

Super-Scalable Algorithms

A Dissertation

Presented to
the faculty of the School of Engineering and Applied Science
University of Virginia

in partial fulfillment
of the requirements for the degree

Doctor of Philosophy

by

Nathan James Brunelle

December 2017

APPROVAL SHEET

This Dissertation
is submitted in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Author Signature: 

This Dissertation has been read and approved by the examining committee:

Advisor: Gabriel Robins

Committee Member: James Cohoon

Committee Member: Kevin Skadron

Committee Member: Mircea Stan

Committee Member: Ke Wang

Committee Member: _____

Accepted for the School of Engineering and Applied Science:



Craig H. Benson, School of Engineering and Applied Science

December 2017

Abstract

We propose two new highly-scalable approaches to effectively process massive data sets in the post- Moore’s Law era, namely (1) designing algorithms to operate directly on highly compressed data, and (2) leveraging massively parallel finite automata-based architectures for specific problem domains. The former method extends scalability by exploiting regularity in highly-compressible data, while also avoiding expensive decompression and re-compression. The latter hardware succinctly encapsulates complex behaviors via simulation of non-deterministic finite-state automata. We evaluate the efficiency, extensibility, and generality of these non-traditional approaches in big data environments. By presenting both promising experimental results and theoretical impossibility arguments, we provide more comprehensive frameworks for future research in these areas.

To all of my mentors, collaborators, and supporters:

*To **Gabriel Robins**, my adviser, for providing wisdom, guidance, and conversation. You inspire all of those around you to ascend to a higher level of rigorous and creative thought. I aspire to your nobility.*

*To **Kevin Skadron**. Your input into my research and career have been invaluable. Your guidance directs others to become formidable intellectuals, scientists, and educators. You deserve highest esteem for your service in nurturing high-quality academics.*

*To **Jack Wadden and Tommy Tracy**, my two most significant research collaborators. Your technical insights have inspired this research and broadened my understanding of computing immeasurably.*

*To the **UVA CAP research group**. You all have created a thriving research community. I am proud to have struggled among you in this endless pursuit lofty goals that we call research.*

*To **Tommy Tracy, Cameron Blandford, Alyson Irizarry, Jack Wadden, and Robbie Hott**. I am profoundly appreciative of your efforts in helping me to edit various papers I have written over the years. I especially thank you for your feedback on this dissertation.*

*To **James Cohoon, Gabriel Robins, Kevin Skadron, Mircea Stan, and Ke Wang**. I greatly appreciate the feedback you have given as members of my Ph.D. committee. Your suggestions, criticisms, and (especially) praises fell on welcoming ears.*

*To my father, **James Brunelle**. To my mother, **Janet Brunelle**. To my grandmother, **Elouise Schultz**. To my siblings **Justin Brunelle, Collin Brunelle, and June Brunelle**. To the rest of my extended family who I have no space to mention here. I could not begin to thank you for your influence on my life and education, as your influences are too grand for anyone to comprehend. I can only offer a vague “thank you”, as no utterance can satisfactorily encapsulate my appreciation.*

Contents

| | |
|--|------------|
| Contents | iii |
| List of Tables | vi |
| List of Figures | vii |
| 1 Super-Scalable Algorithms | 1 |
| 1.1 Compression-Aware Algorithms | 2 |
| 1.2 Automata Processing | 4 |
| 1.3 Lessons and Future Directions | 5 |
| 2 Compression-Aware Algorithms | 7 |
| 2.1 Introduction to Compression-Aware Computation | 7 |
| 2.2 Case study: Compressed Lists | 9 |
| 2.2.1 Arithmetic Sequences Compression | 11 |
| 2.2.2 Context Free Grammar Compression | 12 |
| 2.2.3 Lempel-Ziv '77 Compression | 14 |
| 2.2.4 Lempel-Ziv '78 Compression | 16 |
| 2.2.5 Lessons | 17 |
| 2.3 Problem Statement and Results | 18 |
| 2.4 Related Work | 19 |
| 2.5 Set-of-Lines Compression Scheme | 20 |
| 2.5.1 Compression Algorithm | 20 |
| 2.5.2 Nearest Neighbor Queries | 24 |
| 2.5.3 Range Queries | 27 |
| 2.5.4 Convex Hull | 31 |
| 2.5.5 Experimental Comparison | 31 |
| 2.6 Re-Pair for Graphs | 32 |
| 2.6.1 Topological Sort | 35 |
| 2.6.2 Bipartite Assignment | 36 |
| 2.7 Component-Based Compression | 36 |
| 2.7.1 Topological Sort | 36 |
| 2.7.2 Single Source Shortest Path | 37 |
| 2.7.3 Minimum Spanning Tree | 38 |
| 2.8 Boldi and Vigna: WebGraph Compression | 39 |
| 2.8.1 Bipartite Assignment | 40 |
| 2.9 Summary | 41 |
| 2.A Lemmas and Theorems | 42 |
| 2.B Algorithms Pseudocode | 50 |
| 3 Overview and Complexity-Theoretic Analysis of Automata Processing | 57 |
| 3.1 Finite State Automata | 58 |
| 3.2 Micron's Automata Processor | 59 |
| 3.2.1 Homogeneous Finite Automata | 59 |
| 3.2.2 Bit-parallel Algorithm | 62 |

