Oblivious tight compaction in $O(n)$ time with smaller constant

Samuel Dittmer* Rafail Ostrovsky†

April 9, 2020

Abstract

Oblivious compaction is a crucial building block for hash-based oblivious RAM. Asharov et al. recently gave a $O(n)$ algorithm for oblivious tight compaction. Their algorithm is deterministic and asymptotically optimal, but it is not practical to implement because the implied constant is $\gg 2^{38}$. We give a new algorithm for oblivious tight compaction that runs in time $< 16014.54n$. As part of our construction, we give a new result in the bootstrap percolation of random regular graphs.

1 Introduction

1.1 Oblivious RAM

Oblivious RAM has been a critical object of study in cryptography since its introduction by Goldreich and Ostrovksy [13]. ORAM now arises in a number of contexts such as improving security on SGX chips [9], garbled RAM programs [18] and cloud computing [31]. Many implementations and variations of ORAM exist, including distributed ORAM, where multiple clients jointly access a shared array (see e.g. [6, 17]), parallel ORAM (see e.g. [22]). Many variations have explored optimizations with larger block sizes or client memory, such as the well-known Path ORAM [28], which has blocks of size $\Omega(\log^2 n)$ and $O(\log n) \cdot \omega(1)$ client storage (in blocks).

In this paper, we consider the classic ORAM model, with a single server, a single client, blocks of size $O(\log n)$, and $O(1)$ blocks of client storage. Recently Asharov et al. [1] gave the first ever asymptotically optimal ORAM construction under these conditions, with $O(\log n)$ amortized overhead, which they called OptORAMa. A key building block in their construction is an oblivious tight compaction algorithm with $O(n)$ runtime.

OptORAMa is noteworthy for being asymptotically optimal and deterministic, but it is not intended for practical applications. The implied constant in the big-O notation is enormous, $\gg 2^{38}$.

We construct a new oblivious tight compaction algorithm that matches the asymptotics of OptORAMa and makes a substantial improvement to the constant. We do this in part by removing the requirement that the algorithm be deterministic, and by analyzing the behavior of expander graphs under majority bootstrap percolation. Our algorithm requires $16014.54n$ steps to compact an array of length n with negligible failure probability.

*Stealth Software Technologies Inc. samuel.dittmer@gmail.com

†University of California, Los Angeles. Department of Computer Science and Mathematics. rafail@cs.ucla.edu

1.2 Main results

Our main theorem is our oblivious tight compaction algorithm:

Theorem 1. *There is an algorithm to obliviously tightly compact an array in* $16014.54n$ *time.*

To demonstrate the correctness of our algorithm, we need the following new result on bootstrap percolation, which we prove in Appendix D. The proof follows quickly from the expander mixing lemma and is similar to the proof of Lemma 10 in [8].

Theorem 2. *Let G be a d-regular Ramanujan graph. For $d \geq 13$ odd and $d \geq 16$ even there is some constant $c < 1$ such that every subset $S \subseteq G$ with $|S|/|G| \geq c$ is a percolating set for majority bootstrap percolation.*

1.3 Bootstrap percolation

Bootstrap percolation of a graph is type of cellular automaton where, at each time step, a vertex's behavior is determined by the behavior of its neighbors. It is an area of extensive research in computer science and combinatorics, and also arises in areas from physics studies of the Bethe lattice [7] and Ising model [27] to the study of social networks [16, 10]. One prominent area of research is on the size of percolating sets, which are sets that, if activated initially, will cause the entire graph to be activated.

One common way to phrase this question is to ask when choosing vertices from a graph with probability p gives a percolating set with high probability (see e.g. [2], [5], [30]). In the case of random regular graphs, Balogh and Pittel compute the p^* at which a transition from probability almost zero to probability almost one occurs [4]. We use their result in the full version of our algorithm given in Appendix A. However, we also desire bounds on initial sets above which percolation to the entire graph is guaranteed, rather than occurring with high probability. We give such bounds in Theorem 2 and prove them in Appendix D.

There has been broad interest in the question of the smallest possible percolating set for many different families of graphs, including expander graphs [8], regular graphs [14], d-dimensional hypercubes [20, 21], and powers of complete graphs [3]. Coja-Oghlan, Feige, Krivelevich and Reichman studied the size of the *largest* minimal percolating set for expander graphs [8] in the case of a constant size activation threshold. This question has also been studied for the grid [20], hypercubes [24] and trees [25].

We are interested in majority bootstrap percolation, where a vertex becomes active if a majority of its neighbors are active. Gärtner and Zehmakan gave a result equivalent to establishing an upper bound on the size of the largest minimal percolating set for expander graphs of sufficiently large degree in [12], (see also [32]) but their result is for two-way majority bootstrap percolation, a different cellular automaton.

1.4 Comparison to OptORAMa

To clearly present both the similarities to and the improvements over OptORAMa, we describe below several of the key features of OptORAMa that facilitate its asymptotic optimality and drive its large runtime. We also describe the approach our algorithm takes to address the same problems.

1. In OptoRAMa, large arrays are divided into smaller bins, where all but a constant fraction of bins have their density of marked elements bounded above by some constant. For large enough arrays (compared to W, the word size), this subdivision step happens twice. Bins are

first chosen to have size less than $(W/\log W)^2$, and then those bins are further subdivided into bins of size less than $W/\log W$. These bins are further divided into constant size buckets. This double division causes the initial density requirement to be $\rho < 1/2^{38}$.

In our algorithm, we likewise divide large arrays into much smaller bins. However, we avoid making subdivisions of already subdivided bins. We use bins of size $O(W)$, with an explicit constant we give later.

This improvement is made possible by various bin distribution schemes that can produce overly dense bins with probability $1/\log^a(n)$ or $1/n^c$. We then collect the elements from the dense bins into another large runoff array, which we must also compact.

2. In OptORAMa, private data about small bins are packed into $O(1)$ memory words, so that this data can be accessed obliviously in $O(1)$ time. We use the same bitpacking schemes in a similar way.

3. In OptORAMa, expander graphs are used to simulate random access. This is used in particular for their LooseSwapMisplaced algorithm, which controls density of marked elements. Our LooseSwap algorithm is essentially the "re-randomised" version of OptORAMa's "derandomised" LooseSwapMisplaced algorithm.

   Note that the runtime analysis in § E.3 also gives a lower bound on the runtime of their LooseSwapMisplaced. The cost in OptORAMa of pushing out-of-place balls below $1/2^{38}$ is thus bounded below by the cost of $\textsc{LooseSwap}(A, 1/2, 1/2, 1/2^{38})$, which is $2^{38}|A|$. This gives our very rough lower bound on the runtime of OptORAMa.

4. In OptORAMa, expander graphs are also used as a building block for linear superconcentrators on small bins. OptoRAMa adapts the non-oblivious Pippenger construction [23] that builds from a bipartite expander graph.

   We replace linear superconcentrators with bootstrap percolation on expander graphs. This allows to distribute balls evenly between buckets without forcing us to designate a certain set of vertices as "input" or "output" vertices.

2 Structure of paper

We begin by describing, in Sections 3 and 4, we describe a construction that has $O(n)$ runtime and $O(\log\log n)$ client memory. To improve to the standard $O(1)$ client memory, we give a modified version of our algorithm which reduces the density of marked elements by a factor of $\log n$. Running LooseCompact with density γ and word size $W = \log n \log\log n$ is equivalent to running LooseCompact with density $\gamma/\log n$ and word size $W = \log n$. We describe the details in Appendix A.

We follow [1] in using *tight compaction* to refer to moving all marked elements to the left of all unmarked elements, and *loose compaction* to refer to moving all marked elements into an array whose size is decreased by some constant fraction. As in [1], we note that tight compaction can be reduced to loose compaction.

Our loose compaction algorithm uses an involved construction to control the density of marked elements on part of the array. It then shuffles elements from that part of the array into bins of size $\Theta(\log n)$, i.e. proportional to $W/\log\log W$. We then run BinCompact to loosely compact each bin. This produces a sparse runoff array and a discard array populated with unmarked and dummy elements. The sparse runoff array we compact with a modified bin compaction algorithm BinCompactMargulis. The discard array we compact by running TightCompact again, since the discard array produced in this iteration is made up entirely of dummy elements. We give a more detailed description of how these algorithms interact in Section 3.

In the algorithm BinCompact, we shuffle each bin into buckets of constant size, and identify each bucket with the vertex of an expander graph G. Via bitpacking, we can store the private data of G in a single memory word.

By treating empty buckets as active in the sense of majority bootstrap percolation, and applying Theorem 2, we assign weights to edges of G, so that every non-empty vertex has a majority of edges pointing away from it. This algorithm is essentially nonoblivious, since we can analyze the private data of G locally. Once the edge weights are set, we perform oblivious swaps in the directions indicated by the edge weights. We give a more detailed description of BinCompact in Section 4.

For sparse arrays with density bounded above by n^{c-1}, we use a variant of BinCompact built on Margulis graphs that we call BinCompactMargulis. We use here a tree traversal algorithm that is similar to bootstrap percolation in its effects. We give the details in Appendix B.

We defer proofs of a number of lemmas to Appendix C. The proof of Theorem 2 is given in Appendix D. A careful accounting of runtime cost is given in Appendix E.

