsilon)di/d}{|V|/d}\right)^d = \left(\frac{(1-\epsilon)di}{|V|}\right)^d$. Inserting this into the above expression we get:

$$\begin{aligned} \sum_{i=1}^n \binom{u}{i} \binom{|V|}{(1-\epsilon)di} (p)^i &< \sum_{i=1}^n \binom{u}{i} \binom{|V|}{(1-\epsilon)di} \left(\left(\frac{(1-\epsilon)di}{|V|}\right)^d\right)^i \\ &= \sum_{i=1}^n \binom{u}{i} \binom{|V|}{(1-\epsilon)di} \left(\frac{(1-\epsilon)di}{|V|}\right)^{di} \\ &\leq \sum_{i=1}^n \left(\frac{ue}{i}\right)^i \left(\frac{|V|e}{(1-\epsilon)di}\right)^{(1-\epsilon)di} \left(\frac{(1-\epsilon)di}{|V|}\right)^{di} \\ &\leq \sum_{i=1}^n \left(\left(\frac{(1-\epsilon)di}{|V|}\right)^{\epsilon d} e^{d_u/i}\right)^i \end{aligned}$$

Inserting $|V| \geq (1 - \epsilon) rdn(2u/n)^{1/(\epsilon d)} e^{1/\epsilon}$ gives us

$$\begin{aligned} &\leq \sum_{i=1}^n \left(\left(\frac{(1-\epsilon)di}{(1-\epsilon)rdn(2u/n)^{1/(\epsilon d)}e^{1/\epsilon}}\right)^{\epsilon d} e^{d_u/i}\right)^i \\ &\leq \sum_{i=1}^n \left(\left(\frac{1}{r(2u/n)^{1/(\epsilon d)}e^{1/\epsilon}}\right)^{\epsilon d} e^{d_u/i}\right)^i \leq \sum_{i=1}^n \left(\left(\frac{1}{r^{\epsilon d} \cdot e^d \cdot 2u/n}\right) e^{d_u/i}\right)^i \\ &\leq \sum_{i=1}^n \left(\frac{1}{2r^{\epsilon d}}\right)^i \leq \sum_{i=1}^n \left(\left(1/r^{\epsilon d}\right) (1/2)^i\right) \leq (1/r^{\epsilon d}) \end{aligned}$$

□

Corollary 3.5. *For any finite universe U (we call $|U| = u$) and any $n \leq u$ and any constant ϵ , where $0 < \epsilon < 1$, there exists an $(n, \log u, (1 - \epsilon) \log u)$ -striped expander with $|V| = O(n \log u)$ and it can be constructed probabilistically in time $O(ud)$ with probability at least $1 - 1/u^\epsilon$*

Proof. Follows directly from Lemma 3.4 □

The construction suggested above does not in itself yield a suitable construction of the expanders we need, as the randomized construction will require time and space $\Omega(|U|d)$. It does however demonstrate that expanders exist with very low degree. Furthermore the randomized construction will become useful in Chapter 6 as a method for constructing small expander graphs that can be used as building blocks for larger expander graphs.

3.2 Simple static PDM dictionary

Basic structure By using $O(n \cdot d \cdot \log d + n\beta)$ bits of space we can build a static PDM dictionary and query it with one probe pr. lookup. We basically build the data structure as in the original OPS scheme just using the striped expander from lemma 3.4 instead of the normal expander. The main difference is that we do not write the key $x \in S$ in all its assigned neighbors. In this dictionary we fix ϵ at some constant smaller than $1/2$ ensuring that every key in S has over half of its neighbors assigned to it.

To build the dictionary we take x and concatenate it with its satellite data and distribute this final string among the first $(1 - \epsilon)d$ elements of $\Gamma_{asn}(x)$ in the ordered way of the Γ -function so that the first $\frac{\log u + \beta}{|(1 - \epsilon)d|}$ bits of the final string is placed in the first assigned neighbor, the second in the second and so forth. We then chain together all elements in an assignment neighborhood by placing a pointer in each one to the next (the tail pointing to the head). Every chain will have the same length since β is the same size for every key in S . We additionally mark the head of such a chain with a type bit (tail elements are equally easily spotted since they are the only ones that point to an earlier element).

Space usage The chain pointer will only point within a neighborhood and so we will only use $\log d$ bits on such a pointer. That is, the pointer stored in an element $\Gamma_i(x) \in \Gamma_{asn}(x)$ will be the next number j after i for which $\Gamma_j(x) \in \Gamma_{asn}(x)$. And since there is only d elements in $\Gamma(x)$, we use $\log d$ bits to point to a specific one.

So all in all we store every key in S once, the satellite data of every key in S once, and for all elements in V we store a $\log d$ bit chain pointer. The resulting space consumption is $O(n \log u + n\beta + nd \log d)$ bits.

Lookup When performing a lookup of a key $x \in U$ we can always quickly recognize if $x \notin S$ since we know that at least $(1 - \epsilon)d$ of the neighbors must be \perp . If there on the other hand is at least $(1 - \epsilon)d$ chain elements there might be more than one head element. In that case we check them all, that is, we follow their chain and see if it is exactly $(1 - \epsilon)d$ long and the concatenated appended data spells ' x ' with the first $\log u$ bits. The remaining problem is to distinguish between the right chain and 'ghost' chains when we have several chains with these characteristics.

Since a complete chain of $(1 - \epsilon)d$ elements takes up more than half of a neighborhood there can not be two disjoint chains in one neighborhood and so the only way for a ghost chain to appear is if it shares some elements with the right one. The ghost chain can only share the last part with the right chain since as soon as it meets an element on the right chain it will inevitably follow the right chain to the end. And here we finally reach the tail element which enables us to dismiss any ghost chains since we conveniently stored a

pointer to the correct assignment head.

Drawbacks of this dictionary Note that if $\beta = \Omega(d \log u)$ then we can afford to store the entire key in every assigned neighbor along with a part of the distributed data, without affecting the asymptotic space cost. The space saving trick described above only offers an asymptotic improvement for quite small β .

Furthermore if $\beta = o(d \log u)$ we have that every element in V stores data that is less than a word long and since we must assume that we can not have a block size of less than a word, we will have a poor use of bandwidth. That is, a read on one disk will retrieve many elements into internal memory, but we will only need one of them. All in all we will retrieve at least $d \log u$ bits but will only need $d \log d$ of them.

3.3 A dynamic PDHM dictionary

In this section we will consider a dynamic version of the one-probe data structure that performs well in the parallel disc head model (PDHM) assuming that we have d disk heads.

3.3.1 Basic structure

Our data structure contains the following:

- an array T_V containing $O(nd)$ entries. Each entry takes up $\log n + O\left(\frac{\beta}{(1-\epsilon)d}\right)$ bits up to one disk block. An entry is either vacant, or assigned to a neighboring element in S . In an entry assigned to $x \in S$ the first $\log n$ bits are used for storing a pointer $id_A(x)$ to the array A . The remainder of the entry will be used for storing a fraction of x 's satellite data.
- an array A of size n . Each entry in A is either vacant or contains an element in S and an integer denoting the number of elements in V that is assigned to x . As in the original OPS data structure we will maintain that for each $x \in S$, the number of entries in V assigned to x is at least $(1 - \epsilon)d$.
- an $(n, d, (1 - \epsilon/3)d)$ -expander graph where $0 < \epsilon < 1/2$, $d = \log u$.
- a stack F containing an index into each vacant entry in A .

3.3.2 Performing insertions

When performing an insertion of $x \in U$, we do the following: Probe all the neighbors of x in parallel, getting their id_A values (if they are not vacant, and

ignoring them otherwise). For each of these values i , in parallel decrement the assignment counter in $A(i)$ with the number of times the id_A occurred among the neighbors of x . If the counter of a key i falls below $(1 - \epsilon)d$ we place the key and its associated data $\Delta(i)$ on a separate stack Q on the disk. If the counter is already below $(1 - \epsilon)d$ we ignore the key as it will already have been added to Q .

Encode the satellite data of x into d packets such that any $(1 - \epsilon)d$ packets suffices to restore the data. Obtain a free id_A value by examining F . Replace all neighbors of x with $id_A(x)$ concatenated with one of the d packets obtained by the encoding step.

For each of keys i on the stack Q perform an insertion of i .

From Lemma 1.9 in section 1.4 we get that z insertions into the data structure already containing s elements will cause the assign-all step to occur $O(z + s)$ times, giving us an amortized $O(1)$ parallel I/O complexity for insertions.

A very important thing to observe is that it is critical that we can discover the owner of an assigned element. If we were unable to do this we would be helpless when it came to discovering keys with too low an assignment.

3.3.3 Performing lookups

In order to perform a lookup of $x \in U$ in the above data structure, we perform a parallel probe reading each of the neighbors of x . It is not immediately clear what parts of the retrieved data belongs to x , as x itself is not part of the data blocks. Fortunately each of the retrieved blocks of data contains $id_A(x)$, which we can use in the following way:

- If $x \in S$ we can identify our data as the blocks marked by the id_A that occurs in more than half of the blocks (since we chose $\epsilon < 1/2$).

In order to restore the original data, we decode $\Gamma(x)$. Since we know that $|\Gamma_{asn}(x)| \geq (1 - \epsilon)d$ this is always possible.

- If $x \notin S$ no single id_A belonging to a key $y \in S$ occurs in more than half of the blocks and so we know that $x \notin S$. If such an $id_A(y)$ value occurred in more than half the blocks then $|\Gamma(x) \cup \Gamma(y)| \leq d \cdot 3/2$ and since we have an expansion factor of $(1 - \epsilon/3)d < d \cdot 5/6$ two keys should have at least $(1 - \epsilon)d > d \cdot 10/6$ neighbors, so it can not happen.

3.3.4 Performing deletions

In performing a deletion of x , we obtain $id_A(x)$ by performing a probe for x . We then proceed to remove the blocks from V that is assigned to x .

Furthermore we clear the entry $A(id_A(x))$ and add the now available id_A to F . This can obviously be achieved in $O(1)$ parallel I/O's.

3.3.5 Requirements for coding

In order to distribute the satellite data over all the neighbors of a key we would like a coding scheme that allows us to take b bits and create d packets each containing $O\left(\frac{b}{(1-\epsilon)d}\right)$ bits such that given $(1-\epsilon)d$ of these packets it is possible to reconstruct the original bit string. We will consider two types of such codes, one type only supports erasure correction, so that invalid packets should not be used as input for the decoder but rather identified before hand. The other type supports error correction, such that if enough of the given packets are valid it can still decode even if corrupted packets occur.

We will denote the original bit string M (in our case some associated data), and use m to denote its length. We will let k denote the number of packets that should be sufficient to restore the message, and x the desired number of packets in total.

3.3.6 Reed Solomon codes

Reed Solomon coding[RS60] encodes messages in the form of a polynomial on a field. Given a message bit string M , it is divided into k evenly sized strings. Each of these strings are interpreted as an integer coefficient in a k -order polynomial. Given the evaluation of this polynomial at k different points it will be possible to deduce the original polynomial (the details will not be discussed here), and thus the original message. The Reed Solomon coded packets will simply be the values of this polynomial at x different points. In order to work with an integer field, all arithmetic is done modulo a prime number p . In order to support coefficients of size m/k it must be the case that $\log p > m/k$. And in order for the field to contain enough distinct points to evaluate at, it must be the case that $p > x$ (and therefore $\log p > \log x$).

Reed Solomon codes can handle both erasure and error-correction. If it is necessary to do error correction, only $(x-k)/2$ packets may be invalid, as opposed to the ability to handle $x-k$ erasures.

Packet sizes

On certain parameters the Reed Solomon encoding results in packets that are larger than we could hope for. Consider the case that the $\beta \leq c \log u$ and where $d = O(\log u)$, such that $m = cd$ and $k = (1-\epsilon)d$. In this case we would ideally like d packets of size $O\left(\frac{c}{1-\epsilon}\right)$, unfortunately the Reed Solomon encoding will produce larger packets. We have that $\log p > \log d$

and $\log p > m/k = \frac{cd}{(1-\epsilon)d}$. The first of these restrictions yield a $p > d$ dependence. For large β however, the first restriction is not an issue and the second restriction gives us $\log p > \frac{\beta}{(1-\epsilon)d}$, so that we can use the first prime number whose logarithm is larger than the ideal packet size.

3.3.7 Pure erasure codes

The approach in [AL96] is to relax our expectations on how much coded data is needed to recover the original data and require that corrupted data is discarded before given to the decoder. Optimally any number of packets whose combined size equals the original data should be enough to decode. In [AL96] the requirement is that $1 + \lambda$ times the original data must be available, where $\lambda > 0$ is a constant. In return it has $O(m/\lambda^4)$ encoding and decoding time, and allows packets of a constant number of letters over an alphabet of constant size (these constants are however growing proportional to $\frac{1}{\lambda^4} \log \text{frac}1\lambda$ and $\log \text{frac}1\lambda$ respectively). A further assumption of this code is that each packet is given a unique index number as the order of packets is important. In our case we do not need to store this index as it is implicit in the edge number of the neighbor storing the packet and we know from our lookup algorithm that we will only attempt to decode blocks that we have read from the correct key.

In our case we want to store β bits in d packets so that $(1 - \epsilon) d$ packets suffice to restore it. Denote the packet size by l . In order for the packets from assigned entries in V to contain enough data it must be the case that:

$$(1 + \lambda) \frac{\beta}{l} \leq (1 - \epsilon) d$$

We would like to have $l = \frac{\beta}{(1-\epsilon)d}$ but due to the suboptimality of required data for decoding we instead set $l = (1 + \lambda) \frac{\beta}{(1-\epsilon)d}$, where $0 > \lambda > 1$, so that the same number of packets will provide more encoded data. Increasing the λ parameter decreases the time complexity for decoding and encoding, but decreases space efficiency as well as increasing the required block size of each I/O (or the number of disk heads required) as each entry in V takes up more space.

3.3.8 Time complexity implications of using erasure codes

All known algorithms for encoding data into Reed Solomon codes as well as decoding runs in super linear time of the data length. Since we have discovered that in the dynamic setting we must be able to identify the owner of coded blocks, and thus can disregard blocks belonging to other keys (“corrupted” in the coding sense), we have no real use of the error correcting properties of Reed Solomon codes. The pure erasure code scheme provides

linear time decoding, but with a high constant that depends on the space efficiency requested.

3.3.9 Space consumption

Using an $(n, d, (1 - \frac{\epsilon}{3})d)$ expander with $O(dn)$ entries in V each using $\log n + C \frac{\beta}{(1-\epsilon)d}$, for a total of $nd \left(\log n + C \frac{\beta}{(1-\epsilon)d} \right)$ bits, where $C = 1^\dagger$ if we use Reed Solomon coding, and $C > 1$ for pure erasure codes.

Furthermore we have the array A using $n(\log n + \log u)$ bits and the stack F using $n \log n$ bits. Thus the entire data structure uses $O \left(nd \left(\log n + C \frac{\beta}{(1-\epsilon)d} \right) + n \log n + n \log u \right)$ bits.

In order to maintain that each entry in V takes up at most one disk block we need the following to hold:

$$\log n + C \frac{\beta}{(1-\epsilon)d} \leq B$$

Using the expander from Corollary 3.5 we get the following space use:

$$\begin{aligned} & O \left(n \left(\log u \log n + C \frac{\beta}{(1-\epsilon)} \right) + n \log n + n \log u \right) \\ & = O \left(n \left(\log u \log n + C \frac{\beta}{(1-\epsilon)} \right) \right) \end{aligned}$$

Observe that if $\beta = \Omega(\log u \cdot \log n)$ then we use the same space as would be used by storing the dictionary as a list of key-value pairs.

[†]Ignoring the difference between $2^{m/k}$ and p

3.4 A $O(n(\log u + \beta))$ bits static PDM dictionary

We have previously given a very simple PDM static dictionary data structure using $O(nd \log d + n\beta)$ bits of space. We can however improve the space usage beyond that by some small changes to the PDHM dynamic dictionary. Observe that in the static case the dynamic dictionary can also operate under the PDM as a striped expander can be used to ensure that each probed neighbor resides on a separate disks, just as in the previous static dictionary.

Element ownership In the dynamic data structure we were forced to mark the data in each element of V in such a way that we could identify which key it is assigned to (we denoted this $id_A(x)$) by reading a single node, giving us a slightly suboptimal space usage for small satellite data. Since we in a static dictionary do not support insertions, we will remove this mark.

Saving space We utilize a $(2n + 1, d, (1 - \epsilon/2)d)$ -expander, where $\epsilon < 1/2$. Let A_S be the $(1 - \epsilon)$ -balanced assignment that we know exists in the expander by Lemma 1.6. We will for each key, code itself along with its satellite data into d packets, such that $(1 - \epsilon)d$ of them suffices to restore the data, and then place in (at least) $(1 - \epsilon)d$ of them, according to A_S , a type bit indicating that this entry contains data.

In all other entries of V we will use the type bit to indicate that no data is present (we will denote this value of the type bit as \perp). Since we use a $(2n + 1, d, (1 - \epsilon/2)d)$ expander we know (see 3.1.1) that for all keys $x \notin S$ the type bit will be set to \perp in at least an $(1 - \epsilon)$ fraction of the neighbors of x .

Choice of coding scheme We can no longer use the pure erasure codes since we can no longer identify which packets are our own, and Reed-Solomon codes will for small satellite data provide packets that are larger than constant. Instead we resort to the coding scheme of [GI02] which extends on the previously mentioned erasure codes of [AL96], providing similar performance also for error correction. Specifically it provides error correcting codes for a fraction $(1 - r - \kappa)/2$ of errors, where $\kappa > 0$ and where $0 < r < 1$ denotes the rate of the code (that is the ratio between symbols in the plain-text message and the number of symbols in the coded message) using a constant sized alphabet for the output symbols. These codes have linear time encoding and decoding. The drawback compared to the erasure codes is that the alphabet, though constant, depends exponentially on $1/\kappa$, meaning that the number of redundant bits is significantly higher. We will not give further details on these codes, as we only use these codes in this special case of a static dictionary with small satellite data.

Lookups Let us now consider a lookup of a key $x \in S$. Such a key will have at least $(1-\epsilon)d$ valid packets belonging to itself along with \perp marked entries or entries belonging to other keys. This means we can always successfully decode the packets and validate that the resulting entry belongs to x . We have included the key itself in the encoded string since the coding scheme used might also be able to decode the entries of another key from a very small fraction of packets in some very rare cases, giving us multiple possible results. If the coding scheme ensures that this can not occur, we can leave out the key from the encoded data, but the type bit in each entry of V will still give the space usage $O(nd)$ bits.

Now consider a key $x \notin S$. Such a key will have at least $(1-\epsilon)d$ neighbors with the type bit set to \perp , and since $\epsilon < 1/2$ a majority of neighbors contains \perp and hence we know that the key in question is not inserted.

3.5 Dynamic PDM dictionary

In this section we will consider the possibilities for making a dynamic one probe data structure that performs well in the parallel disc model.

3.5.1 Basic structure

Our data structure is almost the same as the one presented in the previous section. To preserve the performance in the PDM some changes will be appropriate. Instead of using a normal expander we use a *striped expander* as per Definition 3.3. This way we can be sure that we can access all neighbors of a key in a single I/O since each of the neighbors reside on a separate disk.

In order to (attempt to) preserve the constant amortized insertion time we also distribute the stack Q of elements that needs reassigning to the d disks using simple buffering and striping. This also preserves the previous performance.

So far we can grab all neighbors in a constant number of I/O's, and we can efficiently stack nodes for reassignment, but we still need to read the assignment size of the keys who's assigned elements we intend to 'steal'. This requires checking at most d entries in A in the worst case. In order to handle this step in constant time, one might try to distribute A among d discs in such a way that the operations would only involve a constant number of operations on a single disc. This is trivially possible as long as $|A| = O(B \cdot d)$ since each disc can store $1/d$ of A one each disc, while only using a constant number of blocks on each disc. However, if $|A| = \omega(B \cdot d)$ we will no longer be able to store $|A|$ using a constant number of blocks on each disc.

Due to to the 'random' nature of an expander graphs, it seems unlikely that it is possible to give any useful restrictions on which access pattern an insertion can cause. Without such restrictions, our problem of accessing A seems just as hard as the more general problem of emulating the PDHM on the PDM. This is why we turn to a result on emulating the *PDHM* on the *PDM* introduced in [SEK00] in order to efficiently access A during insertions.

3.5.2 Emulation of PDHM by PDM

Before we apply the method on our algorithm we briefly sketch how this emulation takes place.

Writing We use d queues in internal memory to contain the data waiting to be written to each of the d discs respectively. When storing a disc block we choose 2 of the d queues at random, and store the entire block in both

of the queues. In picking the two queues at random, we have to either use a randomly chosen hash function or generate the random assignments ad hoc, storing the assignments in internal memory and thereby using $2 \log d$ bits of internal memory per disc block stored. We can not allow an internal memory space usage that depends on the number of blocks in use so we instead use a hash function that we assume is chosen randomly independent of the keys inserted. The results in the following relies on truly random choices for placing the blocks, so strictly speaking they only apply under the assumption that the random hash function is x -wise independent, where x is the number of blocks written.