| | | |
|----------|---|------------|
| 3.2.3 | Hardware Specifications | 62 |
| 3.3 | Characterizing the Computational Power of the AP | 64 |
| 3.3.1 | Alternating Finite Automata | 65 |
| 3.4 | Micron's AP Accepts the Regular Languages | 68 |
| 3.4.1 | Eliminating Counter Elements | 69 |
| 3.4.2 | Eliminating Boolean Gates | 70 |
| 3.5 | Comparison to Circuit Complexity | 71 |
| 3.5.1 | Circuit Complexity | 71 |
| 3.5.2 | Nick's Class | 72 |
| 3.5.3 | Circuit Complexity of the AP | 72 |
| 3.6 | Summary | 73 |
| 4 | Pseudorandom Number Generation using Parallel Automata | 74 |
| 4.1 | Motivation | 75 |
| 4.2 | Pseudorandom Number Generation | 76 |
| 4.2.1 | Previous Work on PRNGs | 77 |
| 4.2.2 | Markov Chains as Automata | 78 |
| 4.3 | AP-PRNG Algorithm | 78 |
| 4.4 | Hardness Assumption | 80 |
| 4.4.1 | Hardness Problem Statement | 82 |
| 4.4.2 | Prior Art in Automata Learning | 84 |
| 4.5 | Theoretical Performance analysis | 85 |
| 4.5.1 | Stretch | 86 |
| 4.5.2 | Complexity | 87 |
| 4.6 | AP-PRNG in Practice | 87 |
| 4.6.1 | Hardware Constraints | 88 |
| 4.6.2 | Sensitivity Analyses | 89 |
| 4.6.3 | AP-PRNG Performance Model | 93 |
| 4.7 | Sensitivity to Weakly Random Input | 96 |
| 4.7.1 | Entropy Extractors | 97 |
| 4.7.2 | Min-Entropy | 98 |
| 4.7.3 | Striding APPRNG | 98 |
| 4.7.4 | Experimental Results | 100 |
| 4.8 | Automata-based Bloom Filtering | 102 |
| 4.8.1 | Bloom Filters | 102 |
| 4.8.2 | Automata-based Bloom filters | 104 |
| 4.9 | Summary | 111 |
| 5 | Conclusions and Future Directions | 112 |
| 5.1 | Compression-Aware Algorithms | 113 |
| 5.1.1 | Contributions | 113 |
| 5.1.2 | Future Directions | 113 |
| 5.2 | Automata Computing | 116 |
| 5.2.1 | Contributions | 116 |
| 5.2.2 | Future Directions | 117 |
| 5.3 | Automata-based Compression-Aware Algorithms | 118 |
| | Bibliography | 121 |
| A | Compression-Aware Algorithms implementations | 127 |
| 1.1 | Lossless Set of Lines Python Code | 127 |
| 1.2 | Lossy Set of Lines Python Code | 132 |
| 1.3 | Set-of-Lines Nearest Neighbor and Range Searches | 143 |
| B | Poster DCC 2013 | 150 |

| | | |
|----------|--|------------|
| C | Poster DCC 2015 | 153 |
| D | APPRNG Patent Application | 156 |
| E | APPRNG Python Implementation | 172 |
| 5.1 | Moore Machine Simulator | 172 |
| 5.2 | APPRNG Creation | 173 |
| 5.3 | Sample Usage | 174 |
| F | AP Bloom Filter Python Implementation | 177 |
| 6.1 | Finite Automata Simulator | 177 |
| 6.2 | Bloom Filter Data Structure Implementation | 178 |
| 6.3 | Experiment Implementation | 180 |
| G | PhD Defense Presentation Video and Slides | 183 |