3 Compaction algorithms

Algorithm 1 Oblivious tight compaction

```
Require: Array $A$ of size $n$ with some subset of elements marked
Ensure: Obliviously shuffle $A$ so that all marked elements are at the front of the array
 1: procedure TightCompact($A$)
 2:     $s \leftarrow 0$
 3:     if More than half of $A$ is marked then
 4:         $s \leftarrow 1$
 5:         Switch all markings
 6:     end if
 7:     Temporarily mark additional elements until exactly half of $A$ is marked
 8:     TightCompactByTwo($A$)
 9:     Unmark the temporarily marked elements
10:     TightCompact($A[1 : \frac{n}{2}]$)
11:     if $s = 1$ then
12:         Swap element in position $i$ with position $n + 1 - i$
13:         Unswitch all markings
14:     else
15:         Perform dummy access to positions $i$ and $n + 1 - i$
16:     end if
17: end procedure
```

3.1 Analysis of Algorithm 1

The purpose of this algorithm is to reduce tight compaction to the case where exactly half of the elements are marked. First, we observe that compressing marked elements to the left side of an array is equivalent to compressing unmarked elements to the right half. The formal implementation of this observation is given in lines 1-5 and lines 10-15.

Next, assume less than or equal to half of the elements are marked. If we can loosely compact those elements into the left half of the array, then, by induction we can tightly compact the left half of the array. Equivalently, we mark additional elements until exactly half of the elements are marked, and then compact those elements tightly into the left half. The formal implementation is given in lines 6-9.

The correctness of this algorithm is an immediate consequence of induction and the correctness of TightCompactByTwo, that is, Algorithm 2 shown in § 3.2.

In § E.2, we show the runtime of this algorithm is $4621.6n$.

3.2 Analysis of Algorithm 2

In TightCompactByTwo we introduce a number of constants, to be optimized later. The constant α represents the fraction of the array on which we will perform LooseCompact, while β, γ, and γ' are bounds on density of marked elements in subarrays. The degree of the expander graph is given by d, and the word size is $W = \Theta(\log n \log\log n)$. We determine in § 4.2 that we must have $d \geq 13$ to get the desired expansion.

The purpose of TightCompactByTwo is to isolate a subarray $A_1 \subseteq I$ whose density of marked elements is very low. We then call LooseCompact on A_1, which in turn calls BinCompact. We require A_1 to have density $\leq \gamma'$, which ensures that all but n^c of the bins created in LooseCompact will have density $\leq \gamma$, for some constant $c < 1$ (where c is a function of γ, γ' and W; see Lemma C.6).

The algorithm relies on *exactly* half of the elements being marked. This allows us to divide the array in half and control the density in both halves. We write $I = A \cup \tilde{A}$, and treat $\tilde{A}$ as the mirror of A, reversing the role of marked and unmarked balls. To begin, we use SwapMisplaced to force the density of A to be less than β, which pushes the density of $\tilde{A}$ above $1-\beta$.

We divide A into $A = A_1 \cup A_2$, with $|A_1| = \alpha|A|$, and likewise divide $\tilde{A} = \tilde{A}_1 \cup \tilde{A}_2$. We push the density of A_2 below γ' by runing SwapMisplaced on A_1 and A_2, and likewise push the density of $\tilde{A}_2$ above γ'. We have now increased the density of A_1 and decreased the density of $\tilde{A}_1$, so we use SwapMisplaced to push their densities back to β and $1-\beta$.

We can now run LooseCompact on A_1 and $\tilde{A}_1$. This compacts A_1 by a factor of $\frac{d}{d+1}$, and so increases the density of A_1 by a factor of $(1+\frac{1}{d})$. It compacts A by a factor of $(1-\frac{\alpha}{d+1})$. We adjust the borders of A_1 and A_2 so that A_1 again covers α of A. We now need to run SwapMisplaced twice, both to push down the density of the former A_1 array from $(1+\frac{1}{d})\gamma'$ to γ', and to push down the density of what used to be A_2 from β to γ'. We can then run LooseCompact again on the new A_1. This is the loop described in Lines 11-21.

After line 21, we must deal with two additional arrays, B, the union of RunOff(A_1) and R the union of Discard(A_1).

The array R is a mix of unmarked and dummy elements. We show in § 3.3 that the density of unmarked elements is exactly $\frac{1}{d+1}$. We no longer can exchange with the mirror $\tilde{R}$, since $\tilde{R}$ also has mostly dummy elements. However, using a slightly modified version of our loop in lines 23-26, we can compact the non-dummy elements of R into an array of size $\frac{1}{\log n(d+1)}|R|$, and then tightly compact the result.

The array B is sparse, with density bounded above by n^{c-1}. We run SparseLooseCompact repeatedly on B until we have compressed it by a factor of $8\log^3 n$. Note that after compressing it by this factor, B remains sparse.

Algorithm 2 Tight compaction by two

```
Require: Array $I$ of size $n$ with exactly half of $I$ marked, constants $\alpha$, $\beta$, $\gamma$,$\gamma'$, $W$, and $d$
Ensure: Obliviously shuffle $I$ so that all marked elements are at the front of the array
1: procedure TightCompactByTwo($A, \alpha, \beta, \gamma, W, d$)
2:     Divide $I$ into its left and right half, $I = A \cup \tilde{A}$
3:     LooseSwap($A, \tilde{A}, \frac{1}{2}, \frac{1}{2}, \beta$)
4:     For every statement in lines 5-23 other than line 12, execute the statement again on $\tilde{A}$, reversing the role of marked and unmarked elements
5:     Divide $A = A_1 \cup A_2$ with $|A_1| = \alpha n/2$ and $|A_2| = (1-\alpha)n/2$
6:     LooseSwap($A_1, A_2, \beta, \beta, \gamma'$)
7:     $B, R, S \leftarrow \{\}$
8:     LooseCompact($A_1, \gamma', \gamma, W, d$)
9:     Append RunOff($A_1$) to $B$
10:    Append Discard($A_1$) to $R$
11:    while $|A_1| + |A_2| > n/(6\log^3 n)$ do
12:        Divide $A_2 = A'_1 \cup A'_2$ so that $(1-\alpha)(|A_1| + |A_1|') = \alpha|A'_2|$
13:        LooseSwap($A_2, \tilde{A}_2, \frac{\beta}{1-\alpha}, 1 - \frac{\beta}{1-\alpha}, \beta$)
14:        LooseSwap($A_1, A'_2, \gamma'(1+\frac{1}{d}), \beta, \gamma'$)
15:        LooseSwap($A'_1, A'_2, \beta, \beta, \gamma'$)
16:        $A_1 \leftarrow A_1 \cup A'_1$
17:        $A_2 \leftarrow A'_2$
18:        LooseCompact($A_1, \gamma', \gamma, W, d$).
19:        Append RunOff($A_1$) to $B$
20:        Append Discard($A_1$) to $R$
21:    end while
22:    Divide $R = R_1 \cup R_2$ with $|R_1| = \frac{d-1}{d+1}|R|$ and $|R_2| = \frac{2}{d+1}|R|$
23:    LooseSwap($R_1, R_2, \frac{1}{d}, \frac{1}{d}, \gamma'$)
24:    while $|R_1| > \frac{1}{\log n(d+1)}|R|$ do
25:        LooseSwap($R_1, R_2, \gamma'(1+\frac{1}{d}), \frac{1}{2}, \gamma'$)
26:        LooseCompact($R_1, \gamma', \gamma, W, d$)
27:    end while
28:    $R \leftarrow$ TightCompact($R_1 \cup R_2, \alpha, \beta, \gamma, W, d$)
29:    $R \leftarrow$ leading $\frac{1}{d+1}|R|$ elements of $R$
30:    while $|B| > n/(8\log^3 n)$ do
31:        SparseLooseCompact on $B$
32:        Append Discard($B$) to $S$
33:    end while
34:    while $|S| > n/(8\log^3 n)$ do
35:        SparseLooseCompact on $S$
36:    end while
37:    $B \leftarrow B \cup \tilde{S}$
38:    $I \leftarrow$ Batcher sort of $(A \cup B) \cup (\tilde{A} \cup \tilde{B})$, sorted with marked elements first, then unmarked elements, then dummy elements
39:    $I \leftarrow$ leading $|A| + |\tilde{A}|$ elements of $I$
40:    $I \leftarrow \tilde{R} \cup I \cup R$
41: end procedure
```

This step also produces a DISCARD array S that is a mix of unmarked and dummy elements. We compress this array in the same way. Then $B \cup \tilde{S}$ is a sparse array of unmarked elements among dummy elements, and $\tilde{B} \cup S$ is a sparse array of marked elements among dummy elements. The number of non-dummy elements in these two arrays matches exactly the number of dummy elements in $A \cup \tilde{A}$. The Batcher sort in line 37 therefore moves all non-dummy elements to the leading $|A| + |\tilde{A}|$ positions of I. Appending $\tilde{R}$ to the front of the result and R to the end completes the compaction.

We show in § E.3, § 3.3 and § 3.4 that the probability of failure for LOOSESWAP, LOOSECOMPACT and SPARSELOOSECOMPACT, respectively, is negligible in n for $|A| = \Omega(n/\log^3 n)$. The loops in this subsection will run at most $O(\log\log n)$ times, so by a union bound the probability of failure of this algorithm is also negligible. In § E.2, we show the runtime of this algorithm is $2306.8n$.