When the memory used for the write cache is all used, one block from each of the non-empty queues are written to the disc, using one parallel I/O and freeing between B and dB bits of internal memory.

Lemma 3.6 (Restated Theorem 1 from [SEK00]). *Assume that each write request can provide up to $(1 - \kappa)d/2$ blocks for writing and consider a buffer size of $(\ln 2 + \lambda)d/\kappa$ for some constant $\lambda > 0$ and let w_t be the number of times all the write buffers write a block to disk during the t 'th write request. Then $E(w_t) \leq 1 + e^{-\Omega(d)}$*

According to this theorem the number of blocks that can be written, in expectation per parallel I/O used, grows to $(1 - \kappa)d/2$ as d increases, if we use $O(d \cdot B/\kappa)$ bits of internal memory as our cache.

Reading The emulation relies on the randomized duplication of the write operation in order to provide efficient read operations. A randomized specialized version of the max-flow algorithm is used to find which of the two possible locations of each element to read and the order of the read operations. The paper proves the following theorem:

Lemma 3.7 (Restated Theorem 4 from [SEK00]). *Consider a batch of x randomly and dublicately allocated blocks to be read from d disks. Then the number of parallel reads required is at most $\lceil x/d \rceil + 1$ with probability at least $1 - O(1/d)^{\lceil x/d \rceil + 1}$.*

PDHM emulation

Definition 3.8. *Let $PDHM_{d,B,M}(r,w)$ denote the set of problems solvable in the PDHM with d disks, block size B and internal memory M , using r parallel reads and w parallel writes. Let $EPDM_{d,B,M}$ denote the set of problems solvable in the PDM with d disks, block size B and internal memory M , using r parallel reads expected and w parallel writes expected.*

By combining the read and write results the following theorem is obtained in [SEK00]:

Theorem 3.9 (Restated Corollary 22 from [SEK00]). *Using Definition 3.8 we have that for any $0 < \kappa < 1$ and $b \in \mathbb{N}$,*

$$PDHM_{bd,B,M}(r, w) \subseteq EPDM_{d,B,M+O(d/\kappa+bd)}(r(b+1), bw)$$

3.5.3 Storing A using the emulator

We store the array A through the emulator. The emulator will store each block twice and it is up to the emulator to retrieve requested blocks efficiently.

Whenever we need to read an entry in A we, based on the index required, determine the block number, and request this block to be read from the emulator. When we perform updates to A we first read the block as above, modify the contents and request it to be written back.

3.5.4 I/O complexity of insertion

Insertions are made in the same way as in section 3.3, except when it comes to A . The insertion algorithm needs to perform $O(n)$ reads and writes of d blocks from A when inserting n elements, as described earlier. We could handle this with $O(n)$ I/O's in the PDHM. By Theorem 3.9 these operations can be performed in expected $O(n)$ I/O's on the PDM with appropriate choice of parameters for the emulator.

3.5.5 I/O complexity of deletion

Deleting an element, only requires a single access to A , in order to delete the entry belonging to the deleted key. This can be done in constant worst case time as there can be no access conflicts within the emulator when asked to write a single block.

3.6 Conclusion

In this chapter we have introduced four different parallel data structures. The first was a static dictionary data structure for small satellite data for the d -disk PDM, which using $O(nd \log d + n\beta)$ bits of space, supports lookups in a single parallel lookup. We then constructed a dynamic dictionary data structure for use with the d -disk PDHM, using $O(nd \log d + n\beta)$ space, while still supporting lookups in a single parallel probe and also insertions in amortized $O(1)$ I/Os and deletions in $O(1)$ I/O. We then noted that using a different error correction code it was possible to obtain a static PDM dictionary using $O(n \log u + n\beta)$ bits of space while still only needing one probe per lookup. Finally we adapted the dynamic data structure for use with the PDM, maintaining the lookup complexity, deletion complexity and space usage, while achieving $O(1)$ expected amortized I/O's for insertion by using a known result on emulating the PDHM on the PDM. We noted

this reduction in insertion performance was caused by a specific bottleneck related to determining the assignment size of other keys during the insertion. Furthermore we have seen that the expander graphs required to implement these data structures exists with good parameters, in such a way that the right node set can be split into d buckets such that the i 'th neighbor of a key will always reside in bucket i . In doing that we have observed that a randomized construction of such an expander is possible with very high probability of success.

Chapter 4

Two Probe schemes

In this chapter we investigate the implications of allowing two probes instead of just one. In particular we are interested in reducing the space usage compared to the original OPS data structure. The question of whether the space usage of this structure could be reduced by using two random probes instead of one was raised in [OP02]. We consider two schemes that uses one random probe followed by one deterministic probe. We show that significant space can be saved in the static case and give a dynamic data structure that does save some space, but is hampered by a fundamental issue regarding dynamic reassignments. We do not explore whether making the second probe random could offer additional savings.

4.1 Static Two Probe scheme using $O(n \log \log u)$ words

We will start by considering the static case for a two probe scheme. We will see that it is indeed possible to reduce the space usage to $O(n \log d)$ words, assuming the existence of an optimal explicit expander construction, as compared to $O(nd)$ words for the original one probe data structure.

4.1.1 Versus Perfect Hashing

We note that a static two probe scheme with this space usage is not in itself very interesting, as it has to compete with Perfect Hashing [Spr77] which can be used to provide a dictionary where two probes to the description of the hash function and one to the table suffice for lookups while only using $O(n)$ words of space [Pag99]. While this is an extra probe compared to the schemes considered in this section the first of these probes are to a fixed location, which means that it can be cached for later lookups. It is however still theoretically interesting that a randomized two probe scheme can be achieved in close to linear space, assuming the existence of an optimal

explicit expander construction.

4.1.2 The idea

A natural idea for achieving the space usage mentioned above is to choose one assigned neighbor for each inserted key x and store the full entry for x there. In all other assigned neighbors we can instead write a number indicating, seen from x , which of its neighbors has the full entry. This idea entails a problem: In order to save space using this approach we need the entries in V to be of variable size while still being accessible in a single probe. This is unfortunately not possible, though $O(1)$ access can be achieved.

4.1.3 Solving the variable size issue

To solve this problem, we instead rely on two expanders G_1 and G_2 . We will let G_1 be a $(2n + 1, d, (1 - \epsilon)d)$ -expander, while G_2 will be a $(n, d, 1)$ -expander. The critical observation here is that the right side of G_2 , denoted $|V_2|$ can be $O(n)$. We show this fact in the following lemma

Lemma 4.1 (Scaled expansion). *For $\alpha < d/4$, there exists a striped (n, d, α) -expander $U \times D \rightarrow V$ with $|V| = O(\alpha n)$.*

Proof. We will use Lemma 3.4. Choosing $\epsilon = 1 - \frac{\alpha}{d}$ we have $\alpha = (1 - \epsilon)d$. According to the lemma we just need a right side of size $\alpha n (2u/n)^{1/((1-\alpha/d)d)} e^{1/(1-\alpha/d)}$ which is $\alpha n \cdot O(1)$ since $\alpha/d < 1/4$ and $d = \Omega(\log \frac{u}{n})$. \square

Neighbors of a key x in G_1 will contain $\log d$ bits indicating which neighbor of x in G_2 contains the full entry of x . Assignments in G_1 will be as in the original data structure, guaranteeing proper answers for both keys that are inserted and those that are not. The only assignments in G_2 will be to keys from $S \cup \bar{S}$. The space used is $O(nd \log d + n \log u)$ bits. Assuming $d < \log u$ and a word size of $\Omega(\log u)$ bits we achieve the desired space.

The lookup function will do as follows. It will probe a random neighbor in G_1 , if this neighbor does not contain the special symbol \perp , it will probe the indicated neighbor in G_2 . For inserted keys and ϵ -ghosts probing one of their assigned neighbors, this obviously works. The same goes for the remaining keys if they choose one of their $(1 - \epsilon)d$ neighbors containing \perp . If the probe chooses an unfortunate position in V to probe, it might read an edge number meant to be read from another position. In this case it will probe a position in G_2 that is either vacant or contains a key from $S \cup \bar{S}$. In either case we will answer “don’t know”. Hence the properties of the lookup function from the original structure is preserved apart from spending an extra probe. As previously observe that we don’t need to probe the edge function of the expander as they are determined from just using the size of the set of inserted keys, as opposed the hashing based approach where the hash function changes based on the specific keys inserted.

4.1.4 Identifying assignment owners in the dynamic case

If we attempt to use the above technique in a dynamic scenario, we encounter a fundamental problem. When a new key arrives we need to assign some keys from V_1 to it in G_1 . It might be the case that this can only be done by stealing elements from other inserted keys. In the data structure described above we have no efficient means of determining who owns an element of V_1 , unless we are probing the element from the key that owns it, and so we have no way of repairing an assignment that is reduced below the allowed size. The only way to determine such ownership is to scan V_2 , and probe all neighbors of each key found there, verifying that following the edge number found in the neighbor indicates the correct key in G_2 . Even this might not work correctly though, since an element in V_1 belonging to the key x might accidentally also give the correct edge number for the key y . This can be remedied by listing in V_2 , in each keys entry, what edge indices their key owns among their neighbors in G_1 , not using asymptotically more space. This requires $O(n)$ time to obtain the reverse mapping formation. It is natural to instead want to store this mapping, but unfortunately this implies storing the ownership of the $O(nd)$ keys in V_1 . It is possible to store this information using $O(n \log n \log \frac{u}{n})$ bits of space, but this means we have to use more space than in the above static data structure. Using less space for this backwards mapping seems information theoretically impossible.

4.2 Dynamic Two Probe scheme using $O(n \log n)$ words

In the previous section we discovered that it seems essential to be able to discover which keys a right side node is currently assigned to. In this section we will briefly cover a simple trick that allows us to use $O(n \log n)$ words of space instead of the $O(n \log \frac{u}{n})$ word used by the original OPS data structure, while still making the reverse mapping of assignments efficiently available. We will first see how to obtain this space consumption in a static data structure allowing for the reverse mapping of assignments, then we proceed to explain the changes required to obtain the dynamic data structure.

4.2.1 Static data structure with reverse mapping

The original OPS data structure uses $O(n \log \frac{u}{n})$ words of space because it needs to store each $x \in S \cup \tilde{S}$ up to d times in the table. The basic idea is now to replace the many full word size entries stored in the table T_V , representing V , with indexes to a table T_u with $O(n)$ entries, simply enumerating the currently used symbols. If satellite data is used, it will be stored alongside the symbol itself in this table.

Obviously we can get away with using $O(\log n)$ bits for storing an index into this table.

Now the modified lookup algorithm simply chooses a neighbor, reads the index into T_u from T_V , and then reads the real value in T_u .

Space requirements

Instead of having one word per entry in the table T_V we can now fit $O(\lfloor \log u / \log n \rfloor)$ entries into a single word. The table T_u consists of $O(n)$ words, while the table T_V with $O(n \log \frac{u}{n})$ entries now take up

$$O\left(\frac{n \log \frac{u}{n}}{\lfloor \log u / \log n \rfloor}\right) = O(n \log n) \text{ words}$$

4.2.2 Dynamic data structure

The dynamic data structure is only slightly more complicated:

- We now need to dynamically manage the enumeration of elements under insertion and deletion
- The table T_V now also stores a pointer to the table holding the assignment size counters, as well as a counter keeping the number of keys in S , that has the current element as their neighbor.

Pointers and counter We can easily handle the pointers by adding one extra column to the enumeration table, since there can at most be $O(\log n)$ different pointers which are functionally dependent on the key, where n as previously is the maximum number of keys that can be inserted. For the counter we use $\log n$ bits, since the highest possible value of it is n .

Dynamic enumeration Handling the enumeration dynamically can easily be done in constant amortized time. When inserting new symbols we simply add them onto the end of the table, assigning the next available index to them. When inserted keys are deleted we also mark them as deleted in T_u . Since we are working in a table, we only delete and collapse the table for each n operations in order to keep a constant amortized time per update.

Insertion Insertion is not very different than in the original OPS data structure. We will just additionally allocate an index to the key and write this in the assigned neighbors. If insertion of already inserted keys should be supported, we have to in each insertion first check all neighbors to see if the key is inserted, otherwise it could cause trouble to just allocate a new index to the key and then grab all the neighbors. The algorithm would then think that the key of the old index has gone below the allowed assignment size.

4.2.3 Conclusion

In this chapter we have described two schemes, that uses less space than the original OPS data structure by having two probes available. A static and a dynamic. The main idea in both schemes are that we can save space by only storing each data element once and making the other elements contain a pointer to the data.

- The static data structure uses $O(n \log d)$ words of space.
- The dynamic data structure uses $O(n \log n)$ words of space.

Both schemes save space relative to the $O(n \log(\frac{d}{n}))$ words used by the original OPS scheme.

Chapter 5

Worst Case Insertion

5.1 Introduction

We have seen previously how to construct a data structure that uses a single parallel operation to look up an element and its data. In order to support insertions of new data we relied on an insertion algorithm for which only an amortized bound on the insertion time has been proven. In this section we will discuss the possibility of an insertion algorithm that has a good expected running time in the worst case, the search for such an algorithm is inspired by two main issues:

1. The parallel data structure, as well as the data structure covered in 1.4, are mostly of interest in time-critical applications, and therefore it would greatly increase their range of possible uses if it was possible to produce an insertion algorithm which had a good worst case bound. In this regard we would also like to know how the current insertion algorithm handles in the worst case.
2. The current insertion algorithm caused a bottleneck when operating in the Parallel Disk Model, an alternate insertion algorithm might solve this problem.

In order to examine this problem, we will start by relating the problem of adjusting assignments in the expander graph, to that of finding augmenting paths in a subgraph of the expander graph. We then move on to discuss how such augmenting paths can be structured in the subgraph, and analyzing the possibilities of discovering such an augmenting path through a random walk.

5.1.1 Preliminaries

Unless otherwise stated, we discuss the insertion algorithm in the context of the original OPS data structure. In the end of the chapter we relate the

discussed topics to the parallel disk model. In the following definitions and analysis we as usual consider using assignments of size at least $(1 - \epsilon)d$ in an expander graph $G = (U, V, E)$ with the set of inserted nodes being S and left degree d .

Vacant elements A *vacant element* is an element of V that is not currently assigned to a key.

Available elements An *available element* is an element of V that is either vacant or is assigned to a key $a \in S \cup \bar{S}$ where $|\Gamma_{asn}(a)| > (1 - \epsilon)d$. Such an element is termed 'available' because it is possible to assign it to another key, without causing any keys assignment size to decrease below the required level. Observe however that available elements can be dependent on each other, meaning that if you use one, the others might have become unavailable. An example of this is a key a with an assignment size of $(1 - \epsilon)d + 1$. Among the neighbors of a , at least $(1 - \epsilon)d + 1$ are available elements, but using just one of them, causes $(1 - \epsilon)d$ of the remaining available elements to no longer be available.

Occupied key A key in $x \in S$ is an *occupied key* if no available elements are assigned to x i.e. if $|\Gamma_{asn}(x)| = (1 - \epsilon)d$. A key that is not occupied is called an *unoccupied key*

5.2 Augmenting paths

5.2.1 Introduction

We will in this section first in short terms describe the classical matching problem and restate the known method of using augmenting paths to solve it. We will then define the special subgraph where the flipping of augmenting paths will solve our assignment problem and show this correspondence in detail.

Then we will give a proof that the changes done in the assignment by any insertion algorithm can be lower bounded by flipping of some augmenting paths. This could be used to make a lower bound on the insertion problem. If we could show that the shortest augmenting path for some key needing its assignment to be increased is x long, we have a lower bound of x on the insertion problem. Furthermore it shows that trying to find augmenting paths could be an alternative to the approach in 1.4.3 to solve the assignment problem.

And last is a proof that any key in $S \cup \bar{S}$ has an augmenting path of length at most $O(\log n)$.

Figure 5.1: A graph with a matching (solid edges) and an augmenting path (a, b, c, d, e, f) for the node a

5.2.2 Matching and augmenting paths

The classical matching problem A matching in a bipartite graph $G(L, R, E)$ is a partitioning of edges as either matched or unmatched so that nodes in L are matched, i.e connected by a matched edge, to at most 1 node in R and vice versa. The size of a matching is the number of matched nodes. A matching problem could be to maximize the number of matched nodes or to solve problems of the kind: 'does there exist a matching of size x in this graph?'. A perfect matching is one where all nodes are matched and a left or right perfect matching is one where all nodes of L or R are matched respectively.

An augmenting path in a graph with some matching is a path consisting of alternating unmatched and matched edges, beginning and ending with an unmatched edge, i.e the first and last nodes have no matching, but all the other nodes on the path have a matching. When we have found the path, we would want to flip it, that is, to make all unmatched edges matched and all matched edge unmatched and so achieve that the overall number of matched nodes increase by 2. Notice that if an augmenting path has a cycle we can just cut it of and we have a shorter augmenting path with the same effect on the size of the matching. For further details see Chapter 26 in [CSRL01], or another textbook with material on max-flow and bipartite matchings.

Generalization A slightly generalized definition of the matching problem is as an (a, b) -matching problem, where a and b are the number of matched edges a node in the matching from respectively L and R must have for the matching to be valid. In this way the earlier described matching problem is a $(1, 1)$ -matching problem. We can still use augmenting paths to increase an (a, b) -matching, only we have to find and flip a augmenting paths for some unmatched node in L having no R nodes matched to it, and b paths for one in R . And the definition of an augmenting path need to include the point that it must begin with an unmatched edge, else the result will be to decrease the number of matched edges by 1. If all the paths are found before they are flipped they need to be edge disjoint, but more on this later.

5.2.3 Matchings in the induced assignment subgraph

An induced assignment subgraph involves the keys of $S \cup \bar{S}$ and their assigned neighbors. Recall that an assignment is defined as a set of directed edges, going from the assigned element in V to the set $S \cup \bar{S}$ of inserted elements and ϵ -ghosts.

Definition 5.1 (Induced assignment subgraph). *Let $G(U, V, E)$ be an $(n, d, (1 - \epsilon)d)$ -expander, let $S \subseteq U$ where $|S| \leq n$ and let A be a valid $(1 - \epsilon)$ -balanced assignment for $S \cup \bar{S}$ in G . Let E_S be the edge set of directed edges $\{(a, b) \mid a \in S \cup \bar{S}, b \in \Gamma(a)\} \subset E$. Then the directed graph $G'(S \cup \bar{S}, \Gamma(S \cup \bar{S}), E_S \cup A)$ is an (S, A) -induced assignment subgraph of G .*

Observe that there can be more than one induced assignment subgraph of some graph G for some inserted set S , since the same set can have several different valid assignments, but for a specific set S and a specific assignment A there can only be one induced assignment subgraph. This means that in an OPS data structure there is always exactly one present induced assignment subgraph.

Another crucial point, that will become quite important later, is that although this graph is expanding as the mother graph G from left-to-right, all elements in V only have out-degree 1 (and the right-to-left 'expansion' factor is $\frac{1}{(1-\epsilon)d}$).

From induced assignment subgraph to matchings An induced assignment subgraph with a valid $(1 - \epsilon)$ -balanced assignment is a graph with a $((1 - \epsilon)d, 1)$ left perfect matching if we look at it the following way. All edges $(a, b) \in S \cup \bar{S} \times V$, are unmatched and all edges from assigned elements, i.e. all other edges, are matched. Notice that since all edges from V are matched edges all paths in an induced assignment subgraph beginning in a key in $S \cup \bar{S}$ and ending in an element of V is an augmenting path. Furthermore if the path ends in an available element we have increased the assignment size of the key we started out with by one.

We know that there exists a $((1 - \epsilon)d, 1)$ left perfect matching in the graph by Lemma 1.6, the problem is to find it.

Assigning one element by using an augmenting path If we focus on increasing $|\Gamma_{asn}(a_1)|$ by one for a key $a_1 \in S \cup \bar{S}$ our problem (the 1-assignment problem), is the problem of finding a matching for a_1 without decreasing the assignment size below the allowed for any key in $S \cup \bar{S}$.

Now suppose we have some augmenting path $a_1, b_1, a_2, b_2, \dots, a_p, b_p$ from the key a_1 to the available element b_p . We know that all a_i -nodes are keys in $S \cup \bar{S}$, that all b_i -nodes are elements of V , and that b_p is available. This path consists of alternating edges from $S \cup \bar{S}$ to V and from the present assignment. Flipping this path will change the present assignment so that

Figure 5.2: An induced assignment subgraph with a $(\frac{2}{3})$ -balanced assignment, except for a_1 having a too small assignment.

White nodes are available elements of V . Solid lines represent two edges. (1) An unmatched edge pointing left to right, i.e. equivalent to a dotted edge. (2) A matched edge from right to left.

The path $(a_1, b_1, a_2, b_2, a_3, b_3)$ is an augmenting path for a_1

all b_i will be re-assigned to a_i instead of their previous assignment to a_{i+1} . So all the a_i 's have the same assignment size as before except a_1 having 1 extra element assigned to it.