List of Tables

| | | |
|-----|---|-----|
| 1.1 | Overview of techniques for parallelism in computing according to Flynn’s Taxonomy | 4 |
| 3.1 | Constrained vs. Unconstrained resources available on the AP | 68 |
| 4.1 | It is statistically harder to identify correlation between chains with more states. . . | 90 |
| 4.2 | First Generation AP Architectural Parameters | 94 |
| 4.3 | AP PRNG Parameters | 94 |
| 4.4 | AP PRNG Performance Model | 94 |
| 4.5 | AP PRNG performance modeled on different memory technologies. AP PRNG throughput is limited by peak memory throughput for DDR3 and DDR4 technologies. | 95 |
| 4.6 | Striding APPRNG mitigating weakly random input for 571 parallel 8-state automata over 550,000 6-bit inputs. | 102 |
| 4.7 | Minimum Stride needed by Fixing Period in for 571 parallel 8-state automata over 550,000 6-bit inputs to pass all Small Crush tests. | 102 |
| 4.8 | Stochastic Transition Matrix of each Markov Chan used for Automata-Based Bloom Filtering | 106 |
| 5.1 | Letter frequency and corresponding Huffman code for the Huffman Tree shown in Figure 5.1. | 119 |

List of Figures

| | | |
|------|--|----|
| 1.1 | Actual buildings (left), and a corresponding highly compressible / regular CAD model (right). | 3 |
| 2.1 | Running an algorithm on raw data should always be faster than compressing then running a compression-aware algorithm. We claim performance benefits when data is <i>already</i> compressed. | 9 |
| 2.2 | An example of a set of arithmetic sequences representing a set of numbers. The circular points represent the arithmetic sequence starting at 0 and extending every 5 points to 50, $\{0, 5, 10, 15, 20, \dots\}$. The triangular points represent the arithmetic sequence starting at 0 and extending every 12 points to 48, $A_2 = \{0, 12, 24, 36, 48, \dots\}$. The combined sequence is the union of the two, $\{0, 5, 10, 12, 15, 20, 24, \dots\}$ | 11 |
| 2.3 | An example of a 5 th order statistic query on arithmetic sequences representation of a set of numbers. The circular points represent the arithmetic sequence starting at 0 and extending every 5 points to 50, $\{0, 5, 10, 15, 20, \dots\}$. The triangular points represent the arithmetic sequence starting at 0 and extending every 12 points to 48, $A_2 = \{0, 12, 24, 36, 48, \dots\}$. The combined sequence is the union of the two, $\{0, 5, 10, 12, 15, 20, 24, \dots\}$. The combined arrival rate of the two sequences is $\Lambda = \frac{1}{5} + \frac{1}{12} = \frac{17}{60}$, this means the guess point is $\frac{5}{\Lambda} = 5 \cdot \frac{60}{17} = \frac{300}{17} \approx 17.65$, we mark this with a square. | 13 |
| 2.4 | The dependency graph for a given context free grammar (this is the same grammar as is given in Section 2.2.2). | 13 |
| 2.5 | An example of a set of lines representing a pointset. The dashed bold line is defined by $y = -\frac{5}{12}x + 4.25$. Each point is ~ 6.34 units apart along this line, with the left-most point occurring at (1.2, 3.75). | 21 |
| 2.6 | An example of an ϵ -regular point set (solid dots) whose points are within ϵ of the corresponding points of a regular point set (hollow dots). This figure appears in [1]. | 24 |
| 2.7 | If all pre-compression points on one edge of compression-aware convex hull follow an arc extending no more than ϵ from that edge, the discrepancy in the number of points between the lossy-compressed convex hull result, and the non-compressed result could be arbitrarily large. | 31 |
| 2.8 | Runtime of compression-aware range queries do not depend on point count. Tests were performed on sets of 100 lines in 8 dimensions. Number of points per line was varied to vary total number of points. | 32 |
| 2.9 | Runtime of Compression-aware range queries do not depend on number of points returned. Tests performed on sets of 10,000,000 points in 3 Dimensions. We varied the size of the query range in order to change the number of points returned in each query. | 33 |
| 2.10 | The query time for compression-aware range queries is linear by number of lines. Tests performed on sets of 1,000,000 points in 8 Dimensions. | 33 |
| 2.11 | Set-up time for traditional range queries. The higher line represents total initialization overhead required (decompression time plus tree-building time). The lower line shows time required for building the KD-tree. | 34 |