Algorithm 3 Loose swap misplaced

```
Require: Arrays $A_1$ and $A_2$ with densities of marked elements bounded above by $\rho_1$ and $\rho_2$
    respectively and $|A_i| = \Omega(n/\log^3 n)$.
Ensure: Density of $A_1$ bounded above by $\rho_1' < \rho_1$ with all but negligible probability
 1: $t \leftarrow 0$
 2: $T \leftarrow \epsilon n + |A_1| \int_{\rho_1'}^{\rho_1} \left(1 - \frac{|A_1|(\rho_1 - \rho) + |A_2|\rho_2}{|A_2|}\right)^{-1} \frac{d\rho}{\rho}$
 3: for $t < T$ do
 4:     $t \leftarrow t + 1$
 5:     Choose random indices $i_1$ and $i_2$ from $A_1$ and $A_2$
 6:     if $A_1[i_1]$ is marked and $A_2[i_2]$ is unmarked then
 7:         Obliviously swap the contents of $i_1$ and $i_2$
 8:     else
 9:         Perform fake accesses
10:     end if
11: end for
```

3.3 Analysis of Algorithm 4

This algorithm shuffles elements into bins where we can then apply BINCOMPACT. By Lemma C.6, the probability that a bin D_i has more than $\gamma|D_i|$ marked balls is n^{-c}, for some constant $c < 1$.

The proof of correctness is an immediate consequence of the proof of correctness of BINCOMPACT given in § 4.1.

3.4 Analysis of Algorithm 5

This is the sparse analogue of LOOSECOMPACT given in the previous section.

By Lemma C.7, the probability that a fixed bin has more than $\log\log n$ elements is negligible in n. By taking a union bound over the bins, the probability that at least one bin has more than $\log\log n$ marked elements is also negligible in n.

The proof of correctness relies on the proof of correctness of BINCOMPACTMARGULIS given in § B.1.

Algorithm 4 Loose compaction

Require: Array A with $|A| = \Omega(n/\log^3 n)$ and at most $\gamma'|A|$ marked elements, for some constant $\gamma' < \gamma$.

Ensure: Compact A by a factor of $(1 - \frac{1}{d+1})$. The algorithm also returns arrays RunOff(A) and Discard(A). The array RunOff(A) satisfies $|A| = |\text{RunOff}(A)|$ and has $O(n^{-c})$ marked elements. The array Discard(A) satisfies $|\text{Discard}(A)| = |A|$. It is made up of a mix of dummy and unmarked elements.

```
1: procedure LooseCompact($A, \gamma, W, d$)
2:     Shuffle $A$ in the clear with Fischer-Yates.
3:     Divide $A$ into bins $\{D_i\}$ of size $(d+1)W/(1+\lg(d)+\frac{d}{2})$.
4:     $A \leftarrow \{\}$
5:     RunOff($A$) $\leftarrow \{\}$
6:     Discard($A$) $\leftarrow \{\}$
7:     for bin $D_i$ do
8:         BinCompact($D_i, W, d$)
9:         Append RunOff($D_i$) to RunOff($A$)
10:        Append Discard($D_i$) to Discard($A$)
11:        Append $D_i$ to $A$
12:    end for
13: end procedure
```

Algorithm 5 Sparse loose compaction

Require: Array A with $|A| = \Omega(n/\log^3 n)$ and at most $|A|n^{c-1}$ marked elements, for $c < 1$

Ensure: Compact A by a factor of $\frac{2}{3}$. The algorithm also returns an array Discard(A) of size $\frac{2}{3}|A|$. It is made up of a mix of unmarked and dummy elements.

```
1: procedure SparseLooseCompact($A, c, W$)
2:     Shuffle $A$ in the clear with Fischer-Yates
3:     Divide $A$ into bins $\{D_i\}$ of size $3W/2$.
4:     $A \leftarrow \{\}$
5:     Discard($A$) $\leftarrow \{\}$
6:     for bin $D_i$ do
7:         BinCompactMargulis($D_i, W$)
8:         Append $D_i$ to $A$
9:         Append Discard($D_i$) to Discard($A$)
10:    end for
11: end procedure
```

4 Bin compaction with bootstrap percolation

In this section, we give the algorithm BinCompact, and related algorithms needed to implement it. The algorithm BinCompactMargulis used for SparseCompact is similar, and we describe it in B. In both algorithms, the idea is the same: we assign elements to buckets of constant size, place each bucket at the vertex of an expander graph, and use some graph algorithm to assign directions to the edges of the graph.

4.1 Analysis of Algorithm 6

The algorithm BinCompact(A, W, d) replaces A with a new array that is $(1 - \frac{1}{d+1})$ times smaller, with all marked elements from the old A moved into the new A. In this process, some unmarked elements will be removed from A. They are placed into a new array Discard(A), which also contains a number of dummy elements.

The first time we call BinCompact, we generate a d-regular expander graph G of size $|G| = W/(\log W + 1 + \lg d + \frac{d}{2})$. This takes polylogarithmic time, as shown in Lemmas C.2, C.3 and C.4.

Each time we call BinCompact, we also assign to G a set of private data so that the edge set of G and its private data fit into a single memory word. The edge set and private data of G can therefore be accessed and altered without revealing the access pattern.

The array A is required to have size $|A| = (d + 1)|G|$ so that we can assign $d + 1$ elements of A to each vertex of G. We consider a vertex to be Empty if it contains no marked elements, and Nonempty otherwise. As part of the private data of G, for each vertex we store a single bit representing Empty or Nonempty, and another $\lg(d)$ bits to store a variable Count that counts the number of Empty neighbors. Count is used only in the Explore subroutine.

The rest of the private data of G is made up of edge weights. We use $kd/2$ bits to assign a direction to each edge of the graph. Formally, we replace each edge e between vertices v_i and v_j with a pair of directed edges $e(v_i, v_j)$ and $e(v_j, v_i)$. To indicate an edge pointing from v_i to v_j, we assign Outgoing to $wt(e(v_i, v_j))$ and Incoming to $wt(e(v_j, v_i))$. To indicate an edge pointing in the opposite direction, we reverse the labels. Of course, since the weights of $e(v_i, v_j)$ and $e(v_j, v_i)$ are always opposite, it only requires one bit to store both weights. The total storage cost for the private data of G is thus

$$|G|(1 + \lg d + \tfrac{d}{2})$$

bits, which explains our choice of the size of $|G|$.

We can now describe the algorithm BinCompact in terms of the private data of G. In lines 2-25, we mark vertices as Empty or Nonempty and check against certain failure conditions. In lines 26-28, we mark every edge as Incoming or Outgoing. Then in lines 29-47 we swap elements between vertices in the directions indicated by the edge weights, and compact each vertex's elements.

The edge weights are assigned by the subroutine Explore, which guarantees that every vertex with marked elements has more Outgoing edges than incoming edges. We show this in § 4.2. The subroutine ReduceEdges given as Algorithm 8 then ensures every such vertex has exactly one more Outgoing edge.

The loop in lines 32-41 ensures that every bucket D_i holds at most d marked elements, and at most $d + 1$ dummy elements. Indeed, vertices marked initially as Empty will receive at most d marked elements from their d neighbors. Vertices marked initially as Nonempty will have more edges marked Outgoing than Incoming. Note that the swap in line 37 does not necessarily transfer a marked element, but it does guarantee that position j in D_i holds an unmarked element. Let the number of Outgoing edges be o and the number of Incoming edges be i. After lines

Algorithm 6 Loose bin compaction with degree d Rucinski-Wormald graphs

```
Require: Array $A$ with $|A| = (d+1)W/(d \log W + 1 + \lg d + \frac{d}{2})$, and some constant $\gamma$
Ensure: Compact $A$ by a factor of $(1 - \frac{1}{d+1})$. Also returns arrays $\textsc{RunOff}(A)$, which has only
    dummy elements with probability $1 - n^{c-1}$, and $\textsc{Discard}(A)$, which has no marked elements
    and dummy elements distributed independently of $A$.
 1: procedure BinCompact($A, W, d$)
 2:     $B \leftarrow |A|$ dummy elements
 3:     if more than $\gamma|A|$ marked elements then
 4:         $\textsc{RunOff}(A) \leftarrow A$
 5:         Return $B$
 6:     end if
 7:     $k \leftarrow |A|/(d+1)$
 8:     $G \leftarrow$ a $d$-regular expander graph with $k$ vertices $\{v_1, \ldots, v_k\}$
 9:     Divide $A$ into $k$ bins $D_1, \ldots, D_k$ of size $(d+1)$
10:     Let $w$ be a single memory word of $W$ bits
11:     $w \leftarrow wt(v_i) \in \{\textsc{Empty}, \textsc{Nonempty}\} \cup wt(e(v_i, v_j)) \in \{\textsc{Incoming}, \textsc{Outgoing}\} \cup$
    $\textsc{Count}(v_i) \in \{0, \ldots, d\}$
12:     for $i \in \{1, \ldots, k\}$ do
13:         $\textsc{Count}(v_i) = 0$
14:         if $D_i$ contains no marked elements then
15:             $wt(v_i) \leftarrow \textsc{Empty}$
16:         else
17:             $wt(v_i) \leftarrow \textsc{Nonempty}$
18:         end if
19:     end for
20:     if proportion of vertices marked Empty is less than $(1-\gamma)^{d+1} - \varepsilon$ then
21:         $\textsc{RunOff}(A) \leftarrow A$
22:         Return $B$
23:     end if
24:     for $v \in V$ do
25:         Explore($v$)
26:     end for
27:     for $v \in V$ do
28:         ReduceEdges($v$)
29:     end for
30:     for $i \in \{1, \ldots, k\}$ do
31:         Add $d$ dummy elements to $D_i$
32:     end for
33:     for $v_i \in V$ do
34:         for neighbors $\{v_{i_1}, \ldots, v_{i_j}, \ldots, v_{i_d}\}$ of $v_i$ do
35:             if $wt(e(v_i, v_{i_j})) = \textsc{Outgoing}$ then
36:                 $j' \leftarrow$ index of $v_i$ in list of neighbors of $v_{i_j}$
37:                 Swap element $j$ in $D_i$ with element $d + 1 + j'$ in $D_{i_j}$
38:             else
39:                 Perform dummy memory accesses
40:             end if
41:         end for
42:     end for
43:     $A$, $\textsc{Discard}(A) \leftarrow \{\}$
44:     for $i \in \{1, \ldots, k\}$ do
45:         SlowCompact($D_i$)
46:         Append leading $d$ elements of $D_i$ to $A$
47:         Append the remainder of $D_i$ to $\textsc{Discard}(A)$
```