5.2.4 Generality and length of augmenting paths

Augmenting paths as general solution Any algorithm A solving our matching problem does it by flipping edges from assigned to unassigned or vice versa. Let us denote the edges that A would have changed as C . Then some subset of C (it might be all of C) is an augmenting path solving the very same problem. Of course this does not mean that solving the problem by any augmenting path is always the shortest in terms of edges flipped, but just that there exists an augmenting path bounding the number of flips we have to do from below.

Lemma 5.2. *Let $G(U, V, E)$ be a graph in some OPS data structure with the inserted set S and the valid $(1 - \epsilon)$ -balanced assignment $A \subset E$ except for the key $x \in S \cup \bar{S}$ missing one assignment. Let ALGO be any algorithm solving the problem of producing a $(1 - \epsilon)$ -balanced assignment B for $S \cup \bar{S}$. Let $C = (A \setminus B) \cup (B \setminus A)$. Then there exists some $D \subseteq C$ forming an augmenting path for x .*

Proof. Notice that it is not necessarily so that $C \subseteq A$. ALGO can have changed many assignments and for instance increased the assignment of other keys than x . The proof is by simple iterative argument. We iteratively build a valid augmenting path for $p_1 = x$ by including 2 edges from C in each

step, one from $S \cup \bar{S}$ to V (unmatched) and one from V to $S \cup \bar{S}$ (matched). We have 3 cases. Let C_j be the set of edges $(p_1, p_2), (p_2, p_3) \dots (p_{j-1}, p_j)$. We define $C_j = \emptyset$.

Case 1: In $C \setminus C_i$ we have an edge from p_i to some available neighbor. We are through, we now have an augmenting path for p_1 .

Case 2: In $C \setminus C_i$ we have an unmatched edge from p_i to a non-free element p_{i+1} of V . We denote by x the key p_{i+1} is assigned to. We then know that there must be an edge (p_{i+1}, x) in $C \setminus C_{i+1}$. We will see this in the following. Since ALGO have made the previously unmatched edge (p_i, p_{i+1}) matched, p_{i+1} is now assigned to 2 elements, x and p_i . The only way for ALGO to produce a valid assignment is if it also makes (p_{i+1}, x) unmatched. In the next iteration x is then the new p_i .

Case 3: In C_i we have no unmatched edge from p_i to an unavailable element of V . We know that p_i has a non-valid assignment if only edges from C_i are flipped since we would else have landed in case 1 in the previous iteration. If ALGO decided to leave p_i with this insufficient assignment it can not have produced a valid $(1 - \epsilon)$ -balanced assignment. So this can not be the case.

□

So shortest augmenting paths can be good lower bounds for the amount of flipping done, but this is not necessarily the dominating time factor when solving our matching problem. It may very well be that an augmenting path is hard to find.

Length of augmenting paths We will show that any key in $S \cup \bar{S}$ has an $O(\log |S|)$ length augmenting path to some available element in the following lemma, if the graph supports larger expansion than needed for our assignments. This means that when we want to assign neighbors to a newly inserted key, we know that all its neighbors have an augmenting path of at most this length, and thus that the newly inserted node has an augmenting path of this length plus one.

Lemma 5.3 (Path length to a available element). *Let $G(U, V, E)$ be an $(n, d, (1 - \frac{\epsilon}{k})d)$ -expander graph in some OPS data structure, where $k > 1$ is some constant, and where we use $(1 - \epsilon)$ -balanced assignments. Let G_S be some (S, A) -induced assignment subgraph of G . Then from any key in $S \cup \bar{S}$ there exists a augmenting path to some available element of length at most $O(\log |S|)$.*

Proof. For some key x in $S \cup \bar{S}$ we will gradually build the set of keys reachable from x in an increasing amount of steps. The set X_i is the set of

keys reachable from x using at most i elements of V (interleaved with keys of $S \cup \bar{S}$). We will show that if no available elements are encountered each set X_{i+1} is least of size $|X_i| + \Omega(|X_i|)$ and this means that when $i = O(\log |S \cup \bar{S}|)$ our set will contain all of $S \cup \bar{S}$.

Going from X_i to X_{i+1} , we know that $|\Gamma(X_i)| \geq |X_i|(1 - \frac{\epsilon}{k})d$. Since there are no available elements in $\Gamma(X_i)$ we know that precisely $|X_i|(1 - \epsilon)d$ of them are assigned to keys in X_i , but what remains are $|X_i|(\epsilon - \frac{\epsilon}{k})d$ elements of V that must be assigned to some keys outside X_i . There can be a maximum of $(1 - \epsilon)d$ elements assigned to the same key or else they would be available. Hence we have $|X_{i+1}| \geq |X_i| + \frac{|X_i|(\epsilon - \frac{\epsilon}{k})d}{(1 - \epsilon)d}$. At each such step we increase $|X_i|$ by a constant fraction of X_i , and since it cannot increase above $|S|$, we can at most take $O(\log |S|)$ such steps before encountering nodes with available neighbors. \square

The $(1 - \epsilon)d$ -augmenting path problem We do not really want to assign only 1 element to a , but $(1 - \epsilon)d$. We could do this by repeatedly finding and flipping 1 augmenting path. One could think the search process could be optimized by finding the paths all at once, but here some caution has to be exercised. If we want to find all paths before flipping them we must ensure that they are edge disjoint and the set of their end elements can accommodate $(1 - \epsilon)d$ assignments, i.e. their state as available elements are not interrelated. The $(1 - \epsilon)d$ augmenting paths have to be edge disjoint since if two augmenting paths p and q share the edge e then when we flip p , we know for sure that q is broken at e .

5.3 The Structure of the Induced Assignment subgraph

We will in this section try to discern various properties of the structure of the Induced Assignment Subgraph, and discuss how they affect various insertion strategies. Specifically we will see why it is important for the previously introduced 'grab-all' scheme to use an expander that supports an extra large balanced assignment and we show that the worst case complexity the grab-all algorithms is as bad as it could be. We then give some informal thoughts on whether or not a random walk is sufficient to efficiently discover augmenting paths in the Induced Assignment Subgraph. Finally we discuss the benefits of using a random walk for insertions in the parallel dictionary data structures of Chapter 3.

5.3.1 The importance of a larger expansion than assignment constraint

In the original OPS data structure an expansion factor that is larger than the fraction of $\Gamma(x)$ that has to be assigned to x for every $x \in S$ was used in order to obtain efficient updates amortized. To be exact, the expansion factor is increased to $(1 - \frac{\epsilon}{3})d$ while still requiring the usual $|\Gamma_{asn}(x)| \geq (1 - \epsilon)d$ for every $x \in S$. For this setup they prove an amortized running time of $O(d)$ as we have restated in Lemma 1.9. We know from Lemma 3.4 that there exist an expander that support an expansion factor of $(1 - \frac{\epsilon}{k})d$ for any constants $\epsilon, k > 0$.

In the following we will show by giving an example that if $k = 1$ it will neither be possible to ensure a worst-case or an amortized performance on insertions of less than $O(nd)$. After this we will consider the case of some appropriate $k > 1$ and gain some intuition as to why this choice seems to improve the time bound on insertions substantially. We will then focus on the problem of establishing a worst-case bound on insertions for some appropriate $k > 1$. We will not prove any bound but we believe that a worst-case running time of $\text{polylog}(n)$ is possible.

5.3.2 Viewing assignments as sets

In this section we will use sets to illustrate the intersections $\Gamma(x_1) \cap \Gamma(x_2)$ for the keys $x_1, x_2 \in U$, each set will be represented in the figures in this section as an ellipse.

Suppose that for some $x_1, x_2 \in U$ that $\Gamma(x_1)$ and $\Gamma(x_2)$ intersects each other. Viewing this as sets will have the following implications:

- The set $\Gamma(x_1)$ and the set $\Gamma(x_2)$ (the ellipses) will literally intersect each other

- The size of $|\Gamma(x_1) \cap \Gamma(x_2)|$ will be indicated in the intersecting region.
- Solid lines will indicate which of the elements that belongs to x_1 and x_2 . If for instance the set $\Gamma(x_1)$ is “overlapping” the set $\Gamma(x_2)$ then it means that the elements in $\Gamma(x_1) \cap \Gamma(x_2)$ is assigned to x_1 .

To give an example, in figure 5.3, $\Gamma(x_4)$ and $\Gamma(x_5)$ are intersecting in $d\epsilon$ elements and all of the intersecting elements are assigned to x_5 .

Worst-case

Consider figure 5.3. Each ellipse corresponds to the d elements in $\Gamma(x_i) \subseteq V$ where $x_i \in U$ is listed below the ellipse in the figure. For now ignore the dotted ellipses in the left part of the figure. We consider a set $\{y_1\} \cup \{x_1, \dots, x_l\} \subseteq U$ so that $|\Gamma(x_i) \cap \Gamma(x_{i+1})| = d\epsilon$ for $1 \leq i < l$. We now insert x_1, x_2, \dots, x_l (all the white ellipses) into S using the “grab-all” insertion scheme described in [OP02]: Each insertion of an x_i is performed by assigning all elements in $\Gamma(x_i)$ to x_i and reassigning any key in S having less than $(1 - \epsilon)d$ assignments. In the insertion of x_1, x_2, \dots, x_l no extra “grab all” operations beyond the initial one are necessary. After these insertions we have $|\Gamma_{asn}(x_1)|, \dots, |\Gamma_{asn}(x_{l-1})| = (1 - \epsilon)d$ and $|\Gamma_{asn}(x_l)| = d$.

Suppose we now want to insert y_1 (the shaded eclipse in the figure). Since y_1 shares $d\epsilon$ elements with x_1 , the insertion of y_1 will make $|\Gamma_{asn}(x_1)| < (1 - \epsilon)d$, which implies that x_1 has to be reassigned. After x_1 is reassigned $|\Gamma_{asn}(x_2)| < (1 - \epsilon)d$, after x_2 is reassigned $|\Gamma_{asn}(x_3)| < (1 - \epsilon)d$ and so on until x_{l-1} have been reassigned. Setting $l = n - 1$ we have an example of a sequence of insertions that, using the one-probe insertion scheme from 1.4, will require $O(n)$ “grab all” operations. Hence the worst-case running time for this insertion scheme is $\Omega(nd)$.

Figure 5.3: A example requiring $\Theta(nd)$ time worst-case and amortized

Amortized

The example above can easily be generalized to achieve a bad amortized running time. Suppose that we have to insert the sequence $x_1, x_2, \dots, x_l, y_1, y_2, \dots, y_l$. Inserting x_1, \dots, x_l goes well, as in the example above, requiring no additional “grab all” operations, but inserting y_1, \dots, y_l each require l ‘grab all’

operations¹. Setting $l = \frac{1}{2}n$ we get a amortized running-time of:

$$\frac{\frac{1}{2}nd + \frac{1}{2}n \cdot \frac{1}{2}nd}{n} = \Omega(nd)$$

which is no better than the worst case running time.

Expanders in which these cases does not occur

One might object towards this that the cases we have mentioned above might not occur in a specific expander graph for any set S . This is correct, but we are interested in insertion algorithm that works on any expander graph and not just some obscure subset of the set of expander graphs. Furthermore it is obvious that for any choice of S there exists an expander graph in which the described scenarios occur. The same applies to the cases discussed below.

5.3.3 Increased expansion factor

From the examples above we see the importance of setting k larger than 1, i.e. an expansion factor larger than the assignment requirement, as mentioned above. We will therefore in the rest of this section suppose that $k > 1$

In the following we will pretend that an expansion factor of $(1 - \frac{\epsilon}{k})d$ implies that any pair of neighborhoods for distinct keys can have an intersection of at most $(\frac{\epsilon}{k})d$ elements. This is not completely correct since in fact some of the sets can have intersections of larger size, but there can be no intersections containing more than $2(\frac{\epsilon}{k})d$ elements. We can avoid the difficulties of dealing with this by setting $k' = 2k$, and consider the case where the graph expands with a factor $(1 - \epsilon/k')d$ and generously allow pairs of sets an intersection of $(\frac{\epsilon}{k})d$. We will assume this is the case in the following and ignore this issue.

The balloon tree

Since two sets can now only have an intersection of $\frac{\epsilon}{k}d$, the closest we can get to our last example is a tree with branching factor of k , since it will take at least k sets intersecting with the maximum $\frac{\epsilon}{k}d$ elements allowed in order to prevent $d\epsilon$ of the neighbors to a key $x \in S$ to be assigned to x . An example of this with $k = 3$ and 4 occupied nodes is sketched in figure 5.4 (ignore the circle for now). We will denote such a tree of sets a *balloon tree*. Note that the leafs of the balloon tree are unoccupied just like the

¹Notice that each insertion of an y_i does not require $l + i$ “grab all” operations as one might expect but exactly l operations. This is because each insertion of an y_i makes the set with d elements assigned to it move one set to the left, hence when inserting y_i we will for each element in $\{y_i, \dots, y_1, x_1, \dots, x_{l-i}\}$ need to perform a “grab all” operation in order to maintain $(1 - \epsilon)d$ assignments.

rightmost node is our last example (figure 5.3) was unoccupied. Actually we can view our last example as the case $k = 1$: In the last example there was $\frac{\epsilon}{1}d$ intersection, and a branching factor of 1.

Figure 5.4: A worst-case requiring $\Omega(nd)$ time for one insertion

In this tree we are no better off worst-case than in the first example if we use the 'grab all' insertion scheme from [OP02]. Suppose we want to insert an element in S corresponding to a set that intersects $(1 - \frac{\epsilon}{k})d$ with the root of the tree (the circle in figure 5.4). This will cause the root to perform a "grab all" operation which will cause each of the children of the root to perform a "grab all" operation, which will cause the grand children of the root to perform a "grab all" operation etc. The result is that all the keys in the tree has to perform a "grab all" operation, except the leaves. Hence insertion as done in [OP02] will also run in $\Omega(nd)$ in this case, and we can conclude that increasing the expansion requirement by a constant factor will not improve the worst-case performance of the insertion scheme from [OP02].

Notice that since the leaves will not need a "grab all" on the first insert-operation a second insertion in the root node of the balloon tree will not need a "grab all" operation for the parents of the leaves and so on. Hence the number of "grab all" operations needed will decrease very quickly (almost by a factor k) yielding a good amortized performance. Our Further considerations will only deal with worst-case running time.

5.3.4 A new insertion scheme

In the light of the bad worst-case performance of the standard insertion procedure, we will now introduce a new insertion scheme for the one-probe

data structure that will perform remarkably better in the example sketched in figure 5.4. Basically the algorithm is just a random walk in the graph searching for an augmenting path and flipping the edges during the walk. The new insertion algorithm works as follows:

INSERT(x)

1. Assign all vacant elements in $\Gamma(x)$ to x
2. **while** $|\Gamma_{asn}(x)| < (1 - \epsilon)d$ **do** REASSIGN(x)

REASSIGN(x)

1. Chose a random $v \in \Gamma(x) \setminus \Gamma_{asn}(x)$
2. Let y denote the element to which v is assigned
3. Assign v to x
4. Assign all vacant elements in $\Gamma(y)$ to y
5. **if** $|\Gamma_{asn}(y)| < (1 - \epsilon)d$ **then** REASSIGN(y)

Let us describe what is happening here. First INSERT(x) assign all vacant elements from $\Gamma(x)$ to x . If this is enough to achieve the $(1 - \epsilon)d$ assignments to x we are done. If we still need assignments we call REASSIGN(x) that works by taking a random walk in the graph until it reaches a node that can spare an assignment, subsequently flipping all assignments on the path. INSERT(x) keeps on calling REASSIGN(x) until x has enough (i.e. $(1 - \epsilon)d$) elements in V assigned to x .

It seems reasonable that the algorithm halts in all cases with very high probability. The crucial point is how many times REASSIGN has to be called expected facing the worst possible case of insertion.

5.3.5 Finding a worst-case for the new insertion scheme

The new insertion scheme applied on the example sketched by figure 5.4, will run in $O(d)$, since the two sets only intersect in $(\frac{\epsilon}{k})d$ elements and hence no calls to REASSIGN are necessary. We will therefore change the setting to what might be a worst-case for INSERTION.

Consider a case, where the neighborhoods of the elements in S forms $\frac{k}{\epsilon}$ balloon-trees and that the inserted element x intersects $\frac{\epsilon}{k}d$ elements in each of the trees. In this case all elements of x are occupied and we need $(1 - \epsilon)d$ calls to REASSIGN in order to attain the sufficient number of assignments to x . Each call will run in $O(\log_k(n))$ so the total running time will be $O(d \log_k(n))$. Which certainly is the best we can hope for.

Conjecture 5.4. *The balloon forest is a worst-case for the new insertion scheme*

Unfortunately we are not able to prove this conjecture which is crucial to obtaining a bound on the complexity of the new insertion algorithm. We however believe that even if this conjecture doesn't hold the worst-case complexity of the insertion scheme will not increase beyond $d \text{polylog}(nd)$.

5.3.6 Augmenting paths on parallel disk models

When finding augmenting paths in the parallel disk models, we might achieve a substantial speedup by running the search for augmenting paths in parallel. To do this we need to change line 1 of REASSIGN so that it just chooses a random neighbor, since we otherwise need a parallel probe to determine which neighbors we should not continue the random walk through. This however poses three problems:

Edge collisions First, if two random walks both choose to read the assignment of the i th neighbor of the key they are in at the same step, we will need to perform 2 I/O's to the same disk in the PDM. However, given that there are less than d random walks, and each walk at each step chooses uniformly among d neighbors $O(1)$ steps expected should suffice to allow all the walks to take a single step.

Avoiding the bottleneck of assignment size queries Secondly we need to know when we have reached an available node without resorting to a lookup of all it's neighbors, we could do this by a lookup to A , the table we used previously to store assignment size info, but then we would again encounter the bottleneck in the PDM that forced us to emulate the PDHM in order to access A . One might consider one more change to the insertion algorithm such that $\text{INSERT}(x)$ in line 1 only assigns at most $(1 - \epsilon)d$ vacant element to x . Using this modified version of INSERT all keys have exactly $(1 - \epsilon)d$ elements assigned to them at all times. Note that this would be enough since keys that lose an assigned neighbor are immediately given another when we flip an augmenting path to an available element. The objective of REASSIGN would then simply be to locate a vacant node. This seems to make it possible to skip the counter in A in the parallel insertion algorithm, and avoid the PDHM emulation in section 3.5. However it seems this might also in some cases increase the complexity accordingly. To see this consider a simple random walk reaching a node with only one vacant neighbor, in this case the random walk only has probability $1/d$ of visiting the vacant node and ending the random walk, which in the worst case means it might fail and be forced to retry, or is might reach a node with a long augmenting path to another vacant element.

Disjoint augmenting paths Thirdly, since flipping an augmenting path can invalidate other intersecting augmenting paths, it is essential that the

set of edges including in each augmenting path is disjoint from the others. It is tricky to analyze how frequently such collisions occur, but we strongly suspect that the probability of such a collision drops quickly with the number of steps taken by the random walk.

No need for error correcting codes Another, entirely different, advantage of using the above approach is that we no longer need to use error correcting codes. When using an augmenting path to improve an assignment each key immediately receives a new assigned neighbor after losing one. This enables us to efficiently copy the data from the lost neighbor to the new assigned neighbor, implying a simple scheme were we simply distribute the satellite data over $(1 - \epsilon)d$ neighbors and then move these data pieces these as we flip augmenting paths. The only information we need in order to restore the data is a number for each piece of data indicating which part of the data it represents. Such a number can be stored using $\log((1 - \epsilon)d)$ bits which, under the reasonable assumption that $n = \Omega(d)$, will not affect the asymptotic space usage.

5.3.7 Buffering insertions

One obvious idea for improving the upper bound of worst-case insertion below $O(\log n)$ is to keep a buffer of some size in internal memory of the inserted elements. This will give us some freedom in the order of insertions, and a clever insertion strategy might enable us to prevent bad structures like the balloon tree from appearing.

It is however very hard to either prove or disprove that a buffer will improve the performance. Among the challenges of implementing such a strategy is to find a way to efficiently and continuously choose an insertion order. How hard this is relates to how 'unstable' a good scheme is, i.e., can a good plan suddenly turn bad due to a single node being added to the buffer? Another problem is to actually determine augmenting paths for those keys which are present in the buffer. This is a very tough problem as the assignment changes from inserting a single key can actually invalidate all previously present augmenting paths, forcing an update of such information.

5.4 Conclusion

In this chapter we have viewed the problem of insertion as that of finding augmenting paths to available elements in a special directed expander graph, called the Induced Assignment Subgraph. We have shown that any insertion algorithm will need to at least perform work equivalent to flipping such an augmenting path.

We have also shown that the grab-all insertion algorithm from the original OPS scheme can in some cases use an excessive amount of time for a

single insertion, in fact using the same time, asymptotically, as an entire series of n insertions.