| | | |
|------|--|----|
| 2.12 | A sample graph (a) and its representation as a Re-Pair compressed graph (b). The dot-circled vertices are dictionary keys and the destination their outgoing edges are the edges which they replace in the adjacency lists. The dotted edges have 0 weight. | 35 |
| 2.13 | Example graph (a) and its hierarchical compression (b). Capital-lettered vertices represent super nodes, lowercase-lettered vertices represent terminal nodes. The graph labeled <i>S</i> is the highest in the hierarchy. | 37 |
| 2.14 | This figure shows that to merge two minimum spanning trees, simply applying the cut property between the two subsets of vertices is not sufficient | 39 |
| 3.1 | A photograph of the Micron Automata Processor (AP) hardware, manufactured by the Micron Technology corporation in 2016, which can be plugged into commodity PCs. | 60 |
| 3.2 | An example of homogeneous vs. non-homogeneous finite automata. In the automaton on the left all incoming transitions of all states match on the same symbol set, making it homogeneous. In the automaton on the right there is an incoming transition for state “H” which matches on the symbol ‘a’, and another that matches on the symbol ‘b’, thus this automaton is not homogeneous. | 61 |
| 3.3 | Eliminating non-homogeneity in automata by splitting states. | 62 |
| 3.4 | A simplified model of an STE’s construction. The STE decodes each 8-bit symbol to perform a row lookup. If the lookup returns a 1, and the enable signal is active, then the STE propagates its enable signal to the routing matrix. | 63 |
| 3.5 | Examples of accepting DFA, NFA, and AFA paths. Each circle represents an active existential state, each square represents an active universal state. They are arranged left-to-right to demonstrate a sequence of 2 transitions. Note that the DFA is only in one state at a time, whereas NFAs and AFAs may be in multiple | 67 |
| 3.6 | Converting a counter with threshold 3 to states and AND gates. All states match on all inputs in this construction. | 69 |
| 4.1 | An example Markov Chain of a biased coin which lands heads with probability $\frac{1}{3}$ and tails with probability $\frac{2}{3}$ | 78 |
| 4.2 | An example construction of an automaton which, when given a sequence of random input characters, emulates the behavior of the Markov Chain shown in Figure 4.1. | 79 |
| 4.3 | Overview of the AP-PRNG algorithm for M total machines with N states each. Each machine simulates a Markov chain representing an N -sided fair die. The N -permutations are used to define the matching symbols on each of the outgoing transitions on each state and requires $N \log N!$ bits of randomness. The random input string requires $\log N$ random bits per symbol. The output will contain M pseudorandom bits per each input bit. | 81 |
| 4.4 | Example execution of an instance of APPRNG for 4 states, 2 machines, and 11 input symbols. | 82 |
| 4.5 | Each bar represents the average number of Crush failures over four trials for parallel 8-state Markov chains with a reconfiguration threshold of 200,000. The darker bars represent failure rates when interleaving output bits. Spikes in failure rates occur when the same Markov chains always contribute to the same bits in output integers. The lighter bars represent failure rates when successive output from a single Markov chain contributes to a single output integer. This eliminates the spike in failures, but reduced overall performance. | 91 |
| 4.6 | As the reconfiguration threshold increases, it becomes easier for statistical tests to identify non-random behavior. | 92 |
| 4.7 | Output quality of AP PRNG with output permutation greatly increases quality of random output. AP PRNG passes all tests in BigCrush with a reconfiguration threshold of at least 1,000,000, and at most 2,000,000 | 93 |

| | | |
|------|---|-----|
| 4.8 | Percentage of runtime spent reconfiguring vs. AP PRNG throughput with different reconfiguration thresholds. Performance increases dramatically if AP PRNG is able to reconfigure less frequently. | 94 |
| 4.9 | AP PRNG is up to $6.8\times$ more power efficient than the highest-throughput reported GPU PRNG depending on the deployment scenario. | 96 |
| 4.10 | An example of a 2-state automaton that is 2-strided. | 98 |
| 4.11 | An illustration of how automata-based bloom filtering decreases false-positive rates via increased numbers of automata. Each gray circle represents the set of elements accepted by an automaton, with all automata rejecting every element in the set S . A query on any string that falls outside of all the gray circles will result in the response $x \in S$. The more automata running against x , the smaller the probability that x falls in neither the circle for the set S nor any of the gray circles (which would be a false positive). | 105 |
| 5.1 | Huffman Tree for the letter frequencies shown in Table 5.1. This tree could also be viewed as a Moore machine, where once the computation reaches one of the leaves of the tree, it will output its label as the corresponding character. | 119 |

Chapter 1

Super-Scalable Algorithms

Big datasets are emerging in all sectors of society, including industry, government, and academia [2]. Achieving the full transformative potential of the current data deluge requires addressing new and open questions, especially with respect to the scalability of data creation, storage, and processing. The explosive increase in the volume of big data is even overtaking the exponential growth of Moore’s Law.