33-42 the number of marked elements will be at most $d+1-o+i=d$. Likewise the number of dummy elements for an initially NONEMPTY vertex will be exactly $d+1-o+i=d$.

Now D_i has $2d+1$ elements, with at most d elements marked, so we can run the algorithm SLOWCOMPACT(D_i) described in § 4.3. We obtain an output array A of size dk holding all the marked elements from the original A along with an array DISCARD(A) of size $(d+1)k$, as desired.

BINCOMPACT fails when A begins with more than $\gamma|A|$ elements marked, or when fewer than $(1-\gamma)^{d+1}-\varepsilon$ bins contain no marked elements. Since $|A| = \Omega(W/\log W) = \Omega(\log n)$, The first event occurs with probability n^{c-1}, by Lemma C.6. The second event occurs with probability $n^{-c'}$, by Lemma C.8. Thus the failure probability of BINCOMPACT is $n^{-c''}$, for some constant $c'' > 0$. In case of failure, the contents of A are emptied into RUNOFF(A), and BINCOMPACT(A) returns an array of dummy elements.

Algorithm 7 Explore subroutine for Rucinski-Wormald graphs

```
1:  procedure EXPLORE(vertex $v$)
2:      if $wt(v)$ = NONEMPTY then
3:          if COUNT = 0 then
4:              for neighbors $w$ of $v$ do
5:                  if $wt(w)$ = EMPTY then
6:                      COUNT ← COUNT + 1
7:                  end if
8:              end for
9:          else
10:             COUNT = COUNT + 1
11:         end if
12:         if COUNT > $d/2$ then
13:             $wt(v)$ ← EMPTY
14:             for neighbors $w$ of $v$ do
15:                 if $wt(w)$ = EMPTY then
16:                     $wt(e(v,w))$ ← OUTGOING
17:                 else
18:                     $wt(e(v,w))$ ← INCOMING
19:                     EXPLORE(w)
20:                 end if
21:             end for
22:         end if
23:     end if
24: end procedure
```

4.2 Analysis of Explore

We desire to implement majority bootstrap percolation, where EMPTY vertices are treated as active, and NONEMPTY vertices are inactive. Then vertices that are initially NONEMPTY will be activated when a majority of their neighbors are active, so that a majority of its edges will be marked as OUTGOING.

Let S be the collection of vertices marked EMPTY. We show in Corollary D.1 that, for $\frac{|S|}{|G|} \geq$

Algorithm 8 ReduceEdges subroutine for Rucinski-Wormald graphs

```
procedure ReduceEdges(vertex $v$)
    $i \leftarrow$ number of Incoming edges
    $o \leftarrow$ number of Outgoing edges
    while $o - i > 1$ do
        $w \leftarrow$ some neighbor of $v$ with $wt(e(v, w)) =$ Outgoing
        $wt(e(v, w)) \leftarrow$ Off
        $o \leftarrow o - 1$
        ReduceEdges($w$)
    end while
end procedure
```

$(1-\gamma)^{d+1}$, $|S|$ is a percolating set, i.e. majority bootstrap percolation activates every vertex.

However, running majority bootstrap percolation would take time superlinear in $|A|$. The Explore subroutine is similar to majority bootstrap percolation, but activates neighbors of active vertices as they become available, rather than looping through the entire array at each time step.

Claim 4.1. *The set of vertices activated after the* Explore *subroutine has been run on every vertex is the same as when we run majority bootstrap percolation on G.*

Proof. A vertex v activated after t calls to Explore will certainly have been activated after t time steps of majority bootstrap percolation. Conversely, suppose that t_0 is the first t for which some vertex v activated by majority bootstrap percolation is never activated by Explore. Then after time step $t_0 - 1$, a majority of v's neighbors are activated, so a majority of v's neighbors are eventually activated by Explore. But this is a contradiction, since each time we activate a vertex with Explore, we check each of its neighbors to see if they can be activated. This completes the proof. $\square$

4.3 Analysis of Algorithm 9

The loop in lines 3-15 ensures that both the left and right half of A have at most half of their elements marked. The correctness of this algorithm follows by induction. The runtime analysis showing $O(n \log n)$ cost is given in § E.8.

Funding acknowledgements

The first author is supported in part by DARPA/SPAWAR N66001-15-C-4065 and ODNI/IARPA 2019-1902070008.

The second author is supported in part by NSF-BSFGrant 1619348, DARPA/SPAWAR N66001-15-C-4065, ODNI/IARPA 2019-1902070008 US-Israel BSF grant 2012366, JP Morgan Faculty Research Award, Google Faculty Research Award, OKAWA Foundation Research Award, IBM Faculty Research Award, Xerox Faculty Research Award, B. John Garrick Foundation Award, Teradata Research Award, and Lockheed-Martin Corporation Research Award.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official views or policies, either expressed or implied, of the Department of Defense, DARPA, ODNI, IARPA, or the U.S. Government. The U.S. Government

Algorithm 9 Slow loose compaction

```
Require: Array $A$ of size $t$ with at most $t/2$ marked elements
Ensure: compact $A$ so that all marked elements are in the first $\lfloor t/2 \rfloor$ positions
 1: procedure SlowCompact($A$)
 2:     $u \leftarrow \lfloor t/2 \rfloor$, LCount $\leftarrow 0$ and RCount $\leftarrow t - 2u$
 3:     for $i \in \{1, \ldots, u\}$ do
 4:         if $A[i]$ is marked then
 5:             LCount $\leftarrow$ LCount $+ 1$
 6:         end if
 7:         if $A[u+i]$ is marked then
 8:             RCount $\leftarrow$ RCount $+ 1$
 9:         end if
10:         if $|$LCount $-$ RCount$| = 2$ then
11:             Swap $A[i]$ and $A[u+i]$
12:             LCount $\leftarrow$ LCount $\pm 1$
13:             RCount $\leftarrow$ RCount $\mp 1$
14:         end if
15:     end for
16:     SlowCompact($A[1 \ldots u]$) and SlowCompact($A[u+1 \ldots t]$)
17:     for $i \in 1, \ldots, \lfloor u/2 \rfloor$ do
18:         Swap $A[u+i]$ and $A[u+1-i]$
19:     end for
20: end procedure
```

is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

References

[1] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. Cryptology ePrint Archive, Report 2018/892, 2018. `https://eprint.iacr.org/2018/892`.

[2] József Balogh, Béla Bollobás, and Robert Morris. Graph bootstrap percolation. *Random Structures & Algorithms*, 41(4):413–440, 2012.

[3] József Balogh, Béla Bollobás, Robert Morris, and Oliver Riordan. Linear algebra and bootstrap percolation. *Journal of Combinatorial Theory, Series A*, 119(6):1328–1335, 2012.

[4] József Balogh and Boris G Pittel. Bootstrap percolation on the random regular graph. *Random Structures & Algorithms*, 30(1-2):257–286, 2007.

[5] Milan Bradonjić and Iraj Saniee. Bootstrap percolation on random geometric graphs. *Probability in the Engineering and Informational Sciences*, 28(2):169–181, 2014.

[6] Paul Bunn, Jonathan Katz, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient 3-party distributed oram. *IACR Cryptology ePrint Archive*, 2018:706, 2018.

[7] John Chalupa, Paul L Leath, and Gary R Reich. Bootstrap percolation on a bethe lattice. *Journal of Physics C: Solid State Physics*, 12(1):L31, 1979.

[8] Amin Coja-Oghlan, Uriel Feige, Michael Krivelevich, and Daniel Reichman. Contagious sets in expanders. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 1953–1987. SIAM, 2014.

[9] Shujie Cui, Sana Belguith, Ming Zhang, Muhammad Rizwan Asghar, and Giovanni Russello. Preserving access pattern privacy in sgx-assisted encrypted search. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2018.