We have suggested various variations on a random walk in order to find augmenting paths both in the original OPS data structure and in the parallel dictionaries from Chapter 3. We have also suggested a possible worst case scenario for such an insertion algorithm. Finally we have seen that there could be many benefits for the parallel data structures in using this alternative insertion algorithm, if it is possible to give a good bound on its complexity. This discussion has raised several questions that remain open:

- What is the longest shortest augmenting path from a key to an available element when considering all possible inserted sets regardless of the choice of balanced assignment? This is an important question as it gives a lower bound on the complexity of the suggested algorithm as well as any other scheme that doesn't use buffered insertions.
- What is the worst case expected complexity for finding an augmenting path in the induced assignment subgraph using a random walk?
- If we in an external memory model have access to an internal memory buffer that can hold x keys, what will then be the lower bound on the worst case of a single insertion?

Chapter 6

Feasible Expander Graph Construction

6.1 Introduction

The results covered in this thesis as well as many other results obtained regarding the application of expanders all rely on simple existence proofs of optimal expander graphs. We have seen earlier that such existence proofs also imply probabilistic constructions that succeed in constructing an expander with extremely high probability. However, in scenarios where the larger side of the bipartite expander graph is used to represent the domain of possible inputs, as in [OP02], [BMRV00] as well as this thesis, such a construction is not feasible, as an adjacency list or matrix would consume an unacceptable amount of storage space. The random structure of such a graph also makes it problematic to expect any kind of compression scheme to be able to perform any significant space reduction. There have been a major research effort concerning obtaining explicit expander constructions. Recent results have yielded significant improvements for both the unbalanced and balanced cases, however the degree required in the heavily unbalanced constructions are still far from optimal, not even lowering the degree to be polylogarithmic in the size of the universe has been achieved.

The goal of this chapter is to cover the most common as well as the most efficient techniques for constructing expander graphs. We will specifically try to leverage the fact that the one parallel probe scheme we have introduced earlier operates in external memory, which allows us to use some internal memory for storing information regarding the expander graph. Furthermore we will consider how restrictions on the balance of the graph can help improve the left degree of various expander constructions.

We will start out by giving some necessary definitions. We then move on to discuss why the, commonly used, eigenvalue method is inapplicable to our scenario. This is followed by an extensive analysis of the subject

of graph products, which involve combining smaller graphs into larger expander graphs. Specifically we will cover the Cartesian graph products which was used in the most important breakthrough on arbitrarily unbalanced expanders[TS02] and the zig-zag graph product[CRVW01][CRVW02] which has applications in providing slightly unbalanced expanders. In the end of the chapter we combine the zig-zag product with another simple composition achieving a feasible construction of an expander with polylogarithmic degree in u using space on the order of an arbitrary root of n when $u = \text{poly}(n)$. This enable us to efficiently implement our external memory results from the chapter on one probe schemes in the parallel disk models.

6.2 Preliminaries

In this section we formally define what we consider a feasible expander construction. We also introduce concepts such as min-entropy as tools for analyzing and proving the expansion property of graphs. Finally we define the extractor graph which has properties similar to the expander graph.

6.2.1 Expander construction

Definition 6.1 (Expander construction). *A (t, m) -expander construction is a construction that uses m words of space both in preprocessing and in runtime and where the computation of $\Gamma_i(x)$, for any $i \leq |\Gamma(x)|$ and $x \in U$, takes time t .*

For an expander construction of this type to be useful for the parallel probe scheme it is necessary that internal memory is at least m words, so that calculations on the graph does not affect the number of external memory operations. The preprocessing time is not included as a parameter in the definition as it is usually close enough to linear in the space consumed.

Definition 6.2 (Feasible expander construction). *An expander construction is feasible if it has parameters $(\text{polylog}(u), o(n))$.*

This expander is called feasible since any larger amount of internal memory would lead to a trivial solution for an external memory dictionary, since all keys could then be stored in internal memory. An example of feasible parameters could be a $(\text{polylog}(u), \sqrt[c]{n})$ -construction, where c is some constant larger than 1.

Note that if there is no bound on the relation between u and n we should not expect to gain anything by using $o(n)$ words of space. We will therefore in the following frequently consider the interesting case when $u = \text{poly}(n)$.

6.2.2 Graph notation

One can think of a bipartite graph as defined by the edge function $F : U \times D \rightarrow V$ where as usual U is the set of binary strings of length $\lceil \log u \rceil$ and likewise for D and V . Throughout the paper we will interchangeably use a number z and the bit-string representing z and we will adopt the notation that a capital letter is a set of strings and its small counterpart is the number of strings in that set. Given a number z , $[z]$ is then the set of $\lceil \log z \rceil$ length bit-strings representing values from 0 to $z - 1$. Furthermore for we will sometimes refer to an element $z \in (X \times Y) = (a, b)$ as the concatenated bit-string $a \circ b$. The graph defined by F is denoted G_F .

Edge functions A bipartite graph $G_F : (U, V, E)$ with left degree d is defined by the edge function $F : U \times D \rightarrow V$. Since a graph only can be defined by a single edge function and vice versa we will throughout this chapter use the two terms interchangeably, i.e. we can say that an edge function is e.g. a lossless expander and hereby mean that the the graph it defines is a lossless expander.

6.2.3 Expanders as randomness enhancers

Intuitively taking a random step from a randomly chosen left node in an expander graph will end up in a completely random node on the right side. This is due to the fact that there are no bottlenecks in an expander. In fact, given that the 'randomness' in the distribution for choosing the source node is small enough, the distribution of the destination node on the right will in fact be more 'random' than the source distribution. It is easily possible to formalize this intuition, and apply it in expander analysis. In order to do so however, we need to define some notation for distributions and give a more precise definition of randomness.

Probability distributions In the following we will interchangeably use a random variable and its probability distribution. The support of a probability distribution X is the set of strings that have a positive probability in X . $X(S)$ denotes a distribution over the set S . A probability distribution is considered flat if the distribution restricted to its support is uniform. The uniform distribution of a set S is given by the function $\Upsilon(S)$.

Combined with the graph notation introduced above $F(X, \Upsilon(D))$, where F is the edge function of some graph, is the probability distribution on the set V we reach by taking a random step in the graph starting from the probability distribution X .

Entropy

Definition 6.3 (Entropy). *The entropy of a probability distribution X , over a set S is defined as*

$$H(X) \stackrel{\text{def}}{=} \sum_{s \in S} \Pr[X = s] \cdot \log \left(\frac{1}{\Pr[X = s]} \right)$$

The above definition of entropy is the common measure of 'disorder' in a distribution. For most of the following we will instead use min-entropy which is a convenient way to express a lower bound on how much entropy a distribution contains.

Definition 6.4 (Min-entropy). *The min-entropy of a probability distribution X , over a set S is defined as*

$$H_{\infty}(X) \stackrel{\text{def}}{=} \log \frac{1}{\max_{a \in S} \Pr[X = a]}$$

It should be observed that the two measures of entropy are identical for flat distributions. For min-entropy it should also be noted that a distribution with min-entropy $\log x$, must have support of cardinality at least x .

k-source A probability distribution X is a k -source if the min-entropy of X is greater than or equal to k .

Statistical distance between random variables

Definition 6.5 (Statistical difference). *The statistical difference between random variables X and Y is defined as*

$$\frac{1}{2} \sum_{a \in S} |\Pr[X = a] - \Pr[Y = a]|$$

Definition 6.6 (ϵ -closeness). *A distribution X is ϵ -close to a distribution Y if the statistical difference between the two distributions is less than or equal to ϵ .*

Definition 6.7 ((k, ϵ) -source). *A random variable X is a (k, ϵ) -source if it is ϵ -close to some k -source.*

Lemma 6.8. *If A is an (k, ϵ) -source, the support of A is of size at least $(1 - \epsilon)2^k$.*

Proof. Suppose we have some distribution A that is a (k, ϵ) -source and the k -source it is close to is B . We know that the support of B is at least 2^k , but what can we say about the minimum support of A ? If B has the least support it can have, it will be a flat distribution where each non-zero probability is $1/2^k$. Consider that we create A by starting out with

a distribution equivalent to B and then repeatedly making the support 1 smaller. When eliminating one support we increase the statistical difference between A and B by $1/2^k$. We can do this a maximum of $2^k \cdot \epsilon$ times when we want the two to be ϵ -close to each other in the end. This leaves A with a support of $2^k - 2^k \cdot \epsilon = (1 - \epsilon)2^k$. If B was not a flat distribution and therefore had a support of $2^k + x$, for some x , we could have eliminated more support, but never more than $x + 2^k \cdot \epsilon$ which still leaves us with $(1 - \epsilon)2^k$. \square

6.2.4 Expanders

We will further need some definitions of special kinds of expander graphs.

Definition 6.9 (lossless expander¹). *A graph defined by an edge function $F : U \times D \rightarrow V$ is an (n, ϵ) -lossless expander if for any $1 \leq k \leq \log n$ and any k -source X over U , the distribution $F(X, \Upsilon(D))$ is a $(k + \log d, \epsilon)$ -source*

Theorem 6.12 will show that an (n, ϵ) -lossless expander is equivalent to an $(n, d, (1 - \epsilon)d)$ -expander. The two following lemmas we will show the direction from the randomness enhancing function to the expander graph. We will first consider the case when the input is flat.

Lemma 6.10. *Let $F : U \times D \rightarrow V$ be the edge function of an $(n, d, (1 - \epsilon)d)$ expander graph. Then for all flat distributions X s.t. $H_\infty(X) \leq \log n$ we have that $F(X(U), \Upsilon(D))$ is a $(H_\infty(X(U)) + \log d, \epsilon)$ -source.*

Loosely following proof of lemma 2.2.1, (2) \rightarrow (1) from [TSUZ01]. Consider any distribution I over U with $H_\infty(I) = i \leq \log n$, and let I_S be the support of I . Let O be the distribution on V induced by choosing a node according to I , and following one of its outgoing edges uniformly random. Let O_S be the support of O , then $(1 - \epsilon)dn \leq |O_S| \leq dn$. Furthermore we know that for all $z \in O_S$ we have $O(z) \geq 1/dn$, since all nodes in O_S must have at least one edge from I_S and I is flat.

Let $O' = \Upsilon(O_S)$. We know that the flat distribution O' is an $(i + \log d)$ -source. We will now calculate the statistical difference s between O and O' .

$$s = \frac{1}{2} \sum_{z \in O} |O(z) - O'(z)| =$$

$$\frac{1}{2} \left(\sum_{z, O(z) < O'(z)} |O(z) - O'(z)| + \sum_{z, O(z) > O'(z)} |O(z) - O'(z)| \right)$$

Consider all elements z where $O'(z) > O(z)$ (the first sum), for any such z we have

¹ $(\log n, \epsilon)$ -lossless conductor in [CRVW01]

$$|O(z) - O'(z)| \leq \left| \frac{1}{dn} - \frac{1}{|O_S|} \right|$$

since $O(z)$ is at least $1/dn$, and $O'(z) = \frac{1}{|O_S|}$ because O' by its definition is flat over the support of O .

Furthermore observe that since O and O' are probability distributions and thus sum to 1 each, it is the case that $\sum_{z, O(z) < O'(z)} |O(z) - O'(z)| = \sum_{z, O(z) > O'(z)} |O(z) - O'(z)|$

Hence we can conclude that

$$\begin{aligned} s &\leq \frac{1}{2} \cdot 2 \left(\sum_{z, O(z) < O'(z)} |O(z) - O'(z)| \right) \leq |O_S| \cdot \left| \frac{1}{dn} - \frac{1}{|O_S|} \right| \\ &= \left| \frac{|O_S|}{dn} - 1 \right| \leq \left| \frac{(1-\epsilon)dn}{dn} - 1 \right| = \epsilon \end{aligned}$$

□

Lemma 6.11. *Let $F : U \times D \rightarrow V$ be the edge function of an $(n, d, (1-\epsilon)d)$ expander graph. Then for all distributions X s.t. $H_\infty(X) \leq \log n$ we have that $F(X(U), \Upsilon(D))$ is a $(H_\infty(X(U)) + \log d, \epsilon)$ -source.*

Loosely following proof of Lemma 2.2.2 in [TSUZ01]. Consider any distribution I over U with $H_\infty(I) = i \leq \log n$.

The proof strategy is to first divide the input distribution into many flat distributions and then show that the output distributions (called O_j 's) of these are all ϵ -close to some $i + \log d$ -sources. Then it is shown that the distribution we get by combining all these $i + \log d$ -sources is as well an $i + \log d$ -source, and finally that the distribution we get from combining all the O_j 's is ϵ -close to the combined $i + \log d$ -source.

I can be represented as a convex combination of flat distributions. We have $I = \sum \alpha_j I_j$, $\sum \alpha_j = 1$ and $H_\infty(I_j) = H_\infty(I)$.

Let F denote the edge function of the expander, and let $O_j = F(I_j, \Upsilon(D))$ be the distribution on V induced by choosing a node according to I_j and following one of its edges, chosen uniformly random. By Lemma 6.10 each O_j is an $(i + \log d, \epsilon)$ -source. Finally define the distribution O'_j to be the $i + \log d$ -source that O_j is ϵ -close to.

Now let the distributions O and O' be defined as follows for all $z \in V$:

$$O(z) = \sum_j \alpha_j O_j(z)$$

$$O'(z) = \sum_j \alpha_j O'_j(z)$$

We will now show that

$$O = F(I, \Upsilon(D))$$

To see the correctness of this equivalence, consider any $z \in V$ and define the following variables: Let o_z be the probability of reaching z based on a random step from a node chosen by I , let I_j^S be the support of I_j , and let i_j denote the number of nodes I_j^S having edges to z . Then we know that $o_z = F(I, \Upsilon(D))(z) = \sum_j \alpha_j \frac{i_j}{d|I_j^S|}$. Let us now calculate $O(z)$. For each O_j we know that $O_j(z) = \frac{i_j}{d|I_j^S|}$, hence the above equivalence follows by the definition of $O(z)$.

Now, since min-entropy is concave we have, by the definition of O' , that $H_\infty(O') \geq \sum \alpha_j H_\infty(O'_j)$, i.e. O' is an $i + \log d$ source. What remains now is to show that O is ϵ -close to O' . This is easily seen by observing that

$$s = \sum_{z \in O} |O(z) - O'(z)| \leq \sum_j \alpha_j \left(\sum_{z \in O} |O_j(z) - O'_j(z)| \right) \leq \epsilon$$

□

We are now ready to state the equivalence we need between randomness enhancing functions and expander graphs.

Theorem 6.12. *An (n, ϵ) -lossless expander is equivalent to an $(n, d, (1 - \epsilon)d)$ expander, where d is some degree.*

Proof. The proof is in two steps.

- 1 An (n, ϵ) -lossless expander is also an $(n, d, (1 - \epsilon)d)$ expander. Follows directly from Lemma 6.8.
- 2 An $(n, d, (1 - \epsilon)d)$ expander is also an (n, ϵ) -lossless expander. Follows directly from Lemma 6.11.

□

We have now established that vertex expansion is equivalent to being a randomness enhancing function of the type described above. This is helpful as it allows us to work with either of these properties in constructing expanders. We will use the terms (n, ϵ) -lossless expander and $(n, d, (1 - \epsilon)d)$ -expander interchangeably throughout this chapter, and so by denoting an (n, d, α) -expander as lossless we mean that $\alpha = (1 - \epsilon)d$, for some constant $0 < \epsilon < 1$.

Definition 6.13 (Optimal expander). *Suppose the graph G is defined by the edge function $U \times D \rightarrow V$ and is a (n, ϵ) -lossless expander. Then G is an optimal expander if $|D| = O(\log \frac{n}{\epsilon})$ and $|V| = O(n|D|)$.*

Notice that when we have a polynomial relation between u and n the size of D will be $O(\log u)$. Suppose that $n^a = u$, for some constant $a > 1$, then we have that $O(\log \frac{n^a}{n}) = O(\log n^{a-1}) = O(\log u)$.

6.2.5 Extractors

An extractor is mainly different from an expander in that it does not guarantee expansion for all sets smaller than or equal to some n , but instead promises to expand a set of sufficient size to nearly the entire right side of the graph.

Definition 6.14 (Extractor). *A graph defined by an edge function $F: U \times D \rightarrow V$ is a (k, ϵ) -extractor if for any k -source X over U the distribution $F(X, \Upsilon(D))$ is ϵ -close to $\Upsilon(V)$.*

That is, the edge function of an extractor graph can take a random variable $X(U)$ that has a non-uniform distribution (it needs to be a k -source though) and using a small uniform distribution (on D) it then guarantees to output something close to uniform on V .

6.3 Expanders and Eigenvalues

The most common technique for studying graph expansion involves analyzing the eigenvalues of the adjacency matrix of a graph. In fact it is so common that we will discuss why it is not applicable for proving left-to-right expansion properties of very unbalanced bipartite graphs before moving on.

6.3.1 Lower bounds on graph expansion

Verifying the expansion property of a graph is known to be coNP-complete [BKV⁺81]. This fact makes it very interesting to have a means of obtaining a lower bound on the expansion of a given graph. Calculating the eigenvalues of a graph's adjacency matrix is an excellent alternative to a brute force approach. This verification method also reduces the problem of constructing an expander graph to one of constructing a graph with specific eigenvalues. Because of this many explicit constructions with good eigenvalues are now known. To understand what we mean by good eigenvalue we will now go more into the technical details.

6.3.2 The adjacency matrix of expanding graphs

Every $n \times n$ matrix A that is symmetric (A symmetric $\stackrel{\text{def}}{\Leftrightarrow} A_{i,j} = A_{j,i}$ for all $i, j \in 1 \dots n$) has n eigenvalues in \mathbb{R} . Notice that the any adjacency matrix of an undirected graph is symmetric.

Consider an undirected d -regular connected graph G with n nodes, an adjacency matrix A and an expansion factor of α . Let M denote the *normalized* adjacency matrix of G defined as:

$$m_{i,j} = \frac{a_{i,j}}{\text{out-degree}(i)} \text{ for all } i, j \in [1 \dots n]$$

Since G is undirected, M will be symmetric.

6.3.3 The spectral gap

Consider the eigenvalues of M . Since M is symmetric we will have n eigenvalues in \mathbb{R} . We denote them $\lambda_1 \leq \dots \leq \lambda_n$. The main issue is the value of $|\lambda_1 - \lambda_2|$ called the spectral gap or the spectral expansion. For any connected graph $\lambda_1 = 1$ hence the spectral expansion only depends on λ_2 . Let $\alpha'd$ denote the expansion factor. It has been shown (for example in [KR04]) that:

$$\alpha' \geq \frac{|1 - \lambda_2|}{2}$$

Many families of graph with explicit constructions have their origin in this approach, among others Gabber-Galil expanders ($\alpha' = \frac{2-\sqrt{3}}{2}$) and Ramanujan Graphs ($\alpha' = \frac{1}{2}$).

Neither these or any other constructions based on maximizing λ_2 have an expansion factor larger than $d/2$ and it has been shown in [Kah95] that the eigenvalue method can not give a better lower bound, thus also restricting further constructions using this method to this expansion factor. This is critical in the data structures we discussed in 1.4, where it affects the probability of a successful probe. It would also be critical for One-Probe in the parallel disk models as the requirements of these models are $(5/6)d$ in order to ensure the required $\epsilon < 1/2$.

To make matters worse there are other fundamental problems with the eigenvalue method concerning the expansion properties we require. The spectral gap provides a lower bound for the expansion of sets that are up to half the size of the entire graph. Put formally we have that in order to obtain an expansion factor of $\alpha'd$, $|\Gamma(X)| \geq |X|\alpha'd$ has to hold for every $X \subset U$ where $|X| \leq \frac{1}{2d}|U|$. For this to be satisfied the maximal expansion factor that can be obtained by the use of eigenvalues is $|V|/(\frac{1}{2d}|U|)$ and since $|U| \gg |V|$ in very unbalanced graphs the lower bound on the expansion factor obtained by eigenvalues will be close to 0. So we can conclude that the eigenvalue method will not be able to provide any acceptable lower bound on the expansion from the left side in a very unbalanced graph, and we will refrain from further references to this method.

6.4 The Cartesian Graph Product

In this section we will describe two very simple graph products, the full Cartesian product by [Sie04] and the right Cartesian product by [TS02] and we will then give a construction combining the two. The first graph product have in [Sie04]² been used to provide expanders for a universe that is polynomial in the right side of the graph for a relatively small expanding set such that the representation only takes up space proportionate to a smaller polynomial of the expanding set. The second graph product has been used in conjunction with another graph construction to provide the best known construction for arbitrarily unbalanced graphs known so far[TS02]. The combinatorial construction in section 6.4.3 will by increasing the use of internal memory and the right side size obtain a small improvement in the size of the degree compared to the Ta-Shma construction, but only in the case where there is a polynomial relation between $|U|$ and n .

6.4.1 Properties of the Full Cartesian Graph Product

The full Cartesian product takes two bipartite graphs and produce a new bipartite graph of quadratic size and quadratic degree by concatenating the output of the edge-function. In this section we will examine the properties of the Cartesian product on bipartite expanders as it is done in [Sie04] with the generalization that we will show it to work for lossless expanders with a non-constant degree as well as for $(n, d, 1)$ -expanders as shown in [Sie04].