Moore’s Law, named for Intel’s co-founder Gordon Moore, states that transistor density (and consequently computing power) doubles roughly every 18 months. This trend has enabled an ongoing computing golden age for the past half-century. Clearly no exponential growth can be sustained indefinitely, and we therefore must look beyond transistor counts in order to realize future enhancements in computing resources.

While transistor density in silicon has continued to increase, manufacturing costs have also commensurately increased with the higher precision required for high-density chips. These trends have recently forced Intel to slow the rate at which it decreases its transistor size [3], implying that we need to find new ways to satisfy the ever-increasing demand for computing resources. Thus, future advances in computing must rely more heavily on application-side optimization and resourcefulness, rather than on raw transistor counts. Even without smaller transistors, CPUs can be improved through better design, and applications can be written to better utilize the underlying hardware.

In this dissertation, we identify and explore two techniques for designing *super-scalable* data processing algorithms, that are able to maintain the aggressive performance improvements predicted by Moore’s Law, despite the slowing pace of transistor shrinkage. In other words, our work

provides *life-extension* strategies for Moore’s Law.

Our contributions span both the algorithms domain and the hardware domain. First, we develop new *compression-aware* algorithmic techniques that achieve better scaling of computing resources by operating directly on compressed data. Second, we utilize specialized automata-based architectures that yield substantial acceleration on targeted classes of problems.

This research can help algorithm developers as well as hardware designers to increase the practicality and efficacy of next-generation big data processing techniques. This will enable the design of more sophisticated and efficient algorithms and specialized hardware, and could fundamentally change the way in which organizations and governments collect, process, and utilize large datasets. Algorithms permeate all areas of technology, engineering and computer science, and they serve as a promising approach to grappling with issues that arise with processing and mining large datasets. The compression-aware algorithms and automata-based algorithms framework developed here can therefore help usher in the next generation of new super-scalable methods for practical and realistic big-data scenarios.

1.1 Compression-Aware Algorithms

Data compression is typically used in storing and managing massive data sets. Yet, while much data is stored in compressed format, very few classical algorithms are able to process compressed data. We see this disconnect as an opportunity to mitigate the growing gap between dataset sizes and processing capability [4, 5].

In Chapter 2 we investigate a general framework for compression-aware algorithms which operate directly on compressed data sets. Previous approaches either require decompressing the data before operation, or else require the compression to include metadata. The former technique forfeits potential efficiency benefits offered by the compression, while the latter imposition diminishes the benefit of compression and constrains the applicable algorithmic approaches.

To overcome these problems we design algorithms to operate directly on compressed data, without the need for metadata. Each algorithm’s speed and memory space performance dramatically improve with the input’s compressibility (i.e., descriptive complexity). This improvement derives from leveraging highly repetitive or parametrically specified input structures, leading to much smaller inputs (and outputs), and enabling algorithms to manipulate very large composite objects while interacting only with their succinct descriptions. For example, Figure 1.1 shows highly repetitive real-world structures, which are well-suited for a compression-aware approach.