[10] Matıas A Di Muro, Sergey V Buldyrev, and Lidia A Braunstein. Reversible bootstrap percolation: fake news and fact-checking. *arXiv preprint arXiv:1910.09516*, 2019.

[11] Joel Friedman et al. Relative expanders or weakly relatively ramanujan graphs. *Duke Mathematical Journal*, 118(1):19–35, 2003.

[12] Bernd Gärtner and Ahad N Zehmakan. Majority model on random regular graphs. In *Latin American Symposium on Theoretical Informatics*, pages 572–583. Springer, 2018.

[13] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[14] Alberto Guggiola and Guilhem Semerjian. Minimal contagious sets in random regular graphs. *Journal of Statistical Physics*, 158(2):300–358, 2015.

[15] Willem H Haemers. Interlacing eigenvalues and graphs. *Linear Algebra and its applications*, 226(228):593–616, 1995.

[16] Marc Lelarge. Diffusion and cascading behavior in random networks. *ACM SIGMETRICS Performance Evaluation Review*, 39(3):34–36, 2011.

[17] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *Theory of Cryptography Conference*, pages 377–396. Springer, 2013.

[18] Steve Lu and Rafail Ostrovsky. How to garble ram programs? In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 719–734. Springer, 2013.

[19] Grigorii A Margulis. Explicit constructions of graphs without short cycles and low density codes. *Combinatorica*, 2(1):71–78, 1982.

[20] Robert Morris. Minimal percolating sets in bootstrap percolation. *the electronic journal of combinatorics*, 16(1):R2, 2009.

[21] Natasha Morrison and Jonathan A Noel. Extremal bounds for bootstrap percolation in the hypercube. *Journal of Combinatorial Theory, Series A*, 156:61–84, 2018.

[22] Kartik Nayak and Jonathan Katz. An oblivious parallel ram with o (log2 n) parallel runtime blowup. *IACR Cryptology ePrint Archive*, 2016:1141, 2016.

[23] Nicholas Pippenger. Self-routing superconcentrators. *Journal of Computer and System Sciences*, 52(1):53–60, 1996.

[24] Eric Riedl. Largest minimal percolating sets in hypercubes under 2-bootstrap percolation. *the electronic journal of combinatorics*, 17(R80):1, 2010.

[25] Eric Riedl. Largest and smallest minimal percolating sets in trees. *the electronic journal of combinatorics*, pages P64–P64, 2012.

[26] Andrzej Ruciński and Nicholas C Wormald. Random graph processes with degree restrictions. *Combinatorics, Probability and Computing*, 1(2):169–180, 1992.

[27] Sanjib Sabhapandit, Deepak Dhar, and Prabodh Shukla. Hysteresis in the random-field ising model and bootstrap percolation. *Physical Review Letters*, 88(19):197202, 2002.

[28] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310, 2013.

[29] Angelika Steger and Nicholas C Wormald. Generating random regular graphs quickly. *Combinatorics, Probability and Computing*, 8(4):377–396, 1999.

[30] Tatyana S Turova and Thomas Vallier. Bootstrap percolation on a graph with random and local connections. *Journal of Statistical Physics*, 160(5):1249–1276, 2015.

[31] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 293–304, 2012.

[32] Ahad N Zehmakan. Opinion forming in erdős–rényi random graph and expanders. *Discrete Applied Mathematics*, 2019.

A From $O(\log\log n)$ blocks of client memory to $O(1)$ blocks

A.1 Description of algorithm

To get rid of the $\log\log n$ factor in client memory, we begin with an additional step of compaction that reduces density by a factor of $\log n$ in linear time.

This compaction follows the same model as before, by calling first TightCompact, then TightCompactByT to produce arrays with density less than

$$0.15 = \eta' < \eta = 1 - \sqrt[4]{0.51} \approx 0.155$$

Then we shuffle into bins of size $O(\log n / \log\log n)$.

Adapting the argument in Lemma C.6, we find that the probability of density greater than η is

$$e^{-O(\log n/\log\log n)} \ll e^{-\log\log n} = \frac{1}{\log n}.$$

We therefore can run BinCompact for all but $1/\log n$ bins. We transfer these bins, obliviously, to a runoff array.

Then, for each bin, we run a modified version of BinCompact, where instead of using a random expander graph of degree $d = 13$, we use a random expander graph of degree $d = 3$. The probability a bucket of 4 elements is Empty is at least

$$1 - \eta^4 \geq 0.51.$$

By [4, Corollary 11], for any $p > \frac{1}{2}$, all but $O(1)$ elements will be reached by majority bootstrap percolation, with probability $1 - \log^{-a} n$. In the $\log^{-a} n$ probability that it fails, we transfer the entire bin to the runoff array. Otherwise, we transfer the $O(1)$ buckets that are not reached by percolation to the runoff array.

When we complete this process, we have a sorted array with $n \log^{-a} n$ dummy elements, and a runoff array of size $4n$ and density $1/\log^a n$.

We now compact out the dummy elements from the sorted array and the non-dummy elements from the runoff array to give the final tightly compacted array.

Again, we sort into bins and then compact each bin using BinCompact. Now, however, we do not need to use LooseSwap or splitting the array to control density. We show that requisite bounds hold for bin and bucket sorting regardless.

Modifying the argument in Lemma C.6 once more, we find that for choosing $k \log n/\log\log n$ elements from an array with density $1/\log^a n$, and for $b < 1$ such that $a + b > 1$, the logarithm of the probability that $X = \ell > \log^b n$ is bounded above by

$$\begin{aligned}&\ell(\log\log n)(1-a-b) - \log\log\log n \log \ell\\ <& -b \log n \log\log\log n.\end{aligned}$$

Thus the probability that there are more than $\log^b n$ marked elements chosen is negligible in n. This means the density of each bin, and so also the fraction of non-empty buckets, can be made arbitrarily small by choosing n large enough. We can thus apply BinCompact.

We compress the marked elements into a subarray of size $n/\log^{b'} n$, for some $b' < b$, and then recursively call this algorithm. The cost of this recursion is $(1 + \frac{1}{\log^{b'} n}) = 1 + o(1)$.

After running BinCompact, all but $\log^{b-1} n$ fraction of the leading elements of the runoff buckets D_i will be unmarked, while all but $\log^{b-1} n$ fraction of the remaining d elements of each bucket will be dummy elements. Thus we can again use this compaction algorithm to separate dummy and unmarked elements.

A.2 Runtime of main algorithm

Here we give the cost of our main algorithm by analyzing the previous section and modifying the arugment in Appendix E. The first half of the algorithm, achieving $O(1/\log^a n)$ compaction, requires an analysis similar to the analysis given in § E.1 and § E.2. We find

$$\begin{aligned}&\left[\left(5 + \lg(d) + \frac{(1-(1-\gamma)^{d+1}+\varepsilon)(5d+1) - 2\lg(d)}{2d+2}\right)\cdot\big((d+1)+(d^2-1)\big) + \frac{\alpha}{\beta}(d+1)\right] n\\ &+\left[\left(1 - \frac{\beta}{1-\alpha}\right)^{-1}(\log(1+1/d)d + (1-\alpha)\log(\beta/\gamma)) + 2(d^2-1)\log(1+1/d)\right] n\\ &+\left(\frac{1}{\beta} - 2\right) n + (d-1)(1-\frac{2}{d})\log(d/\gamma) n + \frac{2}{d+1} U(n).\end{aligned}$$

Taking $d = 3, \alpha = 0.01$, $\beta = 0.3395$, $\gamma = 0.15$, gives a run time of

$$200.24n.$$

We then have to run the remainder of our algorithm on the array of size $4n$ with density $\frac{1}{\log^a n}$. However, the guarantee of low density eliminates the costs from α, β, and sparse arrays. The cost here is

$$4\left[\log(1+1/d)+\left(5+\lg(d)+\frac{o(1)}{2d+2}\right)\cdot\big((d+1)+(d^2-1)\big)\right]n$$
$$+\frac{2}{d+1}U(n).$$

Which gives a cost of

$$7807.03n.$$

Adding back in the cost of the original algorithm gives a cost of

$$U(n) = 8007.27n,$$

for TightCompactByTwo and a cost of

$$T(n) = 16014.54n$$

for general tight compaction.

B Margulis graph compaction

B.1 Analysis of Algorithm 10

The argument here is similar to the argument in § 4.1. We explain the modifications from BinCompact we make due to the sparsity of the array and the use of a Margulis graph.

The private data of G here is made up of two choices for the weight of a vertex $wt(v)$, either Empty or Nonempty. We also have three choices for the edge weights: Incoming, Outgoing or Off. There are $2|G|$ edges, and each of the above variables require 2 bits, so we require $5|G|$ bits to store the private data of G. This gives $|G| = W/(4\log W + 5)$. We use a bucket size of 3 elements from A per vertex, giving $A = 3W/(4\log W + 5)$.

We show in § B.2 that the algorithm ExploreMargulis fails with negligible probability. The guarantee from ExploreMargulis is that every vertex initially marked Nonempty will have at least 3 edges marked Outgoing, and every vertex will have at most 1 edge marked Incoming. Therefore running lines 22-37 moves all marked elements out of $D_i[j]$, for all i and $j = 1, 2, 3$, and at most one element is moved into $D_i[4]$, for all i. Note that Outgoing edges from vertices initially marked as Empty will never carry marked elements; these edges are marked only to guarantee that no vertex marked Empty has two or more Incoming edges.