Definition 6.15 ((Full) Cartesian Product ([Sie04] Definition 2.12.)). *Let*

$$E_1 : U_1 \times D_1 \rightarrow V_1$$

and

$$E_2 : U_2 \times D_2 \rightarrow V_2$$

We define the full Cartesian product

$$E = E_1 \otimes E_2 : (U_1 \times U_2) \times (D_1 \times D_2) \rightarrow V_1 \times V_2$$

as

$$E(x_1 \circ x_2, y_1 \circ y_2) = E_1(x_1, y_1) \circ E_2(x_2, y_2)$$

We will apply the operator to bipartite graphs in the understanding that we operate on that graphs edge function mapping from the left side to the right side.

In [Sie04] it is proved that the Cartesian product of two expander graphs with expansion factor larger than or equal to 1 will have expansion factor 1

²The Cartesian graph product first appeared in the article [Sie88] by Siegel in 1989

³. The following lemma states that the product of any two lossless expanders will be a lossless expander. We will need the fact from lemma 1.6 that in a (n, ϵ) -lossless expander there exists a $(1 - \epsilon)$ -balanced assignment for all sets $S \subseteq U$ of size at most n , that is, where each node in S can have at least $(1 - \epsilon)d$ neighbors assigned to them all at once. Observe that the other way around applies as well, that is, if we have a bipartite graph $G : (U, V, E)$ where for every set $S \subseteq U$ of size at most n there exists a $(1 - \epsilon)$ -balanced assignment, then G is an (n, ϵ) -lossless expander.

Lemma 6.16 (Cartesian product produces lossless expanders). *Let $G_1 = (U_1, V_1, E_1)$ and $G_2 = (U_2, V_2, E_2)$ be (n, ϵ) -lossless expanders, then $G = G_1 \otimes G_2 = (U, V, E)$ is a $(n, 1 - (1 - \epsilon)^2)$ -lossless expander, where $U = U_1 \times U_2$, $V = V_1 \times V_2$, and $E = E_1 \times E_2$.*

Proof. We will prove that there exist a $((1 - \epsilon)^2)$ -balanced assignment in $G_1 \otimes G_2$ for all sets of size at most n . The keys of every set S of size at most n are composed of some sets of keys $S_1 \subseteq U_1$ and $S_2 \subseteq U_2$ where $1 \leq |S_1|, |S_2| \leq n$. This means that both S_1 and S_2 has some $(1 - \epsilon)$ -balanced assignment (by lemma 1.6). We will denote these assignments $A_1 \subseteq E_1$ and $A_2 \subseteq E_2$ respectively.

Consider the two subgraphs G'_1 and G'_2 induced by the edge sets A_1 and A_2 . These are graphs with left degree between $(1 - \epsilon)d$ and d , and right degree 1. The Cartesian product of G'_1 and G'_2 will produce a subgraph of $G_1 \otimes G_2$ which by the definition of the Cartesian product has left degree between $((1 - \epsilon)d)^2$ and d^2 , and right degree 1. And so the edges induced by this graph can be used as a valid $((1 - \epsilon)^2)$ -balanced assignment in $G_1 \otimes G_2$ since the left degree of $G_1 \otimes G_2$ is d^2 . \square

The strength of the Cartesian product is that it is a very straight-forward way to construct a large lossless expander using a lossless expander of small size.

If $G = (U, V, E)$ is an (n, ϵ) -lossless expander with left degree d then

$$G^i = \underbrace{G \otimes G \otimes G \otimes \dots \otimes G}_{i \text{ terms}}$$

is an $(n, 1 - ((1 - \epsilon)^i))$ -expander graph with a left side of size $|U|^i$, right side of size $|V|^i$ and with left degree d^i . The major problem with this approach, apart from the increase in left degree is that the size of the largest expanding subset remains the same.

Suppose that we want to construct a graph G^i on basis the of some smaller graph G and we want G^i to support lossless expansion for sets of size n . In this case G has to be a $(n, \sqrt[i]{d}, \sqrt[i]{(1 - \epsilon)d})$ -expander. Since the left side of G must contain at least n elements, we cannot store G in internal

³in lemma 2.13

memory if we want to attain a feasible expander construction⁴ of G^i . In order to attain such a feasible construction with the Cartesian product we need a feasible construction of G , and attaining such a construction is just as hard as solving the original problem of attaining a feasible construction of G^i .

6.4.2 The Right Cartesian product and list expanders

In the standard Cartesian product we increase the size of both sides of the bipartite graph involved. If the two graphs shared the same left node set, we can obtain another interesting graph product. We use the same input for both edge functions and concatenate the result, resulting in a multiplicative increase in the right side and the left degree, but leaving the left side as it is.

Definition 6.17 (Right Cartesian Product ([TS02] Definition 2)). *Let*

$$E_1 : U \times D_1 \rightarrow V_1$$

and

$$E_2 : U \times D_2 \rightarrow V_2$$

We define the right Cartesian product

$$E = E_1 \otimes_R E_2 : U \times (D_1 \times D_2) \rightarrow V_1 \times V_2$$

as

$$E(x, y_1 \circ y_2) = E_1(x, y_1) \circ E_2(x, y_2)$$

In order to use this product, We will first consider a weakened version of the expansion property. Recall that we want to use the expander to allow left-side nodes to own a constant fraction of their neighbors. In that setup only a single left node can own a right side node. However, the right side node could be used to represent a list of length P , so that each right side node could be owned by P left side nodes. We will call an expander that ensures assignments given that right side nodes are allowed multiple assignments a 'List Expander'.

Definition 6.18 (List Expander ([TS02] Definition 1⁵)). *A left d -regular bipartite graph $G(U, V, E)$ is a (n, d, ϵ, P) -list expander if and only if for every $S \subseteq U$, $|S| \leq n$, the induced subgraph $G_S = (S, \Gamma(S), E' \subseteq E)$ has a subgraph with right degree at most P and at least $(1 - \epsilon)d|S|$ edges.*

⁴Recall that a feasible expander construction can only use $o(n)$ bits of memory, hence we cannot store all the $\Omega(n)$ edges in G

⁵In [TS02] a "list expander" is called a "strong condenser"

Lemma 6.19 ([TS02] Lemma 1). *Suppose $F_1 : U \times D_1 \rightarrow V_1$ defines a $(n, d_1, \epsilon_1, P_1)$ -list expander and $F_2 : U \times D_2 \rightarrow V_2$ defines a $(P_1, d_2, \epsilon_2, P_2)$ -list expander where $P_1 \geq P_2$. Then $F : U \times (D_1 \times D_2) \rightarrow V_1 \times V_2 = F_1 \otimes_R F_2$ defines a $(n, d_1 \cdot d_2, \epsilon_1 + \epsilon_2, P_2)$ -list expander.*

Elaborated proof. We will show that by removing at most $(\epsilon_1 + \epsilon_2)d|S|$ edges leaving S in $G_{F_1 \otimes_R F_2}$, we can guarantee a right degree of P_2 in $G_{F_1 \otimes_R F_2}$ restricted to S thereby proving the lemma. Let $S \subseteq U$ and $|S| \leq n$. From G_{F_1} consider marking an ϵ_1 fraction of the edges leaving $|S|$ such that the neighboring right side nodes have at most P_1 degree when counting unmarked edges. Similarly mark an ϵ_2 fraction from G_{F_2} . This is possible since both graphs are list expanders. Consider the induced subgraph G' of $G_{F_1 \otimes_R F_2}$ created by ignoring all edges caused by the previously marked edges in G_{F_1} and G_{F_2} . The maximum degree of a right side node (a, b) in G' is $\min\{P_2, P_1\}$ as in order for an edge to go from some $x \in U$ to (a, b) there has to be an edge from x to a in G_1 and from x to b in G_2 . We now count the number of edges missing in G' but present in $G_{F_1 \otimes_R F_2}$: Each time we mark a G_1 edge we remove d_2 edges from $G_{F_1 \otimes_R F_2}$. Marking a G_2 edge removes d_1 edges from $G_{F_1 \otimes_R F_2}$. In total we have removed $(\epsilon_1 \cdot n \cdot d_1) \cdot d_2 + (\epsilon_2 \cdot n \cdot d_2) \cdot d_1 = (\epsilon_1 + \epsilon_2) \cdot n \cdot d_1 \cdot d_2$ as stated in the lemma. \square

The thing to notice in the above lemma, is that the Right Cartesian product allows us to take a list expander with a large list size that expands (into the lists) for a large set and combine it with another list expander with shorter lists that expands for a set that is equivalent only to the length of the firsts expanders lists, and produce a list expander with shorter list size that expands for the large set. The price we pay is an increase in the degree and the size of the right side of the graph. This has been used in [TS02] to iteratively built an expander with no lists, by starting from a list expander with a large list size and then in each step taking the Right Cartesian Product with a graph that has smaller list size. We will examine this technique below.

In order to apply the idea introduced above, we need a family of list expanders with parameters that maximize the decrease in list size in each step, while not yielding too large a right side. Ta-Shma obtains such list expanders by noting that a special type of extractors implies a useful family of list expanders for large list size.

Definition 6.20 (Universal extractor (defined in [TS02] section 3.1.)). *A function $E : U \times D \rightarrow [2^{f(k)}]$ defines a (k, ϵ, f) universal extractor if for all $k' \leq k$ there is a restriction $E' : U \times D \rightarrow [2^{f(k')}]$ of E such that E' defines a (k', ϵ) -extractor.*

Lemma 6.21 (Universal extractors gives list expanders ([TS02] lemma 3)). *Let $E : U \times D \rightarrow V$ define a (k, ϵ, f) -universal extractor with $k - f(k)$*

monotone in k . Define $F : U \times D \rightarrow D \times V = F(x, y) = y \circ E(x, y)$. Then G_F is a $(n, d, 2\epsilon, n/2^{f(\log n)})$ -list expander.

Elaborated proof. Let $S \subseteq U$ and $|S| = n' \leq n$. Without loss of generality we show the above lemma only for expanding sets with a size that is a power of 2. We need to show that we by removing $\epsilon|S|d$ or less edges leaving S in G_F , can guarantee a right degree of at most $n/|V|$. E is a universal extractor and hence we can obtain a restriction $E' : U \times D \rightarrow V'$, $V' = [2^{f(\log n')}]$, defining a $(\log n', \epsilon)$ -extractor from it. Define $F' : U \times D \rightarrow D \times V' = F'(x, y) = y \circ E'(x, y)$. Our goal is now to show that we by removing $\epsilon|S|d$ or less edges leaving $G_{F'}$ can ensure that the right degree is at most $n'/|V'| = 2^{\log n' - f(\log n')} \leq 2^{\log n - f(\log n)} = n/|V|$ (by the monotone property of $k - f(k)$). It is sufficient to show this since $F'(x, y)$ is a substring of the bit-string $F(x, y)$ implying that each right side node in $G_{F'}$ has all the incoming edges of one or more right side nodes in G_F .

Now consider a fixed vertex in $D \times V'$ consisting of a fixed $e' \in D$ and a fixed $v' \in V'$. When we choose a node a and an outgoing edge b uniformly on the left side of G_F , we denote by p the probability of ending up in (e', v') on the right side, or stated more formally: p denotes the probability of $F'(a, b) = (e', v')$. Define $q(e', v') = |Pr_{a \in S, b \in D} [F'(a, b) = (e', v')]| - \frac{1}{|V'|d}$. In the statement q expresses p 's deviation from the uniform distribution on the right side of $G_{F'}$. Since E' is an (n', ϵ) -extractor, we know that $\sum_{e' \in D, v' \in V'} q(e', v') \leq \epsilon$. We can attain the in-degree of (e', v') in $G_{F'}$ by an expression we define as $c(e', v') = |\{a \in S \mid E'(a, e') = v'\}|$. Using this we get that:

$$\begin{aligned} q(e', v') &= \left| \frac{c(e', v')}{|S|d} - \frac{1}{|V'|d} \right| \\ \Rightarrow q(e', v') + \frac{1}{|V'|d} &\geq \left| \frac{c(e', v')}{|S|d} \right| \\ \Rightarrow q(e', v')|S|d + \frac{|S|}{|V'|} &\geq |c(e', v')| \end{aligned}$$

Both $c(e', v')$ and $|S|/|V'|$ are integers (the latter is a power of two) and therefore we can safely floor the $q(e', v')|S|d$ term:

$$|c(e', v')| \leq \lfloor q(e', v')|S|d \rfloor + |S|/|V'|$$

By erasing $\sum_{e' \in D, v' \in V'} \lfloor q(e', v')|S|d \rfloor \leq \epsilon|S|d$ edges we know that each node on the right side of $G_{F'}$ has at most degree $|S|/|V'| = 2^{\log n' - f(\log n')} \leq 2^{\log n - f(\log n)} = n/|V|$. Thus we have shown what we needed for $G_{F'}$ and therefore also for G_F as previously mentioned. \square

A good universal extractor

In the following we use $k = \log n$ in our calculations to simplify the expressions. We assume without loss of generality that n is a power of 2. In order

to obtain appropriate list expanders from Lemma 6.21, we would like a universal extractor with $f(k) = \frac{k}{2}$ or less since this yields a list expander with list size $\sqrt{2^k}$, and so the next list expander in the iteration only needs to support expansion for $\sqrt{2^k}$, which it can do with a list size of $\sqrt[4]{2^k}$. This immediately yields an iteration consisting of $\log k$ steps, that doesn't produce too large a right side.

One universal extractor with the required parameters is the modified Trevisan's extractor from [RRV99]. We let $E_{TR}^{k,\epsilon} : U \times D_{TR} \rightarrow [2^k]$ where $|D_{TR}| = 2^{O(\log^2(\log(u)/\epsilon))}$ denote such an extractor which is a $(k, \epsilon, f(k) = \frac{k}{2})$ -universal extractor. According to Lemma 6.21 $E_{TR}^{k,\epsilon}$ yields a $(k, |D_{TR}|, 2\epsilon, \frac{k}{2})$ -list expander with the edge function $F_{TR}^{k,\epsilon} : U \times D_{TR} \rightarrow D_{TR} \times [2^{\frac{k}{2}}]$.

The construction

Consider the function defined by

$$F = \underbrace{F_{TR}^{k,\epsilon_0} \otimes_R F_{TR}^{k/2,\epsilon_0} \otimes_R F_{TR}^{k/4,\epsilon_0} \otimes_R \dots \otimes_R F_{TR}^{c,\epsilon_0}}_{\log k \text{ terms}} \otimes_R C \quad (6.1)$$

where $\epsilon_0 = \frac{\epsilon/2}{\log k}$. What happens in the equation is that we keep on reducing the list size using Lemma 6.19 until we reach a constant list-size of c . When we reach this size we construct a $(k, D_{TR}, \epsilon_0, 1)$ -list-expander C in internal memory. Since the list size of C is 1, C is a normal expander, hence F will be a normal expander as well.

The parameters

We will now consider the parameters of F . Since all the component list expanders in 6.1 have a left-degree of at most $|D_{TR}|$ it follows by Lemma 6.19 that the left-degree of F is at most

$$|D_{TR}|^{\log k} = 2^{O(\log^2(\log(u)/\epsilon_0)) \cdot \log \log n} = 2^{O(\log^2(\frac{2 \cdot \log u \cdot \log k}{\epsilon})) \cdot \log \log n} = 2^{O(\log^2 \log u \log \log n)}$$

Consider the size of the right-side of G_F . Since the right side of the graph defined by F_{TR}^{i,ϵ_0} is $|D_{TR}| \cdot \frac{k}{2^i}$, we get by Lemma 6.19 that the size of the right-side of G_F is at most

$$\prod_{i=1}^{\log k} |D_{TR}| \cdot \frac{k}{2^i} \leq |D_{TR}|^{\log k} \cdot n = 2^{(\log \log u)^2 \log \log n} \cdot n$$

At last we calculate the error of F by Lemma 6.19 to be

$$\sum_{i=1}^{\log k} 2\epsilon_0 = 2 \cdot \frac{\epsilon/2}{\log k} \cdot \log k = \epsilon$$

We can now state the result from [TS02] as the following theorem:

Theorem 6.22 (Result from [TS02]). *For any $\epsilon < 1$ and $n \leq u$, there exist an explicit construction of a (n, ϵ) -lossless expander graph G_F with the edge function $F : [u] \times [d] \rightarrow [nd]$, where $d = 2^{O((\log \log u)^2 \log \log n)}$*

In the original presentation in [TS02] the construction 6.1 is stopped earlier by an expander based on error-correction codes with the purpose of reducing the time to calculate an edge. As this is less important for our application we use the simpler version of the construction stated above.

6.4.3 Trading space for a smaller degree

In the previously introduced parallel data structure, the high degree of the construction above might be a problem, as it implies that a very large number of disks must be used in parallel. In this section we consider a trade-off where we reduce the degree of the above construction at the cost of increasing the number of nodes in the right side of the resulting expander graph.

The general idea is that we by using some internal memory can reduce the number of steps in the iteration (6.1). We achieve this reduction by constructing an expander graph in internal memory, use the full Cartesian product to enlarge this graph, and use the enlarged graph to end the iteration of (6.1) before it ended in Ta-Shma's construction. The rest of this section is divided into three parts:

1. Constructing the enlarged graph.
2. Using the enlarged graph to reduce the length of the iteration in (6.1).
3. Analyzing the left-degree and right-side of the achieved construction.

The construction will assume that u is polynomial in the size of n . We will also assume that $o(n)$ bits of internal memory are available, which is the case by definition of a feasible expander construction (definition 6.2).

Assume in the following that $u = n^a$, for some constant $a \geq 1$, and that we have internal memory of $\sqrt[x]{n}$ words, where $x > 1$.

Constructing the enlarged graph We start out by constructing an $(\sqrt[x]{n}, \epsilon)$ -lossless expander graph in internal memory. As shown in Lemma 3.4 we can do this probabilistically and achieve the desired expander with very high probability. We denote this graph G_M , defined by the edge function $M : [\sqrt[x]{n}] \times [\log \sqrt[x]{n}] \rightarrow [\sqrt[x]{n} \cdot \log \sqrt[x]{n}]$, there $b > 1$ is some constant.

We can now construct an enlarged version of G_M using the full Cartesian product:

$$G_C = (G_M)^{ax} = \underbrace{G_M \otimes G_M \otimes G_M \otimes \dots \otimes G_M}_{ax \text{ terms}}$$

Using Lemma 6.16 we get that G_C is an $(\sqrt[b]{n}, 1 - ((1 - \epsilon)^{ax}))$ -lossless expander defined by the edge function $C : [u] \times [(\log \sqrt[b]{n})^{ax}] \rightarrow [\sqrt[b]{u} \cdot (\log \sqrt[b]{n})^{ax}]$.

Using the Right Cartesian product Since the size of the universe of G_C is u we can use G_C as the graph ending the iteration of (6.1). Since G_C support expansion for sets of asymptotic larger size, than the graph G_F used by Ta-Shma did, we can reduce the iteration of (6.1) to:

$$F = \underbrace{F_{TR}^{\log(n), \epsilon_0} \otimes_R F_{TR}^{\log(n)/2, \epsilon_0} \otimes_R \dots \otimes_R F_{TR}^{\log(n)/bx, \epsilon_0}}_{\log bx \text{ terms}} \otimes_R C \quad (6.2)$$

The size of G_F The degree in G_F , our resulting lossless expander, will be $d_f = 2^{\log d_{TR} \cdot \log(bx) + \log(\log \sqrt[b]{n})^{ax}}$ and the right side size is as in the construction above determined by the number of bits in the output of F . If the right side of G_C was optimal, ie. n times the degree, we would also have had an optimal right side in G_F , but this is not the case as G_C , because of the application of the full Cartesian product, has an oversized right size.

We can now count the the number of bits in the output of F by adding the bits from the following three components: the degree d_f , all the right sides of the Trevisian graphs, and the right side of G_C :

$$\begin{aligned} \log d_f + \left(\sum_{i=1}^{\log(bx)} \frac{\log n}{2^i} \right) + \frac{\log n \cdot a}{b} &= \log d_f + \left(\log n - \log n \cdot \frac{1}{bx} \right) + \log n \cdot \frac{a}{b} = \\ \log d_f + \log n + \left(\log n \cdot \frac{a}{b} - \log n \cdot \frac{1}{bx} \right) &= \log d_f + \log n + \log n \cdot \frac{ax - 1}{bx} \end{aligned}$$

If $\frac{ax-1}{bx}$ is less than 1, G_F will have a right side of the still somewhat sub-optimal degree times n times some root of n . Recall that the parameter a is given by the difference in u and n , i.e. roughly how unbalanced G_F can be, while b is a parameter we can increase to a suitable larger constant. Increasing b will only result in lengthening the Travisian iteration and thereby increasing the constant exponent in the degree.

6.4.4 Conclusion

We have in this section seen the presently best known explicit construction of a lossless expander graph, which is also the result used in the original OPS article [OP02]. This construction by Ta-Shma gives an expander with a degree of $d_t = 2^{O((\log \log u)^2 \log \log n)} = (O(\log u))^{O(\log \log n \cdot \log \log u)}$ and a right side size of $O(n \cdot d_t)$. The value d_t is unfortunately considerable worse than polylogarithmic in u .