B.2 Analysis of Algorithm 11

Claim B.1. *Let H be the set of vertices visited when we call* SubExploreMargulis(v)*, for some vertex v. Then H is a tree with all but negligible probability.*

Proof. Suppose by way of contradiction that $C \subseteq H$ is a cycle. By Lemma C.1, the shortest cycle of G has length at least $0.756 \log\log n$. Every vertex in C initially marked as Empty is adjacent to a vertex initially marked Nonempty. Therefore at least 1/3 of the elements of C must be initially marked. But by Lemma C.7 the probability that there are more than $0.25 \log\log n$ marked elements in A is negligible in n. Since

$$0.25\log\log n < \frac{0.756\log\log n}{3}$$

this is impossible. □

Algorithm 10 Loose compaction for bins of Margulis graphs

```
Require: Array $A$ with $|A| \leq 3W/(4\log W + 5)$
Ensure: Compact $A$ so that none of the final $\frac{2|A|}{3}$ elements are marked.
procedure BinCompactMargulis($A, W, b = 3, \ell = 3, d = 4$)
    $p \leftarrow \min\{\text{prime } i \big| i^3 - i \geq |A|/3\}$
    $k \leftarrow p^3 - p$
    $G \leftarrow$ a 4-regular Margulis expander graph with $k$ vertices, i.e. a Cayley graph for $SL_2(\mathbb{F}_p)$
    Label the vertices $V$ of $G$ as $V = \{v_1, \ldots, v_k\}$
    Divide $A$ into $k$ bins $D_1, \ldots, D_k$ of size 3, adding dummy elements if necessary
    Let $w$ be a single memory word of $W$ bits
    $w \leftarrow wt(v_i) \in \{\text{Empty}, \text{Nonempty}, \text{Terminal}\} \cup wt(e(v_i, v_j)) \in \{\text{Incoming}, \text{Outgoing}, \text{Off}\}$
    for $i \in \{1, \ldots, k\}$ do
        if $D_i$ contains no marked elements then
            $wt(v_i) \leftarrow$ Empty
        else
            $wt(v_i) \leftarrow$ Nonempty
        end if
    end for
    for $v \in V$ do
        ExploreMargulis($v$)
    end for
    for $i \in \{1, \ldots, k\}$ do
        Add 1 blank space to $D_i$
    end for
    for $v_i \in V$ do
        for First three neighbors $\{v_{i_1}, v_{i_2}, v_{i_3}\}$ of $v_i$ do
            if $wt(e(v_i, v_{i_j})) =$ Outgoing then
                Swap element $j$ in $D_i$ with element 4 in $D_{i_j}$
            else
                Perform dummy memory accesses
            end if
        end for
        for $j \in \{1, 2, 3\}$ do
            if $D_i[j]$ is marked and $wt(e(v_i, v_{i_4})) =$ Outgoing then
                Swap element $j$ in $D_i$ with element 4 in $D_{i_4}$
            else
                Perform dummy memory accesses
            end if
        end for
    end for
    for $i \in \{1, \ldots, k\}$ do
        $A[i] \leftarrow D_i[4]$
    end for
end procedure
```

Algorithm 11 Explore subroutine for Margulis graphs

```
procedure ExploreMargulis(vertex $v$)
    if $wt(v) = $ Nonempty then
        SubExploreMargulis($v$)
    end if
end procedure
procedure SubExploreMargulis($v$)
    if $wt(v) = $ Nonempty then
        $wt(v) \leftarrow$ Empty
        for neighbors $w$ of $v$ do
            if $wt(e(v, w)) = $ Off then
                $wt(e(v, w)) \leftarrow$ Outgoing
                SubExploreMargulis($w$)
            end if
        end for
    else
        for neighbors $w$ of $v$ do
            if $wt(e(v, w)) = $ Off then
                if $wt(w) = $ Nonempty then
                    $wt(e(v, w)) \leftarrow$ Outgoing
                    SubExploreMargulis($w$)
                end if
            end if
        end for
    end if
end procedure
```

Since H is a tree, all edges are oriented away from the root v, and every vertex has at most one INCOMING edge. Additionally, all initially NONEMPTY vertices have all four neighbors in H, and so three OUTGOING edges. Note that all vertices in H with fewer than four edges in H were initially marked EMPTY and all neighbors of such vertices in GnH were also initially EMPTY. Since SUBEXPLOREMARGULIS never assigns a weight to an edge between two EMPTY vertices, no swaps will occur between H and GnH.

C Combinatorial Lemmas

We state here a number of combinatorial lemmas we need throughout the paper, and give a citation if the result is well-known, or a proof otherwise.

Lemma C.1. *[Margulis [19]] There is a family of 4-regular expander graphs $\{G_k\}_{k\geq 0}$ with girth satisfying $|G_k| \geq 0.756 \log G_k$, with $G_k = (p_k^3 - p_k)$, for p_k the kth prime.*

The following result is due to Friedman [11] proving a conjecture of Alon.

Lemma C.2 (Friedman)**.** *The probability that a random d-regular graph G on n vertices, with dn even and $n > d$, has second eigenvalue $\leq 2\sqrt{d-1} + \epsilon$, is $1 - O(n^{-c})$.*

The following is due to Steger and Wormald [29], though the algorithm was first studied by Rucinski and Wormald [26].

Lemma C.3 (Steger, Wormald)**.** *There is an algorithm for generating a random d-regular graph, uniformly, in time $O(nd^2)$.*

Their algorithm is not difficult to describe. They take dn points, and, while possible, choose two points, and, if suitable, draw an edge between the corresponding vertices of G. If successful, the algorithm terminates. If unsuccessful, the algorithm resets. The following result is an immediate consequence of the well-known polynomial time algorithms for computing the characteristic polynomial and approximating roots.

Lemma C.4. *It is possible to test whether a d-regular graph G on n vertices is Ramanujan in time $poly(n)$.*

Combining these three results gives us an algorithm for generating a d-regular expander graph on n vertices in time $poly(n)$. Since our expander graphs will have $O(\log n)$ vertices, and we only need to generate $O(\log n)$ such graphs, this is sufficient for our needs.

Lemma C.5. *Consider a random walk along the integers $\mathbb{Z}$ where the probability of moving from i to $i+1$ is $f(\frac{i}{n})$ for some increasing function f. Let X be a random variable denoting the time it takes to walk from ρn to $\rho' n$. Then*

$$\Pr\left(\left|X - n\int_{\rho}^{\rho'} \frac{dx}{f(x)}\right| > \epsilon n\right) = e^{-8(\rho'-\rho)\epsilon^2 n}.$$

Proof. Let X_i be the time it takes to walk from ρn to $\rho' n$. From the definition of f, we have

$$\mathbb{E}[X_i] = \frac{1}{f(i/n)},$$

so that

$$\mathbb{E}[X] = \sum_{i=\rho n}^{\rho' n} \mathbb{E}[X_i] = n \sum_{i=\rho n}^{\rho' n} \frac{1}{nf(i/n)}.$$

The right hand side is a Riemann sum for $1/f(i)$ from ρ to ρ' including both the left and right endpoints, which gives

$$\frac{1}{f(\rho')} + n \int_{\rho}^{\rho'} \frac{dx}{f(x)} \leq n \sum_{i=\rho n}^{\rho' n} \frac{1}{nf(i/n)} \leq \frac{1}{f(\rho)} + n \int_{\rho}^{\rho'} \frac{dx}{f(x)}.$$

For sufficiently large n, we therefore have

$$|\mathbb{E}[X] - n \int_{\rho}^{\rho'} \frac{dx}{f(x)}| \leq \frac{\epsilon n}{2}.$$

By the Chernoff-Hoeffding Theorem, we have

$$\Pr\left(|X - \mathbb{E}[X]| \geq \frac{\epsilon n}{2}\right) \leq e^{-8(\rho'-\rho)\epsilon^2 n},$$

and applying the triangle inequality completes the proof.

$\square$

Lemma C.6. *Fix constants $\gamma' < \gamma$, and let A be an array of length n with $\gamma' n$ marked balls. Then if $k \log n$ balls are chosen at random from A, the probability there are more than $\gamma k \log n$ marked balls chosen is $n^{-\frac{(\gamma-\gamma')^2 k}{\gamma+\gamma'}}$.*

Proof. Let X be the number of marked balls chosen. The probability of $X = \ell$ is bounded above by

$$\begin{aligned}
&\binom{k\log n}{\ell}\left(\gamma' + \frac{\ell}{n}\right)^{\ell}\left(1 - \gamma' + \frac{\ell}{n}\right)^{k\log n - \ell} \\
&\leq \binom{k\log n}{\ell}(\gamma')^{\ell}(1-\gamma')^{k\log n-\ell}\left(1 + \frac{k\log n}{n\gamma'}\right)^{k\log n}\left(1 + \frac{k\log n}{(1-\gamma')n}\right)^{k\log n} \\
&\leq \binom{k\log n}{\ell}(\gamma')^{\ell}(1-\gamma')^{k\log n-\ell}\left(1 + O\left(\frac{\log^2 n}{n}\right)\right).
\end{aligned}$$

The probability of $X = \ell$ is similarly bounded below by

$$\binom{k\log n}{\ell}(\gamma')^{\ell}(1-\gamma')^{k\log n-\ell}\left(1 - O\left(\frac{\log^2 n}{n}\right)\right).$$

Therefore, up to a factor of $\left(1 \pm O\left(\frac{\log^2 n}{n}\right)\right)$, we can approximate X by a binomial random variable with $k \log n$ trials and probability γ'. Let

$$\delta = \frac{\gamma}{\gamma'} - 1.$$

By a Chernoff bound, we have

$$\begin{aligned}
&\Pr(X \geq \gamma k \log n) \\
&= \Pr(X \geq (1+\delta)\mathbb{E}[X]) \\
&\leq e^{-\frac{\delta^2}{2+\delta}\mathbb{E}[X]} \\
&= n^{-\frac{\delta^2 k\gamma'}{2+\delta}} \\
&= n^{-\frac{(\gamma-\gamma')^2 k}{\gamma+\gamma'}},
\end{aligned}$$

as desired. □

Lemma C.7. *Fix a constant $c < 1$, and let A be an array of $\Omega(n/\log^3 n)$ balls with $\leq n^{c-1}|A|$ marked balls. Then if $k\log n$ balls are chosen at random from A, the probability there are more than $\ell \log\log n$ marked balls chosen is negligible in n, for every choice of $k, \ell > 0$.*

Proof. The probability a randomly selected ball is marked is n^{c-1}. Let X be the number of marked balls chosen out of $k\log n$. Then

$$\begin{aligned}
\Pr(X > \log\log n) &\leq 2^{k\log n}\left(\frac{n^c + \ell\log\log n}{n}\right)^{\ell\log\log n}\left(1 - \frac{n^c - \ell\log\log n}{n}\right)^{1-\ell\log\log n} \\
&= n^{k\log 2 + (c-1)\ell\log\log n + o(1)},
\end{aligned}$$

which is negligible in n, as desired. □

Lemma C.8. *Fix constants ε, γ and k. Let A be an array of kn balls, with γkn balls marked. Shuffle A and divide it into bins of size k. The probability that there are fewer than $\left((1-\gamma)^k - \varepsilon\right)n$ empty bins is e^{-cn}, for some constant c.*

Proof. We give two proofs.

In the first proof, we observe that the probability an individual bin is empty is

$$\approx (1-\gamma)^k.$$

If Y_i is the event that bin i is empty, then the events $\{Y_i\}$ are negatively associated, and so we can apply a Chernoff bound to give the desired result.

In the second proof, we let X_i be random variables representing the number of bins with i marked balls in that bin, for $0 \leq i \leq k$.

Suppose that $X_i = nx_i$. We then have the following two equations:

$$\sum x_i = 1 \tag{C.1}$$

$$\sum ix_i = k\gamma \tag{C.2}$$

The number of ways to arrange marked balls to give $X_i = nx_i$ is

$$\binom{n}{nx_0, nx_1, \cdots, nx_k} \cdot \prod_{i=0}^{k} (\binom{k}{i})^{nx_i}.$$

We maximize this expression using Lagrange multipliers. Taking the logarithm, dividing by n, and taking the gradient gives

$$\nabla \mathbf{x} = \left(-1 - \log x_i + \log \binom{k}{i}\right)_i .$$

The method of Lagrange multipliers tells us

$$-1 - \log x_i + \log \binom{k}{i} = \lambda_A + i\lambda_B.$$

Taking $i = 0$, then $i = 1$, then i arbitrary allows us to cancel λ_A and λ_B to give

$$x_i = x_0 \frac{\binom{k}{i} x_1^i}{x_0^i}.$$

Returning to our original equations, we have

$$1 = x_0 \sum_{i=0}^{k} \binom{k}{i} (\frac{x_1}{x_0})^i = x_0(1 + \frac{x_1}{x_0})^k$$

and

$$k\gamma = x_0 \sum_{i=0}^{k} i\binom{k}{i} (\frac{x_1}{x_0})^i = x_1 k(1 + \frac{x_1}{x_0})^{k-1}$$

. Solving gives

$$\frac{x_1}{x_0} = \frac{\gamma}{1-\gamma}$$

and

$$x_0 = (1-\gamma)^k.$$

In particular, this means for $x_0 < (1-\gamma)^k - \varepsilon$, the probability of this event occurring is exponentially smaller in n. □

D Proof of Theorem 2

Proof. Choose $c > \frac{1}{2}$ and suppose by way of contradiction there is some set U with $\frac{|U|}{|G|} > c$ that does not activate all of G under percolation. Let S be the set that is activated by U. Then we also have $\frac{|S|}{|G|} > c$, and all vertices in $T = G \backslash S$ have fewer than $\lceil \frac{d+1}{2} \rceil$ edges to S. By the expander mixing lemma (see e.g. [15]) we have

$$\begin{aligned} e(T,T) &\le \frac{d|T|^2}{|G|} + \lambda |T| \\ &\le \left(\frac{d|T|}{|G|} + 2\sqrt{d-1}\right)|T|, \end{aligned}$$

when G is an expander graph. Taking $T = G \backslash S$ gives

$$\begin{aligned} e(T,T) &\le \left(d - \frac{dS}{G} + 2\sqrt{d-1}\right)|T| \\ &\le \left(d(1-c) + 2\sqrt{d-1}\right)|T|. \end{aligned}$$

First consider the case when d is odd. We have

$$2\sqrt{d-1} < \frac{d+1}{2}$$

for $d \geq 13$, so that, for c sufficiently close to 1, we have

$$e(T,T) < \frac{d+1}{2}|T|$$

and

$$e(S,T) > \frac{d-1}{2}|T|.$$

Thus at least one vertex of T has at least $\frac{d+1}{2}$ edges to S, and can be activated, a contradiction.

The proof for d even is similar, and relies on the fact that $d = 16$ is the smallest value with

$$2\sqrt{d-1} < \frac{d}{2}.$$

□

Corollary D.1. *For $d = 13$, there is some constant γ such that every set S with $|S|/|G| \geq (1-\gamma)^{d+1}$ is a percolating set for majority bootstrap percolation.*

Proof. This is an immediate consequence of Theorem 2 but we derive γ explicitly. We desire

$$\left(d - d(1-\gamma)^{d+1} + 2\sqrt{d-1}\right) < \frac{d+1}{2},$$

and so choose $\gamma < 0.005307$. □

E Runtime analysis of warm-up algorithm

E.1 Runtime of Algorithm 1

To analyze the cost, suppose the cost of this algorithm is $T(n)$ and the cost of Algorithm 2 is $U(n)$. Then:

Lines 2-6 require n steps to scan and count markings, and n steps to scan again and switch markings.

Line 7 requires $U(n)$ steps.

Line 8 requires n steps.

Line 9 requires $T(n/2)$ steps.

Lines 10-14 require n steps.

We thus have

$$T(n) = 4n + U(n) + T(n/2).$$

Solving the recurrence gives

$$T(n) = 8n + \sum_{k\geq 0} U(n/2^k).$$

We show that $U(n)$ is linear in n in § E.2, so we have

$$T(n) = 8n + 2U(n).$$

By § E.2 this simplifies to

$$T(n) = 4621.6n$$

E.2 Runtime of Algorithm 2

The cost of Line 3 is

$$(\frac{1}{\beta} - 2)n,$$

and the cost of Line 6 is

$$\log(\beta/\gamma) \left(1 - \frac{\beta}{1-\alpha}\right)^{-1} \alpha n.$$

Both bounds come from the results in § E.3.

By the analysis in § E.4, the cost of Line 8 and Line 18 over the course of the loop is

$$\begin{aligned}&\left(5 + \lg(d) + \frac{(1-(1-\gamma)^{d+1} + \varepsilon)(5d+1) - 2\lg(d)}{2d+2}\right) \alpha n \sum_{k\geq 0} \left(1 - \frac{\alpha}{d+1}\right)^k \\ =&\left(5 + \lg(d) + \frac{(1-(1-\gamma)^{d+1} + \varepsilon)(5d+1) - 2\lg(d)}{2d+2}\right) \cdot (d+1)n\end{aligned}$$

Lines 13-15 require again the estimates from § E.3. The cost of line 13 is

$$\begin{aligned}&\sum_{k\geq 0} \left(\frac{1}{\beta} - \frac{1-\alpha}{\beta}\right) \alpha \left(1 - \frac{\alpha}{d+1}\right)^k n \\ =&\frac{\alpha}{\beta}(d+1)n.\end{aligned}$$

The cost of line 14 is

$$\begin{aligned}&\sum_{k\geq 0} \log(\gamma^{\frac{d+1}{d}}/\gamma)(1 - \frac{\beta}{1-\alpha})^{-1} \alpha \left(1 - \frac{1}{d+1}\right) \left(1 - \frac{\alpha}{d+1}\right)^k \\ =&\log(1 + 1/d)(1 - \frac{\beta}{1-\alpha})^{-1} dn\end{aligned}$$

And the cost of line 15 is

$$(1-\alpha)(1 - \frac{\beta}{1-\alpha})^{-1} \log(\beta'/\gamma)n.$$

In lines 22-28, the size of Discard(A_1) is equal to the size of A_1 before running LooseCompact, so that the total size of R is

$$\sum_{k\geq 0} \left(1 - \frac{\alpha}{d+1}\right)^k \alpha n.$$

Summing the geometric series gives $|R| = (d+1)n$. The array R is a mix of unmarked elements and dummy elements, and we desire to compact out the unmarked elements. We show in § 4.1 that the density of unmarked elements in R is exactly $\frac{1}{d+1}$.