We have also seen the combinatorial construction which can be used in an external memory model where we have $o(n)$ internal memory. The result of the combinatorial construction have a left degree of $2^{O(\log \log u) \cdot O(\log \log u)} = (O(\log u))^{O(\log \log u)}$, which is significantly worse than polylogarithmic in u , but still somewhat better than the degree of the Ta-shma construction. The cost of this is an increase in $|V|$ by a factor of an arbitrarily small constant root of n which is considerable. We will in section 6.7 obtain a construction that improves the degree beyond what we obtained here, without increasing the right side of the graph.

6.5 The Original Zig-Zag product

The main purpose of this section is to make the reader gain some intuition about the Zig-Zag product in its basic form, which will be useful in the enhanced version covered in section 6.6, which plays a critical role in our feasible construction of an expander graph.

6.5.1 The setup

Suppose we have a $|D_1|$ -regular graph $G_1(U_1, E_1)$ and a $|D_2|$ -regular graph $G_2(D_1, E_2)$ (as in figure 6.1) where $|U_1| \gg |D_1|$. We will denote G_1 as *the large graph* and G_2 as *the small graph*. Notice that the **degree** of the nodes in the large graph has to be the same as the **number** of nodes in the small graph.

A node in the Zig-Zag product $G_1 \otimes G_2$ is a pair of nodes, one from each of the input graphs, i.e. for every $x_1 \in U_1$ and $x_2 \in D_1$ there is a node (x_1, x_2) in the graph $G_1 \otimes G_2$.

The tricky part of the Zig-Zag product concerns determining the edges in $G_1 \otimes G_2$, how this is done is explained in section 6.5.2. As usual we assume that we have an ordering of the edges of a node.

6.5.2 How to follow an edge in the zig-zagged graph

The degree of $G_1 \otimes G_2$ will be $|D_2|^2$ so that the degree depends solely on the degree of the small input graph. To clarify the process of following an edge in the resulting graph, we will number edges incident to a key in $G_1 \otimes G_2$ by pairs $(r', r'') \in D_2 \times D_2$.

Say we want to follow the (r_2, r_3) th edge from the node $(x_1, x_2) \in U_1 \times D_1$. This is done in the following way:

- Step 1** (In the small graph) We begin at the node x_2 and follow the r_2 th outgoing edge from x_2 and arrive at some node say r_1 .
- Step 2** (In the large graph) We begin at the node x_1 and follow the r_1 th edge from x_1 and arrive at some node say y_1 . Notice that we are using the location (node number) in the small graph to decide which edge to follow in the large graph. This is why the number of nodes in the small graph has to be equal to the degree of the large graph.
- Step 3** (In the small graph) We continue in the small graph from where we left off, in node r_1 , and follow the edge number r_3 . We arrive at some node say y_2 .

Hence in the graph $G_1 \otimes G_2$ the node (y_1, y_2) is the current location after following the (r_1, r_2) th edge from the node (x_1, x_2) .

An example. Let us consider an example of a step in the zig-zagged graph $G_1 \otimes G_2$. The two argument graphs G_1 and G_2 are shown in figure 6.1.

Figure 6.1: **Left** G_1 : $|U_1| = 18$, $|D_1| = 4$ **Right** G_2 : $|D_1| = 4$, $|D_2| = 3$.

Consider the illustration above. Number the edges so that the large graph has edges numbered from $\{1, 2, 3, 4\}$ and the small graph has edges numbered from $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, in this way:

The dotted lines : numbered with **1** and **a** respectively
 The dashed lines : numbered with **2** and **b** respectively
 The normal solid lines : numbered with **3** and **c** respectively
 The thick solid lines : numbered with **4**

Suppose we want to follow the (\mathbf{a}, \mathbf{b}) th edge from the node $(a, 1)$ in $G_1 \otimes G_2$. The initial node $(a, 1)$ corresponds to a pair of locations in the two graphs, namely the node a in the large graph (G_1) and the node 1 in the small graph (G_2). The three steps in G_1 and G_2 are as follows:

- Step 1** (In the small graph) We follow the \mathbf{a} th edge in the small graph (the dotted line), from node 1 and arrive at the node 2.
- Step 2** (In the large graph) Based on the current location in the small graph (i.e. 2) we follow the 2th edge in the large graph (the dashed line) from node a and arrive at B .
- Step 3** (In the small graph) Finally we follow the \mathbf{b} th edge in the small graph, (the dashed line) from the node 2 and arrive at 3. We are now located

at node $(B, 3)$ in the zig-zagged graph $G_1 \otimes G_2$.

6.5.3 Definition of the Zig-Zag product

We hope that the above example has given the reader some intuition about how the zig-zag product works. We will now define the Zig-Zag product in terms of edge functions.

Definition 6.23 (The original Zig-Zag product[RVW01]). *Let the graphs G_1 , G_2 , and $G_1 \otimes G_2$ be defined by the edge functions E_1, E_2 , and E respectively. And let:*

$$\begin{aligned} E_1 & : U_1 \times D_1 \rightarrow U_1, \\ E_2 & : D_1 \times D_2 \rightarrow D_1, \text{ and} \\ E & : U \times D \rightarrow U \end{aligned}$$

$$\begin{aligned} \text{where } U & \equiv U_1 \times D_1 \text{ and} \\ D & \equiv D_1 \times D_2 \end{aligned}$$

The edge function E is defined as follows: For any

$$\begin{aligned} x_1 & \in U_1, \\ r_1, x_2 & \in D_1, \text{ and} \\ r_2, r_3 & \in D_2 \end{aligned}$$

$$E(x_1 \circ x_2, r_2 \circ r_3) \stackrel{\text{def}}{=} y_1 \circ y_2, \text{ where}$$

$$\begin{aligned} (\text{Step 1:}) \quad r_1 & \stackrel{\text{def}}{=} E_2(x_2, r_2) \\ (\text{Step 2:}) \quad y_1 & \stackrel{\text{def}}{=} E_1(x_1, r_1) \\ (\text{Step 3:}) \quad y_2 & \stackrel{\text{def}}{=} E_2(r_1, r_3) \end{aligned}$$

6.5.4 The expansion property

The idea behind the Zig-Zag product is to use input graphs with good expansion in order to produce a larger expander graph. In the following we will restrict our attention to bipartite graphs. The Zig-Zag product also applies to non-bipartite graphs, but we have no interest in those.

We will now restate the theorem that guarantees that the Zig-Zag product of expander graphs is also an expander. We will follow this up with an intuitive argument for the correctness. For a formal proof refer to [CRVW01]. First we give an alternative definition of an expander to ease the readability of the following theorem.

Definition 6.24 (simple expander). *Suppose the graph G is defined by the edge function $E : U \times D \rightarrow V$. Let E have the property that for all distributions X over U s.t. $H_\infty(X) \leq \log(u) - a$ we have that $E(X, \Upsilon(D))$ is ϵ -close to a $H_\infty(X) + a$ -source. Then G is a (U, D, V, ϵ, a) -simple expander.*

Theorem 6.25 (Original zig-zag product entropy flow [RVW01]). *Suppose G_1 is a $(U_1, D_1, V_1, \epsilon_1, a_1)$ -simple expander and G_2 is a $(D_1, D_2, D_1, \epsilon_2, a_2)$ -simple expander, then $G_1 \otimes G_2$ is a $(U_1 \times D_1, D_2 \times D_2, V_1 \times D_1, \epsilon_1 + \epsilon_2, \min\{a_1, a_2\})$ -simple expander.*

Entropy flow intuition In arguing for the correctness of the above theorem informally we will, to ease the intuitive understanding, pretend that the source graphs have a more powerful property than is actually the case. More specifically, we will assume a random step on G_1 adds a_1 bits of entropy up to a maximum of $\log |V_1|$ and that a random step on G_2 adds a_2 bits of entropy up to a maximum of $\log |D_1|$. This is not correct as the only guarantee is that the output distribution induced by taking a random step based on some input distribution is ϵ -close to a distribution with the desired entropy. Under this assumption the product graph will also have this stronger property. While only being an intuitive argument this approach makes it possible to grasp the principle behind the definition.

Suppose we have an input probability distribution $X = X_1 \times X_2$ where X lacks at least $\min\{a_1, a_2\}$ bits in being uniform as required in the theorem. We want to show that after taking a random step from X in $G_1 \otimes G_2$ the resulting distribution has gained at least $\min\{a_1, a_2\}$ bits of entropy compared to X . We will consider two cases, one where X_2 lacks at least a_2 bits of entropy in being uniform and one where X_2 is uniform. It can be shown that these two cases are sufficient, for details see [CRVW01] or the proof of entropy flow in the new zig-zag product (proof of theorem 6.29).

We now take a random step in $G_1 \otimes G_2$, that is, we calculate $E(X_1 \times X_2, R_2 \times R_3) = Y_1 \times Y_2$ and consider the entropy in the output distribution $Y = Y_1 \times Y_2$ compared with the entropy of the input distribution $X = X_1 \times X_2$. We divide our proof into two cases:

$X_2 | X_1$ lacks at least a_2 bits in being uniform :

- Step 1** Since we have supposed that a random step in G_2 adds a_2 bits of entropy to the output distribution R_1 , and X_2 lacks at least a_2 bits of entropy, all a_2 bits of entropy is added to R_1 which then has entropy $H(X_2) + a_2$.
- Step 2** Depending on the entropy in X_1 we can add $a' \leq a_1$ bits of entropy from R_1 in addition to what is contained in X_1 . Thus the output distribution Y_1 now contains $H(X_1) + a'$ bits of entropy.
- Step 3** Intuitively R_1 'lost' a' bits of entropy in the previous step since it is now dependent on Y_1 and we are interested in counting the total entropy. Hence we have $H(R_1 | Y_1) = H(X_2) + a_2 - a'$ entropy as the input distribution to this step. G_2 is capable of adding a_2 bits of entropy as long as the output entropy does not exceed $\log |D_1|$ bits. It might be the case though that $H(R_1) = \log |D_1|$ and so we

can only add a' bits of entropy in this step, since exactly so many bits of R_1 are non-random. Hence the output distribution $Y_2|Y_1$ of this step has entropy $\min\{H(X_2) + a_2, H(X_2) + a_2 - a' + a_2\} = H(X_2) + \min\{a_2, 2 \cdot a_2 - a'\}$ depending on whether a' is larger than a_2 or not.

Total In the final output distribution (Y_1, Y_2) we now have entropy $H(X_1) + a' + H(X_2) + \min\{a_2, 2 \cdot a_2 - a'\} \geq H(X) + \min\{a_1, a_2\}$ as required.

$X_2|X_1$ is uniform :

Step 1 Since X_2 is uniform a random step will result in a uniform output distribution R_1 and so no improvement in entropy is attained in this step.

Step 2 Since X_2 is uniform we know that X_1 must lack at least $\min\{a_1, a_2\}$ bits of entropy in being uniform. Hence we can shift $\min\{a_1, a_2\}$ bits of entropy from R_1 to Y_1 in addition to the entropy contained in X_1 by taking a random step in G_2 .

Step 3 In the step above R_1 'lost' $\min\{a_1, a_2\}$ bits of entropy since it is now dependent on Y_1 , so we now have $H(R_1|Y_1) = H(X_2) - \min\{a_1, a_2\}$ entropy. Since $H(X_2) = \log |D_1|$, Step 3 will add exactly $\min\{a_1, a_2\}$ bits of entropy, giving us an output entropy $H(Y_2|Y_1) = H(X_2)$.

Total In the final output distribution (Y_1, Y_2) we now have entropy $H(X_1) + \min\{a_1, a_2\} + H(X_2) = H(X) + \min\{a_1, a_2\}$ as required.

Back in the expander graph setting The entropy flow intuition above, though informal, shows that entropy flow of the zig-zagged graph is determined by the entropy flow of the graph with the least entropy flow. But what we would really like to say something about is the relation between the expansion factor and the degree of the graph $G_1 \otimes G_2$.

Recall Lemma 6.10 showing that a lossless expander, viewed as an edge function, determine the expansion properties of the graph based on the output distribution of a random step from a flat distribution over some set S . And recall that when viewing an expander as a graph, $\Gamma(S)$ is the support of the output distribution. By Theorem 6.25 we have that the output distribution of a graph made by the Zig-Zag product will be a $(\log |S| + \min\{a_1, a_2\}, \epsilon)$ -source, and hence its support is at least $(1 - \epsilon)2^{\min\{a_1, a_2\}}|S|$ by Lemma 6.8.

6.5.5 Conclusion

The graph we would like to produce is one where the expansion factor is optimal as a function of the degree, i.e. an $(n, d, (1 - \epsilon)d)$ -expander, for

some constant $0 < \epsilon < 1$. But as we have seen the resulting graph will have a degree that is quadratic in the degree of the small input graph and will only guarantee to have the smallest expansion factor of the two input graphs. If the two input graphs both have optimal expansion factors this will give the resulting graph the expansion factor from the small graph and it will then have a ratio that is a square root of the degree between the degree and the expansion factor. If the degree is not a constant then neither is the expansion factor and then we will not end up with an *lossless expander*.

6.6 The new Zig-Zag product

6.6.1 Introduction and preliminaries

The new version of the zig-zag product resembles the original product in that it uses three steps in some graphs in order to define the resulting graph. The major change is the addition of a third graph to this process.

The problem with the original version was that only one of the steps in the small graph was guaranteed to add any entropy to the final output. The new Zig-Zag product introduces entropy buffers associated with both the the large and the small graph. A third graph takes the buffers as input and carries their entropy to the final output. The formal definition is given below followed by an intuitive work-through of the three steps in the input graphs and some intuition concerning the entropy flow in them. Theorem 6.29 then formalizes the guarantee of the desired entropy flow in the resulting graph. In theorem 6.31 we use the new Zig-Zag product to obtain a *lossless expander*. Finally we use Theorem 6.31 to obtain a feasible construction for a limited range of parameters by restricting the balance of the graph .

In order for the zig-zag product to produce a *lossless expander* the edge functions of two of the input graphs must have some special properties: The graphs must be respectively a *permutation expander* and a *buffered lossless expander*, both of them defined below.

Buffered lossless expander. The buffered lossless expander is a *lossless expander* with an extra property. The edge function of such an expander is comprised so that the first part of its output gives an element in a normal non-lossless expander, while the whole output gives an element in a *lossless expander*.

Definition 6.26 (buffered lossless expander⁶). *A graph is an (n, ϵ, a) -buffered lossless expander if it is defined by an edge function F , composed of a pair of functions $\langle F', F'' \rangle: U \times D \rightarrow V \times B$ where F' defines a $(n, d, (1 - \epsilon) \cdot a)$ -expander and F defines a (n, ϵ) -lossless expander.*

⁶ $(\log n, \epsilon, \log a)$ -buffer conductor in [CRVW01]

The internal expander, defined by F' , has the same size universe, left degree, and expanding set as the enclosing lossless expander, but has a smaller expansion factor and thus a smaller right side (i.e. $|V| < |V \times B|$). The B part of the output is called the buffer. Thus when we take a step in the graph the entropy that is added will be distributed between V and B depending on how lossy $G_{F'}$ is.

Permutation expander. A permutation expander is simply a permutation of the input, that is, it has right degree 1. But it has an expander graph inside it like the buffered lossless expander.

Definition 6.27 (permutation expander⁷). *A graph is an (ϵ, a) -permutation expander if it is defined by an edge function F , composed of a pair of functions $\langle F', F'' \rangle: U \times D \rightarrow V \times B$ where the output of F is a permutation over the input and F' defines a $(\frac{|V|}{d}, d, (1 - \epsilon)a)$ -expander.*

6.6.2 Performing the new Zig-Zag product

Intuition. In the following it will be beneficial to recall the intuition from the original version of the Zig-Zag product. In the new version it is not as easy to follow the intuition in terms of graphs, so we will focus on following the entropy flow.

Definition 6.28 (The new Zig-Zag product). ⁸ *Let the bipartite graph $G_E \stackrel{\text{def}}{=} Z(G_{E_1}, G_{E_2}, G_{E_3})$, with the edge function $E: U \times D \rightarrow V$, be the resulting graph from Zig-Zag'ing the three bipartite graphs G_{E_1} , G_{E_2} , and G_{E_3} . The edge functions of the three argument graphs are:*

$$\begin{aligned} E_1 \equiv (E'_1, E''_1) & : U_1 \times D_1 \rightarrow V_1 \times B_1, \\ E_2 \equiv (E'_2, E''_2) & : U_2 \times D_2 \rightarrow D_1 \times B_2, \text{ and} \\ E_3 \equiv & : U_3 \times D_3 \rightarrow V_3, \end{aligned}$$

where

$$\begin{aligned} U_3 & \equiv B_1 \times B_2, \\ U & \equiv U_1 \times U_2, \\ D & \equiv D_2 \times D_3, \text{ and} \\ V & \equiv V_1 \times V_3 \end{aligned}$$

E is then defined as follows: For any

$$\begin{aligned} x_1 & \in U_1, \\ x_2 & \in U_2, \\ r_2 & \in D_2, \text{ and} \\ r_3 & \in D_3 \end{aligned}$$

⁷ $(\epsilon, \log a)$ -permutation conductor in [CRVW01]

⁸from [CRVW01]

$$\begin{aligned}
 E(x_1 \circ x_2, r_2 \circ r_3) &\stackrel{\text{def}}{=} y_1 \circ y_2, \text{ where} \\
 (\text{Step 1:}) \quad r_1, z_2 &\stackrel{\text{def}}{=} E_2(x_2, r_2) \\
 (\text{Step 2:}) \quad y_1, z_1 &\stackrel{\text{def}}{=} E_1(x_1, r_1) \\
 (\text{Step 3:}) \quad y_2 &\stackrel{\text{def}}{=} E_3(z_1 \circ z_2, r_3)
 \end{aligned}$$

This new product manages to preserve many interesting properties of the input graphs. The result which is most interesting for us, revolves around constructing a lossless expander as shown in the theorem below. In the theorem and the following entropy flow intuition we will use the notation that for some set A its size, $|A|$, will be denoted by the corresponding small letter a .

Theorem 6.29 (Restated [CRVW01] theorem 6.2). *Let:*

$E_1 = \langle E'_1, E''_1 \rangle: U_1 \times D_1 \rightarrow V_1 \times B_1$ be a (ϵ, a_1) permutation expander

$E_2 = \langle E'_2, E''_2 \rangle: U_2 \times D_2 \rightarrow D_1 \times B_2$ be a (ϵ, a_2) buffered lossless expander

$E_3: (B_1 \times B_2) \times D_3 \rightarrow V_3$ be a $(v_3/a_3, \epsilon)$ lossless expander

Let $E: U \times D \rightarrow V$ be the Zig-Zag product of the above defined objects. If the following holds:

$$\log a_1 \geq \log(d_2 \cdot a_3 \cdot u_2 \cdot \log(1/\epsilon)/v_3)$$

$$\log v_3 \geq \log(d_1 \cdot u_1 \cdot d_2 \cdot a_3 \cdot \log(1/\epsilon) \cdot \frac{1}{v_1 \cdot a_2})$$

Then E defines an $(\frac{v_1 \cdot v_3}{a_3 \cdot a_2}, 5\epsilon)$ -lossless expander.

Proof. Due to its length the full proof has been placed in appendix B. \square

Intuitive entropy flow In order to understand why this theorem is correct we will again fall back to the intuitive entropy flow argumentation used in the discussion of the original Zig-Zag product, and pretend that a random step in each graph actually adds a specific amount of entropy to the output as compared to being within a small statistical distance of some distribution that does. We will see that under the constraints of the above theorem, the new product preserves this stronger property. We denote the inputs and output variables as they appear in Definition 6.28 and in Theorem 6.29. We will interchangeably refer to a variable, and its distribution and will so define the input variable / distribution as $x = x_1 \circ x_2$. With this notation we wish to verify that a random step on E is indeed capable of adding $\log a_3 + \log d_2$ bits of entropy if the input entropy is $\log v - \log a_3 - \log d_2$ or less.

The original Zig-Zag product could 'loose' entropy in one of its steps. The first scenario occurred in the first step when x_2 , contained so much

entropy that the first step was unable to add any more entropy to its output distribution.

In order to fix this, the idea is to replace the first graph with a buffered lossless expander G_{E_2} . The first part of the output of E_2 , denoted r_1 is similar to that of the first step in small graph in the original product. The idea is to have the output of the enclosing lossless expander E_2 be large enough so that, while the inner part of the output might be saturated, the total output will never lose any entropy.

As in the case of the original Zig-Zag product, let us consider two extreme cases for this in the following steps. As in the original Zig-Zag product, it can be shown that it is sufficient to consider these cases, since all other cases can be regarded as a convex combination of input distribution that fall into one of the extremes.

Step 1

Recall that our random input to this step is $\log d_2$ long and that the two functions comprising the buffered lossless expander is defined like this: $E'_2 : U_2 \times D_2 \rightarrow D_1$ and $E''_2 : U_2 \times D_2 \rightarrow B_2$.

Case I: $x_2|x_1$ contains less than $\log d_1 - \log a_2$ bits of entropy The similar case in the original Zig-Zag product, did not drop any entropy in this step, and the same is the case now. E_2 can add $\log a_2$ bits of entropy to the entropy contained in x_2 . Hence the entropy of the output distribution of E'_2 is $H(r_1) = H(x_2) + \log a_2$ and the entropy of the output distribution of E''_2 is $H(z_2) = \log d_2 - \log a_2$.

Case II: $x_2|x_1$ contains at least $\log d_1 - \log a_2$ bits of entropy Under these conditions the output r_1 will be (nearly) uniform, as E'_2 receives more entropy than it is able to handle losslessly. This is where the original Zig-Zag product potentially lost entropy. In the new product E_2 ensures that $r_1 \circ z_2$ contains $H(x_2) + \log d_2$ entropy. The trick is how to recover the entropy from z_2 , which is done in Step 3.

Step 2

The second problem of the original Zig-Zag product occurred when x_2 contained little entropy, and x_1 contained a lot, that is, in “Case I”. In this case the second step was not able to use the entropy in r_1 , which then contained too much entropy when we re-used it in step 3.

In the second step of the new Zig-Zag product we utilize a permutation expander. The inner graph $G_{E'_1}$ corresponds to the large graph in the original product. Since the permutation expander output is a permutation of both its inputs, it will not lose any entropy, while it may still be the case

that the inner expander receives too much entropy. Note that this permutation expander could very well be replaced with a buffered lossless expander. The reason for the use of a permutation expander is that it is much easier to obtain an explicit construction of such a graph.

Case I: Since E_1 is a permutation expander we have that $H(x_1 \circ r_1) = H(y_1 \circ z_1)$. The buffer z_1 can contain at most $\log d_1 + \log u_1 - \log v_1$ entropy, since this is its maximum size.

Since $H(x_1) + H(r_1) = H(x_1) + H(x_2) + \log a_2$ we therefore we know that $H(y_1) \geq H(x) + \log a_2 - \log d_1 - \log u_1 + \log v_1$. Note that this in itself is no improvement over the original Zig-Zag product.

Case II: We have that $H(x_1) \geq H(x) - \log u_2$ due to the length of the bit-string x_2 . The output of E'_1 can have at most $\log v_1$ entropy, giving us $H(y_1) = \min\{\log v_1, H(x) - \log u_2 + \log a_1\}$.

With these two steps we have created the first output y_1 , very similar to the first output in the original Zig-Zag product.

Step 3

In the original product r_1 acted as a buffer in the third step, but it was a buffer only for an overflow of entropy in the second step. In the new Zig-Zag product we have a combined buffer containing 'left-over' entropy from both the previous steps. The only reason this is possible is that the two extreme cases which causes overflow, are mutually exclusive (and that the remaining cases can be regarded as convex combination of the two extremes as mentioned earlier). So far we are not protected against the losses in the original zig zag product, apart from storing the 'lost' entropy in the buffers.

In order to recover the buffered entropy contained in z_2 and z_1 we use the lossless expander E_3 . If the buffers do not contain more than $\log(v_3/a_3)$ entropy conditioned on y_1 , E_3 will succeed in adding all the entropy from r_3 as well as that in the buffers to y_2 , ensuring that the resulting graph contains $H(x) + a_3 + \log d_2$ bits of entropy as required.

The remaining part of the work lies in analyzing exactly how large the third graph must be in order to be large enough to handle the buffers losslessly. The possibility of adjusting the graph used in the last step is a key benefit of adding a third graph to the inputs, instead of reusing the small graph.

Let y_1^{min} denote the minimum entropy in y_1 from the case analysis above. We have

$$y_1^{min} = \min\{\log v_1, H(X) - \log u_2 + \log a_1, H(X) + \log a_2 - \log d_1 - \log u_1 + \log v_1\}$$

We also by the conditions in the theorem have that

$$H((z_1 \circ z_2)|y_1) = H(X) + \log d_2 - H(y_1) \leq$$

$$\log\left(\frac{v}{a_2 a_3}\right) + \log d_2 - y_1^{\min}$$

The two inequalities in Theorem 6.29 have simply been chosen so they ensure that the above value stays within $\log v_3 - \log a_3$, for each of the possible values of y_1^{\min} , when $H(X) \leq \log v - \log a_2 - \log a_3$.

6.6.3 Using the Zig-Zag product

We will now use the zig-zag product to construct lossless expanders by applying Theorem 6.29 with a careful choice of parameters. In order to do this we first cover how to obtain the required input graphs. We then in Corollary 6.32 leverage internal memory to obtain somewhat unbalanced expanders with good parameters.

The graphs used The permutation expander used is an explicit construction stated in [CRVW01] and restated here as lemma 6.30. By explicit we here mean nearly free in terms of space usage. The other two graphs are constructed probabilistically and stored in main memory.

Lemma 6.30 (Restated [CRVW01] Lemma 4.4). *For every u , $a \leq u$, and $\epsilon > 0$, there is an explicit (ϵ, a) -permutation expander defined by the edge function $F : U \times D \rightarrow U \times D$ with $d = O(a^{O(1)} \cdot \frac{1}{\epsilon})$.*

Proof omitted. (see page 10 in [CRVW01]). During the proof they use eigenvalues as expansion measure. Notice that the internal expander is unbalanced.

The lossless expander We are now ready to use Theorem 6.29:

Lemma 6.31 (Restated [CRVW01] theorem 7.1.). *For every u , $t \leq u$, and $\epsilon > 0$, there exists an (n, ϵ) -lossless expander defined by the edge function $F : U \times D \rightarrow V$, where $|V| = \frac{u}{t}$, with*

- $d = \text{poly}\left(\frac{\log t + 1}{\epsilon}\right)$
- $n = O\left(\frac{u \cdot \epsilon}{t \cdot d}\right)$

And F can be computed in time $\text{poly}(\log u, \log \frac{1}{\epsilon})$ given two appropriate expanders of size $s = \text{poly}(t, \frac{1}{\epsilon})$, which can be found probabilistically in time $\text{poly}(s)$ or deterministically in time $2^{\text{poly}(s)}$.

Restated proof. Let $a_1 = t \cdot \left(\frac{\log(t)+1}{\epsilon}\right)^c$ and $u_2 = a_1^c$ where c is a constant which will be chosen later. Let $E_1, C_1 : U_1 \times D_1 \rightarrow U_1 \times D_1$ be an $(\epsilon/5, a_1)$ permutation expander with $d_1 = \text{poly}(a_1 \cdot 1/\epsilon)$ and let $u_1 = \frac{u}{u_2}$.

Let $E_2, C_2 : U_2 \times D_2 \rightarrow D_1 \times B_2$ be an $(u_2, a_2, \epsilon/5)$ buffered lossless expander with:

$$\begin{aligned} d_2 &= O(1/\epsilon^2 \cdot \log u_2) \\ a_2 &= O\left(\frac{d_2}{1/\epsilon^2}\right) \\ b_2 &= u_2 \cdot d_2 \cdot 4/\epsilon \cdot (1/d_1) \end{aligned}$$

Let $m_3 = u_2/t$ and let $E_3 : (D_1 \times B_2) \times D_3 \rightarrow M_3$ be an $(m_3/a_3, \epsilon/5)$ lossless expander where:

$$\begin{aligned} d_3 &= O(\log(d_1 \cdot b_2) \cdot 1/\epsilon) \\ a_3 &= O(d_3 \cdot 1/\epsilon) \end{aligned}$$

Let $E : U \times D \rightarrow M$ be the zigzag product of the three graphs mentioned above having:

$$\begin{aligned} d &= d_2 \cdot d_3 \\ m &= u_1 \cdot m_3 = \frac{u}{t} \end{aligned}$$

By 6.29 we have that E is an (n, ϵ) lossless expander for $n = \frac{m}{a_3 d_2}$, providing the following holds:

$$\begin{aligned} a_1 &\geq d_2 \cdot a_3 \cdot \frac{u_2}{m_3} \cdot 1/\epsilon \\ m_3 &\geq d_1 \cdot \frac{u_1}{u_1} \cdot \frac{d_2}{a_2} \cdot a_3/\epsilon \end{aligned}$$

In the first condition the right side is $\frac{u_2}{m_3} \cdot O(c/\epsilon \cdot \log t) = t \cdot O(c/\epsilon \cdot \log t)$. The left hand side is $a_1 = t \cdot \left(\frac{\log(t)+1}{\epsilon}\right)^c$, so it is satisfied for sufficiently large constant c .

In the second condition the right side is $d_1 \cdot O(c/\epsilon \cdot \log t)$. The left hand side is $m_3 = \frac{u_2}{t} = a_1^c/t$. Since $a_1 > t$ the second condition also holds for sufficiently large constant c . In case t and c is so large that $u_2 > n$, the product graph can be constructed by brute force or probabilistically while still maintaining the claim in the theorem. □

The authors of [CRVW01] used this result to construct a constant degree expander. This can be done if the variable t is small, which is the case when the graph is close to being balanced.

If we attempt to use the above Lemma in order to construct an arbitrarily unbalanced graph, we observe that t tends to u , implying an unacceptable use of space. If we restrict our attention to the case where $n = u^{1/a}$ we have that $t = \frac{u}{nd} = \frac{u^{1-1/a}}{d}$. The space usage $\text{poly}(t, 1/\epsilon)$ is obviously not within $o(n)$ for an arbitrary choice of a , but for a certain range of a it is possible for us to construct an expander that is feasible as we will see below.

Corollary 6.32 (Unbalanced expander construction). *Let $d = \text{poly}(\frac{\log u}{\epsilon})$. There exists a $(\epsilon \sqrt[b]{u}/d, \epsilon)$ -lossless expander $F : U \times [d] \rightarrow [\sqrt[b]{u}]$ constructed using $O(\sqrt[b]{u}/\epsilon^c)$ words of space and with the same computation time as the construction in Lemma 6.31, for any constant $b > 1$ and $p = \frac{1}{1-1/(bc)}$ where c is some fixed constant.*

Proof. We wish to use $O(\sqrt[b]{u}/\epsilon^c)$ space. The construction in Lemma 6.31 requires $O((t/\epsilon)^c)$ space. We wish to have a t value that is some root of u , in order to unbalance the graph as much as possible using this construction method. Therefore we choose $O((t/\epsilon)^c) = O(\sqrt[b]{u}/\epsilon^c) \implies O(t) = O(\sqrt[b]{u})$. Using this value for t we get a right side $v = u/t = O(\frac{u}{u^{1/(bc)}}) = O(u^{1-\frac{1}{bc}})$. Setting $p = 1/(1 - \frac{1}{bc})$ and observing that the remaining required properties are given by 6.31 ends the proof. Note that we can choose the constant in $O(t)$ such that v is exactly $\sqrt[b]{u}$, yielding the right side given in the corollary. \square

This corollary is not in itself of much use, yielding a feasible construction for a very restricted polynomial balance between u and n . If for example we wish to have $p = 2$ then we must also have $b \leq 2/c$ which, depending on c , might not be possible for $b \geq 1$. Even so the zig-zag product has allowed us to create a substantially unbalanced expander graph, which we improve in the next section.

6.7 Obtaining a feasible construction

We will here combine the Corollary 6.32 with a simple composition, the Telescope product, to achieve a feasible expander when there is a polynomial relation between u and n .

Corollary 6.32 does not in itself give us what we require, but we can use it in conjunction with a simple composition that allows us to utilize two slightly unbalanced lossless expanders to produce a more unbalanced lossless expander. The composition method was used on condensers by Ta-Shma et al. in [TSUZ01]. We will denote it the Telescope product.

Lemma 6.33. *(The Telescope Product) Let c_1, c_2 be constants where $c_2 \leq c_1$ and let $F_1 : U_1 \times D_1 \rightarrow V_1$ be a $(\frac{c_1 \cdot v_1}{d_1}, \epsilon_1)$ -lossless expander and let $F_2 : V_1 \times D_2 \rightarrow V_2$ be a $(\frac{c_2 \cdot v_2}{d_2}, \epsilon_2)$ -lossless expander. Then for $x_1 \in U_1, e_1 \in$*

D_1 and $e_2 \in D_2$ the graph $F_2(F_1(x_1, e_1), e_2) : U_1 \times (D_1 \times D_2) \rightarrow V_2$ is a $(\frac{c_2 \cdot v_2}{d_1 \cdot d_2}, 1 - (1 - \epsilon_1)(1 - \epsilon_2))$ -lossless expander.

Proof. Consider any set $S \subseteq U_1$ of size $s \leq \frac{c_2 \cdot v_2}{d_1 \cdot d_2}$. This set is small enough to be fully expanded by F_1 , since $\frac{c_2 \cdot v_2}{d_1 \cdot d_2} \leq \frac{c_1 \cdot v_1}{d_1}$ by the assumption that $c_2 \leq c_1$. The expansion of S hence yields a set of neighbors $V' \subseteq V_1$ of size $v' \geq (1 - \epsilon_1) \cdot \frac{c_2 \cdot v_2}{d_2}$. This set is small enough to be fully expanded by F_2 , yielding a set of neighbors $V \subseteq V_2$ of size $v \geq (1 - \epsilon_2)(1 - \epsilon_1) \cdot v_2 \cdot c_2$. \square

In the lemma above we ignored that the result of the Telescope product can be a multi-graph, i.e. there can be several edge between the same two nodes. If we remove all but one edge in each multi-edge, the degree will no longer be $d_1 d_2$, but the expansion properties will not be affected. We fix the left degree by giving every key x with multi-edges removed some replacement edges that x does not already possess in some fixed order (as we want to be able to probe them later).

We need to discover whether a key x whose edge (x, r) is being calculated has some other edge (x, r') ending in the same element. There is a simple way to handle this. When we wish to evaluate the edge $(x \in U_1, \Gamma_i(x) \in V_2)$, for some i , we simply evaluate all $d_1 \cdot d_2$ edges from x every time. In the applications of this product below, this will not affect the asymptotic complexity of calculating an edge.

We will in the proof of the following theorem use the Telescope product repeatedly on a family of variable sized expanders to produce a graph with the left side of the largest graph, the right side and expanding set size of the smallest graph, and where the degree is the product of all the degrees.

Theorem 6.34. *Let:*

- b be any constant larger than zero
- $w = \left(\frac{\log u}{\log n}\right)^{O(\log(1/\epsilon) \log_p(\log u / \log n))}$
- $d = \text{polylog}(u)^{\log_p(\log u / \log n)} \cdot w$
- $p = \frac{1}{1-1/bc}$, where c is the fixed constant from Theorem 6.32

Then there exists a $(\text{poly}(d), O(w^c \cdot \log_p(\log u / \log n) \sqrt[b]{u}))$ construction of an (n, ϵ) -lossless expander with degree d with size of the right side $O(nd)$.

Proof. Corollary 6.32 gives us a feasible construction of a lossless expander of the form $U \times [\text{poly}(\log(u)/\epsilon')] \rightarrow [\sqrt[b]{u}]$, expanding for sets up to size $\frac{\epsilon' \sqrt[b]{u}}{\text{poly}(\log(u)/\epsilon')}$ for some fixed constant $c > 1$. We will compose a number of lossless expanders of this type in the following, iteratively increasing the ratio between the left and right side.

Consider the following family of lossless expanders obtained by Corollary 6.32:

$$F_{i,\epsilon'} : [u_{i-1}] \times [\text{poly}(\log(u)/\epsilon')] \rightarrow [u_i]$$

Where $u_i = p^{i-1}\sqrt{u}$, such that each $F_{i,\epsilon'}$ is an $(\frac{\epsilon' u_i}{\text{poly}(\log(u)/\epsilon')}, \epsilon')$ -lossless expander, as defined as in Corollary 6.32 ($p = \frac{1}{1-1/(bc)}$).

Now define the repeated telescope product recursively by this function:

$$F^{(j)}(x, e_1 \circ e_2 \circ \dots \circ e_j) \stackrel{\text{def}}{=} \begin{cases} F_{j,\epsilon'}(F^{(j-1)}(x, e_1 \circ e_2 \circ \dots \circ e_{j-1}), e_j) & \text{if } j > 1 \\ F_{1,\epsilon'}(x, e_1) & \text{otherwise} \end{cases}$$

Observation 6.35. $F^{(q)} : [u] \times [d_q] \rightarrow [u_q]$ is an $(\epsilon' u_q/d_q, 1 - (1 - \epsilon')^q)$ -lossless expander, where $d_q = (\text{poly}(\log(u)/\epsilon'))^q$.

It is not hard to see from the observation above that for sufficiently high i , $F^{(i)}$ will be sufficiently unbalanced while still expanding nearly optimally (depending on the value of ϵ').

We wish to bound how high a value of i that will be required. Consider what happens as i increase. At the beginning the product graph will expand for a set that is a lot larger than we need but we will have a right side that is too large. At some point the expanding set will become too small. The construction with the exact parameters requested can be obtained by using one of the last graphs that were not unbalanced enough in combination with a graph that is less unbalanced than the ones we can obtain from Corollary 6.32 (and thus easy to obtain).

We can bound the value of i from above by the lowest k so that $u_k < n \implies p^{k-1}\sqrt{u} < n$, since in this case the expanding set must have become smaller than n since $|V|$ is smaller than n . We have $i \leq k$ and

$$k \stackrel{*}{\leq} \log_p \log_p u - \log_p \log_p n + O(1) = \log_p(\log_p u / \log_p n) + O(1) = \log_p(\log u / \log n) + O(1)$$

An explanation of (*): $\log_p \log_p(u) + O(1)$ is the number of times you need to take the p 'th root of u before you reach one. $\log_p \log_p(u) - \log_p \log_p(n) + O(1)$ is the number of times you need to take the p 'th root of u before you reach n .

We can now give a lower bound on the value of ϵ' in the component graphs when final error has to be less than the constant ϵ , we set $(1 - \epsilon')^k = (1 - \epsilon) \implies \epsilon' = 1 - \sqrt[k]{1 - \epsilon}$. Observe that $\epsilon' \geq \epsilon^k$.

We can now determine an upper bound on degree achieved in this construction. We have

$$\begin{aligned} d_k &\leq \text{poly}(\log(u)/\epsilon')^k = \text{polylog}(u)^k \cdot \text{poly}(1/\epsilon')^k \\ &\leq \text{polylog}(u)^k \cdot \text{poly}(1/\epsilon^k)^k = \text{polylog}(u)^k \cdot \text{poly}(1/\epsilon)^{k^2} \\ &\leq \text{polylog}(u)^{\log_p(\log u / \log n)} \cdot \text{poly}(1/\epsilon)^{\log_p^2(\log u / \log n)} \\ &= \text{polylog}(u)^{\log_p(\log u / \log n)} \cdot 2^{O(\log(1/\epsilon)) \log_p^2(\log u / \log n)} \\ &\leq \text{polylog}(u)^{\log_p(\log u / \log n)} \cdot \left(\frac{\log u}{\log n}\right)^{O(\log(1/\epsilon) \log_p(\log u / \log n))} \end{aligned}$$

Finally we can now bound the size $|V|$ of the right side of the final product. Since we have exaggerated the degree and the number of steps we have that $n = \Omega(\epsilon'|V|/d_k) \implies |V| = O(\frac{1}{\epsilon'}nd_k)$. Since ϵ' is not a constant, we will not get a completely optimal expansion. However $\frac{1}{\epsilon'} \leq \frac{1}{\epsilon^k} \leq \left(\frac{\log u}{\log n}\right)^{O(\log(1/\epsilon)\log_p(\log u/\log n))}$, which disappears in the poly notation for d_k giving us

$$|V| = O\left(n \cdot \text{polylog}(u)^{\log_p(\log u/\log n)} \cdot \left(\frac{\log u}{\log n}\right)^{O(\log(1/\epsilon)\log_p(\log u/\log n))}\right)$$

This factor could have been removed by using a constant error on the last expander graph, but to keep the analysis simple we do not do that.

This gives us a (n, ϵ) -lossless expander with a degree of d_k which equals the d stated in the theorem while using $O(\log_p(\log u/\log n) \sqrt[b]{u}/\epsilon')$ words of internal memory for any constant $b > 0$. The space usage is achieved by noting that we use $\log_p(\log u/\log n)$ different expanders from the family F , and by Corollary 6.32 each of them requires at most space $O(\sqrt[b]{u}/\epsilon^c)$. \square

One may wonder if it is possible to apply the same technique for non-constant values of b , possibly reducing the space to $o(n)$. As b becomes smaller than any constant value, the graph obtained from 6.32 will quickly tend to only being unbalanced by constant factor. In this case the number of iterations required in the above composition will be $O(\log u/n)$, yielding a degree that is much worse than that of the currently best constructions.

For the interesting case where $u = \text{poly}(n)$ Theorem 6.34 yields a very good feasible construction of an expander:

Corollary 6.36. *Assume $u = O(n^a)$ where a is a constant and let:*

- b' be any constant larger than zero
- $w = a^{O(\log(1/\epsilon)\log_p(a))}$
- $d \leq \text{polylog}(u)^{O(\log_p(a))} \cdot w$
- $p = \frac{1}{1-1/abc}$, where c is as in Theorem 6.32

Then there exists a $(\text{poly}(d), O(w^c \cdot \log_p(a) \sqrt[b']{n}))$ construction of an (n, ϵ) -lossless expander with size of the right side $O(nd)$.