The cost of line 23 is

$$(d-1)(1 - \frac{2}{d}) \log(d/\gamma')n,$$

and the cost of line 25 is

$$\begin{aligned}&\sum_{k\geq 0} 2\log(1 + 1/d)(1 - \frac{1}{d+1})^k (d-1)n \\ =&2(d^2 - 1)\log(1 + 1/d).\end{aligned}$$

The cost of line 26 is

$$\left(5+\lg(d)+\frac{(1-(1-\gamma)^{d+1}+\varepsilon)(5d+1)-2\lg(d)}{2d+2}\right)(d-1)n\sum_{k\geq 0}\left(1-\frac{1}{d+1}\right)^k$$
$$=\left(5+\lg(d)+\frac{(1-(1-\gamma)^{d+1}+\varepsilon)(5d+1)-2\lg(d)}{2d+2}\right)\cdot(d^2-1)n$$

In addition, the cost of line 28 is $\frac{2}{d+1}U(n)$.

Line 30-33 requires $(3/2)\cdot 14|B|/3$ steps, by § 3.4. The size of $|B|$ is the sum of $|A_1|$ over all sizes of $|A_1|$ that arose in the previous loop. This is the same computation we did for the size of R above, so we have $|B|=(d+1)n$, and the total cost is $7\cdot(d+1)n$.

The size of $\textsc{Discard}(B)$ is equal to the input size of B, so that the size of S after line 33 is

$$\sum_{k\geq 0}(\frac{1}{3})^k(d+1)n=\frac{3}{2}|B|.$$

Therefore the cost from lines 34-36 is 3/2 the cost of lines 30-33, i.e. $\frac{21}{2}(d+1)n$.

Summing all of these expressions gives, as the total cost:

$$\left[\left(5+\lg(d)+\frac{(1-(1-\gamma)^{d+1}+\varepsilon)(5d+1)-2\lg(d)}{2d+2}\right)\cdot\big((d+1)+(d^2-1)\big)+\frac{\alpha}{\beta}(d+1)\right]n$$
$$+\left[\left(1-\frac{\beta}{1-\alpha}\right)^{-1}(\log(1+1/d)d+(1-\alpha)\log(\beta/\gamma))+2(d^2-1)\log(1+1/d)+\frac{35}{2}\cdot(d+1)\right]n$$
$$+\left(\frac{1}{\beta}-2\right)n+(d-1)(1-\frac{2}{d})\log(d/\gamma)n+\frac{2}{d+1}U(n).$$

The effect of the final $\frac{2}{d+1}$ term is equivalent to multiplying the rest of the runtime by $\frac{d+1}{d-1}$.

Taking $d=13,\alpha=0.01$, $\beta=0.278928$, $\gamma=0.005307$, gives

$$U(n)=2306.8n.$$

E.3 Runtime of Algorithm 3

When the density of A_1 is ρ, the number of marked balls in A_2 is $|A_1|(\rho_1-\rho)+|A_2|(\rho_2)$. Thus the probability that a random ball from A_1 is marked is $1/\rho$ and the probability that a random ball from A_2 is unmarked is

$$\left(1-\frac{|A_1|(\rho_1-\rho)+|A_2|\rho_2}{|A_2|}\right)^{-1}.$$

Writing X for the number of random swaps required to force the density of A_1 to be less than ρ_1' and applying Lemma C.5 gives

$$\left|X-|A_1|\int_{\rho_1'}^{\rho_1}\left(1-\frac{|A_1|(\rho_1-\rho)+|A_2|\rho_2}{|A_2|}\right)^{-1}\frac{d\rho}{\rho}\right|<\epsilon n,$$

with all but negligible probability.

We bound the integral in two ways. When $\rho_2=1-\rho_1$ and $|A_1|=|A_2|$, the integral simplifies to:

$$|A_1|\int_{\rho_1'}^{\rho_1}\frac{d\rho}{\rho^2}=\frac{1}{\rho_1}-\frac{1}{\rho_1'}.$$

This is the bound we use to evaluate the cost of lines 3 and 13 in Algorithm 2.

We can also bound the term

$$\left(1 - \frac{|A_1|(\rho_1 - \rho) + |A_2|\rho_2}{|A_2|}\right)^{-1} \leq \left(1 - \frac{|A_1|\rho_1 + |A_2|\rho_2}{|A_2|}\right)^{-1}$$
$$= \frac{|A_2|}{|A_1|\rho_1 + |A_2|(1-\rho_2)}.$$

This gives

$$X < \epsilon n + \frac{|A_2|}{|A_1|\rho_1 + |A_2|(1-\rho_2)} \log\left(\frac{\rho_1}{\rho_1'}\right),$$

the estimate we use in our evaluation of lines 6, 14 and 15 of Algorithm 2.

E.4 Runtime of Algorithm 4

Line 2 requires $|A|$ steps.

Line 9 requires $\left(4 + \lg(d) + \frac{(1-(1-\gamma)^{d+1}+\varepsilon)(5d+1)-2\lg(d)}{2d+2}\right)|D_i|$ steps.

Thus the entire algorithm requires

$$\left(5 + \lg(d) + \frac{(1-(1-\gamma)^{d+1}+\varepsilon)(5d+1)-2\lg(d)}{2d+2}\right)|A|$$

steps.

E.5 Runtime of Algorithm 5

Line 2 requires $|A|$ steps.

Lines 6-10 require $11A/3$ steps, by § E.9.

This gives a total cost of $14A/3$ steps.

E.6 Runtime of Algorithm 6

The graph generation in line 8 only needs to occur once per call to LooseCompact, and by Lemmas C.2,C.3 and C.4 can be completed in polylogarithmic, i.e. $o(n)$ time.

Lines 3-23 can be accomplished with a single scan through A, followed by writing to $\textsc{Count}(v_i)$ and $wt(v_i)$, which requires $|A| + 2k$ steps. If either of the runoff conditions are triggered, we must spend another $|A|$ steps obliviously swapping the contents of A into $\textsc{RunOff}(A)$. This gives a total cost of $2|A| + 2k$ steps.

Lines 24-26 require $d + (1-(1-\gamma)^{d+1}+\varepsilon)k \cdot (5d+1)/2$ steps, by § E.7.

Lines 30-39 require dk swap steps.

Line 42 requires $d\lg(d)$ steps, by § E.8, so that lines 41-44 require $dk\lg(d)$ steps.

Replacing k everywhere with $|A|/(d+1)$ gives

$$\frac{(4d+8) + 2d + (1-(1-\gamma)^{d+1}+\varepsilon)(5d+1) + 2d + 2d\lg(d)}{2(d+1)}|A|$$
$$= \left(4 + \lg(d) + \frac{(1-(1-\gamma)^{d+1}+\varepsilon)(5d+1)-2\lg(d)}{2d+2}\right)|A|$$

total steps.

E.7 Runtime of Algorithm 7

The if statement on line 2 runs for every vertex, giving a cost of d steps. The rest of the algorithm only runs for the initially NONEMPTY vertices. Lines 4-8 and lines 13-21 will be run at most one time per vertex. Each loop requires one step, since we access and write over the single memory word containing the private data of G. Line 10 may be run up to $(d+1)/2$ times. This gives a total cost of $(5d+1)/2$ per NONEMPTY vertex. With all but negligible probability, there are less than $(1-(1-\gamma)^{d+1}+\varepsilon)$ NONEMPTY vertices, giving a total cost of

$$d + \left(1-(1-\gamma)^{d+1}+\varepsilon\right)\left(\frac{5d+1}{2}\right)$$

E.8 Runtime of Algorithm 9

The cost of lines 3-15 is $t/2$ swaps. Lines 17-19 are not oblivious swaps - i.e., they could be represented by re-indexing, and cost nothing. Thus if the cost of the algorithm is $L(t)$, we have

$$L(t) = t/2 + 2L(t/2)$$

so that

$$L(t) = t/2\lg(t).$$

E.9 Cost of Algorithm 10

The argument here is similar to the argument in § 4.1.

As in § E.6, we generate the required expander graph once, which contributes polylogarithmically to the total cost.

Lines 9-15 require $4k|D_i| = 4|A|/3$ steps, three steps to read and one step to write.

Lines 16-18 require $|G| = |A|/3$ steps.

Lines 22-37 require $6k|D_i| = 2|A|$ swap steps.

This gives a total of $11|A|/3$ steps.

E.10 Runtime of Algorithm 11

The cost of the subprocedure SUBEXPLOREMARGULIS is either 10 or 13 steps, depending on the branching statement in line 7. lines 3-16 by direct calculation, is 16 steps. Because SUBEXPLOREMARGULIS will be called at most $O(\log\log n)$ times, the total cost is $o(\log n)$. Therefore the total cost of EXPLOREMARGULIS is $(1+o(1))|G|$.