Proof. Set $b = ab'$ in Theorem 6.34 and observe that $\log_p(\log u/\log n) = \log_p a$. \square

This an important result, as it allows us to construct an expander with much better degree than what is possible in the current explicit constructions. Furthermore this is done using an acceptable amount of space, enabling us to implement our parallel one probe data structure efficiently when $u = \text{poly}(n)$.

Comparing the new construction to the Ta-Shma result

For polynomial differences in u and n the new construction is a considerable improvement over the Ta-Shma construction from Theorem 6.22 when comparing the important factors of degree size and right side size. Both in the Ta-Shma construction and the construction from Theorem 6.34 the right side size is asymptotically equal to the degree times n so we will just compare the degrees. Suppose $u = n^a$, where a is some constant. The Ta-Shma degree is then:

$$(O(\log n^a))^{O(\log \log n \cdot \log \log n^a)} = (O(a \log n))^{O(\log \log n \cdot a \log \log n)} = (O(\log n))^{O((\log \log n)^2)}$$

While the degree of the construction from Theorem 6.34 is, for some constant $k > 1$, the considerable better:

$$O(\log^k(n^a)) = O((\log n)^k)$$

6.8 Conclusion

In this chapter we have covered various expander construction methods. In doing this we have showed that if we are willing to use space sub-linear in the expanding set, it is possible to construct an expander graph with left degree polylogarithmic in u by combining the improved zig-zag product with a simple composition, while still maintaining the standard polylogarithmic time to calculate an edge. We have seen that this construction is considerably better than the best known general explicit construction when $u = \text{poly}(n)$.

The result immediately applies in an external memory model since it is here reasonable to assume we have some limited amount of internal memory accessible 'for free'. This means that the results in the chapter concerning the parallel disk models are substantially more applicable in practice. The result could also be used in a single disk external memory version of the original OPS data structure or in any other external memory data structure using unbalanced lossless expander graphs.

Chapter 7

Conclusion

Our original main goals for this thesis was to show that any one probe scheme with the properties of the original OPS data structure was optimal in terms of space usage, and to consider the possibility of reducing the inherent redundancy of the structure by adapting it for use in parallel external memory. We have by using a simple information theoretic argument shown that the original OPS data structure is indeed optimal in terms of space, achieving our first goal.

In considering the second goal we have constructed various parallel dictionaries with the shared property that they all in a single parallel probe can retrieve the satellite data of a given key present in the dictionary with bandwidth efficiency a constant fraction of optimal. In the static case we have obtained dictionaries that use space asymptotically equivalent to storing the key and value pairs in a table. For most reasonable sizes of satellite data we have achieved a similar result for the dynamic case.

In order to enable these data structures to be efficiently implemented we have given a way to construct the necessary expander graphs in the case where some reasonable amount of space is available to store it and the universe of keys is of size polynomial in the number of inserted keys. This is a very interesting trade-off between the degree of the expander graph and the space used to store its representation. Since the degree has direct implications for the size of the right side of the bipartite expander graph, where we store the data of our dictionary, this result immediately applies in external memory expander based dictionaries. The degree it is possible to achieve by using this trade-off is a significant improvement over the current explicit constructions.

To see a compact listing of the exact results refer to the summary at the beginning of this report.

Appendix A

Notation

$\Gamma(\cdot)$	The neighbor function mapping a key $x \in U$ to the neighbor elements in V or mapping a set of elements $X \in U$ to $\cup_{x \in X} \Gamma(x)$
$\Gamma_{asn}(\cdot)$	The elements in $\Gamma(x)$ that is assigned to x
$\Gamma_i(x)$	The i 'th neighbor of x , assuming a fixed ordering of the Γ -function.
OPS	One Probe Search, used frequently in the phase 'the original OPS data structure', meaning the result from [OP02].
U	The universe of a dictionary and / or the left node side of a bipartite expander graph. The elements are called keys. The keys are bit-strings.
u	The size of the set U .
S	The set of inserted keys in a dictionary.
n	The maximum size of S in some dictionary and / or the maximum size of expanding set in an expander graph.
V	The set of positions in memory that some dictionary can probe and / or the set of right side elements of a bipartite expander graph.
T_V	An ordered table of V , i.e. the random access memory of some dictionary.
d	The left degree of some bipartite expander graph and / or the number of disk heads or disks in some external memory dictionary.
D	The set of bit-strings with numerically value 0 to $\lceil \log d \rceil$.

ϵ	The error rate in the lookup function of a dictionary and / or the error rate in the expansion factor of an expander.
ϵ -ghost	See definition 1.7.
\tilde{S}	The set of ϵ -ghosts for an inserted set S .
\tilde{S}	In some dynamic expander based dictionaries this is the set of keys we currently believe to be the ϵ -ghosts for the inserted set S .
\perp	Special character to write in an element of T_V when the answer to probing this element should be 'no'.
β	The size of the satellite data for some inserted key. This value is assumed to be of equal size for all inserted keys.
<i>PDHM</i>	The Parallel Disk Head Model. See definition 3.1
<i>PDM</i>	The Parallel Disk Model. See definition 3.2
$\log x$	$= \log_2 x$
$[z]$	The set of bit-strings with the numerical value from 0 to $z - 1$, both inclusive.
$a \circ b$	The concatenated bit-string of the strings a and b .
$G_F(U, V, E)$	A graph defined by the edge function $F : U \times D \rightarrow V$, for some left degree $ D $.
$X(S)$	Denotes some distribution over the set S .
$X(z)$	$\Pr[X = z]$, where z is an elements in the probability distribution X .
$\Upsilon(S)$	Denotes the uniform distribution over the set S .
$H(X)$	The entropy of the distribution X . See definition 6.3.
$H_\infty(X)$	The min-entropy of the distribution X . See definition 6.4.
(k, ϵ) -source	See definition 6.7.
$X_1 X_2$	The random variable X_1 conditioned on the random variable X_2
$\langle F_1, F_2 \rangle$	A function consisting of a pair of functions. If $F_1 : A \times B \rightarrow C$ and $F_2 : A \times B \rightarrow D$, then $\langle F_1, F_2 \rangle : A \times B \rightarrow C \times D$

Appendix B

The zig-zag theorem

Theorem B.1 (Restated [CRVW01] theorem 6.2). *Let:*

$F_1 = E_1, C_1 : U_1 \times D_1 \rightarrow M_1 \times B_1$ be a (ϵ, a_1) permutation expander

$F_2 = E_2, C_2 : U_2 \times D_2 \rightarrow D_1 \times B_2$ be a (ϵ, a_2) buffer expander

$F_3 : (B_1 \times B_2) \times D_3 \rightarrow M_3$ be a $(m_3/d_3, \epsilon)$ lossless expander

Let $F : U \times D \rightarrow V$ be the zigzag product of the above defined objects. If the following holds:

$$\log a_1 \geq \log(d_2 \cdot a_3 \cdot u_2 / (\epsilon m_3))$$

$$\log m_3 \geq \log(d_1 \cdot u_1 \cdot d_2 \cdot d_3 \cdot \frac{1}{\epsilon \cdot m_1 \cdot a_2})$$

Then E defines an $(\frac{m_1 \cdot m_3}{a_3 \cdot a_2}, 5\epsilon)$ -lossless expander.

Restated and completed proof. For input values $x \in U$, $r_2 \in D_2$ and $r_3 \in D_3$ the computation $F(x, r_2 \circ r_3)$ produces the intermediate values $x_1, x_2, r_1, z_1, y_1, z_2$ and y_2 , we will let the correspond capital lettered variables indicate the corresponding random variables in the computation of $F(X, R_2 \circ R_3)$. Here we will assume that X is a k -source over U with $k \leq \log(M/a)$, R_2 is uniformly distributed on D_2 and R_3 uniformly distributed on D_3 . Our final goal is to prove that (Y_1, Y_2) is a $(k + \log d, \epsilon)$ source.

We will first show that Y_1 contains enough entropy. In order to do this we need the following Lemma:

Lemma B.2. *Let X be any k -source then X is ϵ -close to the convex combination $\alpha(X'_1, X'_2) + \beta(X''_1, X''_2)$ of two k' sources where:*

- 1 $k' = k - \log(1/\epsilon)$

- 2 For every x_1 in the support of $X'_1, H_\infty(X'_2 | X'_1 = x_1) < \log d_1 - \log a_2$

- 3 For every x_1 in the support of $X''_1, H_\infty(X''_2 | X''_1 = x_1) \geq \log d_1 - \log a_2$

Proof. Let B denote the set of string in the support of X'_1 such that $H_\infty(X'_2|X'_1 = x_1) < \log d_1 - \log a_2$. Define the component sources as :

$$(X'_1, X'_2) \stackrel{\text{def}}{=} (X_1, X_2)|X_1 \in B$$

$$(X''_1, X''_2) \stackrel{\text{def}}{=} (X_1, X_2)|X_1 \notin B$$

Define:

$$\alpha \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } \Pr[X_1 \in B] < \epsilon \\ 1 & \text{if } \Pr[X_1 \in B] > 1 - \epsilon \\ \Pr[X_1 \in B] & \text{otherwise} \end{cases}$$

and define $\beta = (1 - \alpha)$. These definitions trivially satisfy 2 and 3 above.

It follows easily that X is ϵ -close to $\alpha(X'_1, X'_2) + \beta(X''_1, X''_2)$. Finally its not hard to verify that if $\alpha > 0$ then (X'_1, X'_2) is a k' -source and if $\beta > 0$ then (X''_1, X''_2) is a k' -source. \square

This Lemma allows us to reason about the distribution Y_1 by only considering the two extreme cases listed above as 2 and 3. The two extreme cases are covered in the following two lemmas:

Lemma B.3 (Case I). *Assume that X is a k' source (for some k') and that for every x_1 in the support of X_1 , $H_\infty(X_2|X_1 = x_1) < \log d_1 - \log a_2$. Then Y_1 is a $(k' + \log(a_2 \cdot m_1 / (d_1 \cdot u_1)))$ -source.*

Proof. For any x_1 in the support of X_1 define $k_1 = H_\infty(X_2|X_1 = x_1)$. Since for every such x_1 we have that $k_1 < \log d_1 - \log a_2$ we can conclude that from the properties of E_2 that $(R_1|X_1 = x_1) = E_2((X_2|X_1 = x_1), R_2)$ is a $(k_1 + \log a_2, \epsilon)$ -source. Since for every x_1 as above, we have that $k_1 \geq k' - \log \frac{1}{\Pr[X_1 = x_1]}$, it is not hard to argue that (X_1, R_1) is a $(k' + \log a_2)$ -source. Since E_1, C_1 is a permutation we also have that its output (Y_1, Z_2) is a $(k' + \log a_2)$ -source. Since Z_2 is a distribution over a $\log b_1 = \log(d_1 \cdot u_1 / m_1)$ long bit-string, Y_1 must be at least a $(k' + \log \frac{a_2 \cdot m_1}{d_1 u_1})$ -source, and therefore also a $(k' + \log \frac{a_2 \cdot m_1}{d_1 u_1}, \epsilon)$ -source \square

Lemma B.4 (Case II). *Assume that X is a k' source (for some k') and that for every x_1 in the support of X_1 , $H_\infty(X_2|X_1 = x_1) \geq \log d_1 - \log a_2$. Then Y_1 is a $\min\{\log m_1, k' + \log(a_1 / u_2)\}$ -source.*

Proof. Since for every x_1 in the support of X_1 where $H_\infty(X_2|X_1 = x_1)$ we can conclude from the properties of E_2 that for every such x_1 , $(R_1|X_1 = x_1) = E_2((X_2|X_1 = x_1), R_2)$ is ϵ -close to uniform. This implies that (X_1, R_1) is ϵ -close to $(X_1, \Upsilon(D_1))$ and thus that $Y_1 = E_1(X_1, R_1)$ is also ϵ -close to $E_1(X_1, \Upsilon(D_1))$. Since (X_1, X_2) is a k' -source and X_2 is distributed over $\log u_1$ -bit strings we can deduce that X_1 is a $(k' - \log u_2)$ -source. Since E_1 is an $(\epsilon, \log a_1)$ -extractor we have that $E_1(X_1, \Upsilon(D_1))$ is a

$(\min\{\log(m_1), k' - \log n_2 + \log a_1\}, \epsilon)$ -source. Finally we can conclude that Y_1 is a $(\min\{\log(m_1), k' - \log u_2 + \log a_1\}, 2\epsilon)$ -source. \square

We can now prove that Y_1 always contains enough entropy by combining the above three lemmas into the following:

Lemma B.5. Y_1 is a $(k_{out}, 3\epsilon)$ -source where

$$k_{out} = \min\{\log m_1, k + \log\left(\frac{a_1\epsilon}{u_2}\right), k + \log\left(\frac{\epsilon a_2 m_1}{d_1 u_1}\right)\}$$

Proof. From the previous three lemmas we can conclude that Y_1 is 3ϵ -close to a convex combination $\alpha Y' + \beta Y''$ where Y' is a $(k' + \log \frac{a_2 \cdot m_1}{d_1 u_1} - \log 1/\epsilon)$ -source and Y'' is a $\min\{\log m_1, (\log(k' \cdot a_1/u_1))\}$. Therefore both Y' and Y'' are k_{out} sources and so is their convex combination $\alpha Y' + \beta Y''$. \square

In the next step of the proof we will show that Y_1 together with the two buffers is a $(k + \log d_2, \epsilon)$ -source.

Lemma B.6. (Y_1, Z_1, Z_2) is $(k + \log d_2, \epsilon)$ -source.

Proof. For any x_1 in the support of X_1 , define $k_1 = H_\infty(X_2|X_1 = x_1)$. Since E_2, C_2 is a (u_2, ϵ) lossless expander we have that for every such x_1 , the conditional distribution $((R_1, Z_1)|X_1 = x_1) = E_2, C_2((X_2|X_1 = x_1), R_2)$ is a $(k_1 + \log d_2, \epsilon)$ -source. For every x_1 as above we have that $k_1 \geq k - \log \frac{1}{\Pr[X_1=x_1]}$ and it follows that (X_1, R_1, Z_1) is a $(k + \log d_2, \epsilon)$ -source. Because E_1, C_1 is a permutation we have that (Y_1, Z_2, Z_1) is also a $(k + \log d_2)$ -source. Note that it is essential that E_1, C_1 is one-to-one, as R_1 is not necessarily random. \square

Combining this with Lemma B.5 giving the entropy of Y_1 and Lemma B.6 giving the entropy of Y_1 combined with the buffers, we can now deduce the following:

Lemma B.7. (Y_1, Z_1, Z_2) is 4ϵ -close to a source (Y'_1, Z'_1, Z'_2) where

Y'_1 is a k_{out} -source

(Y'_1, Z'_1, Z'_2) is a $k + \log d_2$ -source.

In order to complete the proof, we recall the following conditions from the theorem:

$$\log a_1 \geq \log(d_2 \cdot a_3 \cdot u_2 / (\epsilon m_3))$$

$$\log m_3 \geq \log(d_1 \cdot u_1 \cdot d_2 \cdot d_3 \cdot \frac{1}{\epsilon \cdot m_1 \cdot a_2})$$

We wish to prove that the product is a lossless expander for input that is a $\log \frac{m}{d_2 d_3}$ -source, so assume $k \leq \log \frac{m}{d_2 d_3}$.

Under these condition the following is the case:

$$\begin{aligned} k_{out} &= \min\{\log m_1, \log\left(\frac{ka_1\epsilon}{u_2}\right), \log\left(\frac{ka_2m_1}{d_1u_1}\right)\} \\ &\geq \min\{\log m_1, \log \frac{k \cdot d_2}{m_3 d_3}\} \end{aligned}$$

Now, since $k \leq \frac{m}{d_2 d_3} = \frac{m_1 m_3}{d_2 d_3}$ we can conclude that Y'_1 is a $(\log(\frac{kd_2 d_3}{m_3}))$ -source.

For y in the support of Y'_1 define $h_z = H_\infty(Z'_1, Z'_2 | Y'_1 = y)$. h_z expresses the 'randomness' in the buffers that is not dependent on the randomness already counted in Y'_1 .

Now consider the case that $h_z > m_3/d_3$, in which case the lossless expander F_3 cannot expand optimally, we do get that $(Y'_2 | Y'_1 = y)$ is ϵ -close to uniform over M_3 and is thus an $(\log m_3, \epsilon)$ -source. If $h_z \leq m_3/d_3$, then F_3 will output a distribution that is a $(h_z + \log d_3, \epsilon)$ -source.

Adding this up with Y'_1 we get that (Y'_1, Y'_2) is a $(\min\{\log H_\infty(Y'_1) + \log m_3, \log m_3 + \log d_3\}, \epsilon)$ -source. The last part comes from noting that $H_\infty(Y'_1, Y'_2) \geq H_\infty(Y'_1) + h_z + \log d_3 - \log 1/\epsilon = H_\infty(Y'_1, Z'_1, Z'_2) + \log d_3 - \log 1/\epsilon$.

For the first part of the min expression above, recall that $H_\infty(Y'_1)$ is a $(\log(\frac{kd_2 d_3}{m_3}))$ -source and that $(\log k - \log m_3 + \log d_2 + \log d_3) + \log m_3 = \log k + \log d$ as required. For the other part of the min expression, just note that $H_\infty(Y'_1) + m_3 = \log k + \log d_2 + \log d_3 = \log k + \log d$ as required. It now follows that $H_\infty(Y'_1, Y'_2)$ is a $(\log k + \log d, \epsilon)$ -source. \square

Bibliography

- [AL96] Alon and Luby. A linear time erasure-resilient code with nearly optimal recovery. *IEEE TIT: IEEE Transactions on Information Theory*, 42, 1996.
- [BKV⁺81] M. Blum, R. M. Karp, O. Vornberger, C. H. Papadimitriou, and M. Yannakakis. The complexity of testing whether a graph is a superconcentrator. *Inf. Process. Lett.*, 13(3):164–167, 1981.
- [BMRV00] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? pages 449–458, 2000.
- [CRVW01] Michael R. Capalbo, Omer Reingold, Salil P. Vadhan, and Avi Wigderson. Randomness conductors and constant-degree expansion beyond the degree $\sqrt{2}$ barrier. 2001.
- [CRVW02] Michael R. Capalbo, Omer Reingold, Salil P. Vadhan, and Avi Wigderson. Randomness conductors and constant-degree lossless expanders. In *STOC*, pages 659–668, 2002.
- [CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
- [DadH90] M. Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 6–19, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
- [GI02] Venkatesan Guruswami and Piotr Indyk. Near-optimal linear-time codes for unique decoding and new list-decodable codes over smaller alphabets. In *STOC '02: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, pages 812–821, New York, NY, USA, 2002. ACM Press.
- [HMP01] Torben Hagerup, Peter Bro Miltersen, and Rasmus Pagh. Deterministic dictionaries. *J. Algorithms*, 41(1):69–85, 2001.

- [JP05] Morten Skaarup Jensen and Rasmus Pagh. Optimality in external memory hashing. 2005.
- [Kah95] Nabil Kahale. Eigenvalues and expansion of regular graphs. *J. ACM*, 42(5):1091–1106, 1995.
- [Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.
- [KR04] V. Kabanets and A. Rafiey. Cmpt 881: Pseudorandomness lecture notes, lecture 8: Spectral expansion, 2004.
- [OP02] Anna Östlin and Rasmus Pagh. One probe search. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 439–450. 2002.
- [Pag99] Rasmus Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In *WADS '99: Proceedings of the 6th International Workshop on Algorithms and Data Structures*, pages 49–54, London, UK, 1999. Springer-Verlag.
- [RRV99] Ran Raz, Omer Reingold, and Salil Vadhan. Extracting all the randomness and reducing the error in Trevisan’s extractors. pages 149–158, 1999.
- [RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. 8(2):300–304, June 1960.
- [RVW01] Omer Reingold, Salil P. Vadhan, and Avi Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders. 2001.
- [SEK00] Peter Sanders, Sebastian Egner, and Jan H. M. Korst. Fast concurrent access to parallel disks. In *Symposium on Discrete Algorithms*, pages 849–858, 2000.
- [Sie88] Alan Siegel. On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. *FOCS*, pages 20–25, 1988.
- [Sie04] Alan Siegel. On universal classes of extremely random constant-time hash functions. *SIAM J. Comput.*, 33(3):505–543, 2004.
- [Spr77] Renzo Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Commun. ACM*, 20(11):841–850, 1977.

- [TS02] Amnon Ta-Shma. Storing information with extractors. *Inf. Process. Lett.*, 83(5):267–274, 2002.
- [TSUZ01] Amnon Ta-Shma, Christopher Umans, and David Zuckerman. Loss-less condensers, unbalanced expanders, and extractors. In *STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 143–152, New York, NY, USA, 2001. ACM Press.
- [Yao81] Andrew Chi-Chih Yao. Should tables be sorted? *J. ACM*, 28(3):615–628, 1981.