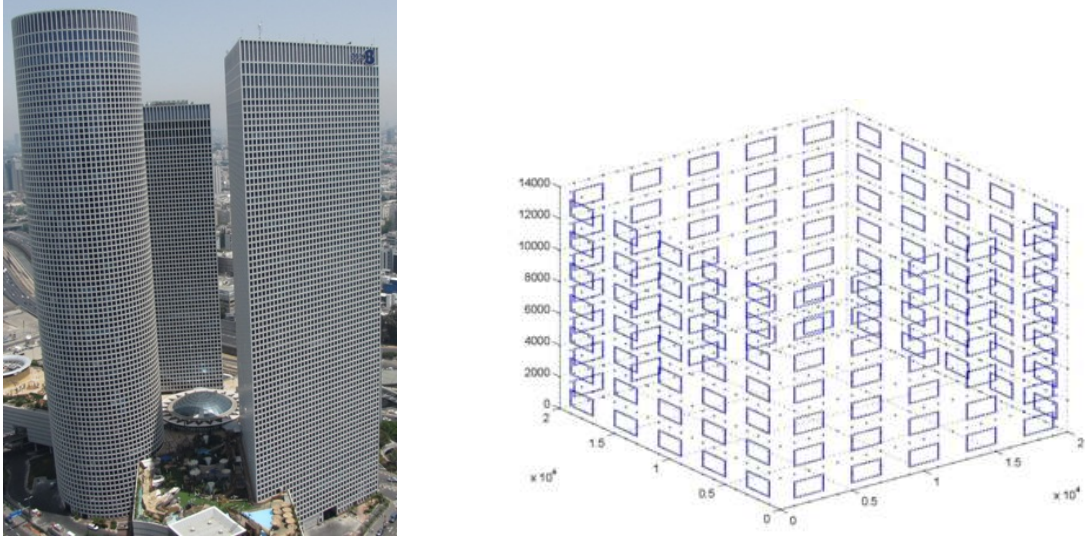


Figure 1.1: Actual buildings (left), and a corresponding highly compressible / regular CAD model (right).

By way of analogy, one can intuitively appreciate the benefit of operating on compressed data (rather than on raw data), by considering the benefits of shipping unpopped popcorn (rather than popped corn). Transporting unpopped corn is much more efficient, since it takes up significantly less space. A typical popcorn kernel expands by $40\times$ when popped, thus transferring unpopped kernels requires $1/40^{\text{th}}$ the “effort” required to transport the popped kernels. The advantage gained here is twofold: (1) by shipping the unpopped corn we could use a smaller truck, and (2) it takes less time to load and unload the truck because every crate of popcorn contains more kernels.

Similarly, operating on compressed data yields analogous benefits. Because the data is in its compressed form, the memory requirements of our algorithm will be greatly reduced (our truck can be smaller). Also, operations on the compressed data have greater impact, since the increased entropy of the data means that the algorithm makes more progress with each bit that it observes or manipulates (i.e., each crate stores and moves more kernels).

Moreover, the advantage of operating on compressed data could extend far beyond a constant-factor improvement (i.e., the $40\times$ advantage of shipping unpopped corn). In fact, highly-compressible data can sometimes be compressed by an exponential amount relative to its original size (e.g. highly repetitive circuit diagrams such as memories). This means that even if Moore’s Law slows down to a polynomial growth (rather than exponential growth), a compression-aware algorithm leveraging this polynomial speedup can still achieve exponential speedup on highly-compressible data, relative to computing on the equivalent uncompressed raw data.

| | | Instruction | |
|------|----------|-------------|-------------------|
| | | Single | Multiple |
| Data | Single | SISD (CPU) | MISD (AP) |
| | Multiple | SIMD (GPU) | MIMD (Multi-core) |

Table 1.1: Overview of techniques for parallelism in computing according to Flynn’s Taxonomy

1.2 Automata Processing

The ubiquity of web connectivity has caused runaway increases in the size and number of massive data sets as well as epic volumes of web traffic. Meanwhile, the computer architecture community has partially shifted away from serial von Neumann CPU designs toward adopting heterogeneous computing models, using multiple types of architectures as accelerators for performance-critical tasks. Typical single-core CPUs are ill-suited for the efficient computation of many tasks, as they are only able to operate sequentially over a single piece of data at a time. This means that a CPU’s efficiency tends to be bottlenecked by the speed of it’s slowest operation. Many applications can gain dramatic performance benefits through parallel execution, motivating the design of parallel architectures.

There are several different approaches to achieving parallel computation, typically arranged into a taxonomy originally outlined by Michael Flynn [6], as summarized in Table 1.1. Graphics processing units (GPUs), which process a single instruction in parallel across multiple data points, can perform vector-parallel floating point arithmetic to yield sizable speedups on appropriately parallelizable tasks. We call this technique for parallelism “single instruction multiple data” (SIMD). Many-core CPU designs, such as Xeon Phi, utilize a massive number of parallel tandem CPUs to achieve performance benefits by their ability to perform multiple independent tasks in parallel. We call this parallelism technique “multiple instruction multiple data” (MIMD). Field programmable gates arrays (FPGAs) and application-specific integrated circuits (ASICs) enable highly specialized hardware designs to optimize domain-specific tasks.

We are currently seeing an increasing interest in automata-based architecture designs as a new addition to the pantheon of specialized co-processors, such as the Micron Automata Processor (AP) [7]. The promise of these architectures lies in their ability to efficiently simulate non-deterministic computations (in the form of non-deterministic finite automata) in hardware, thus providing acceleration in a “multiple instruction single data” (MISD) manner, corresponding to the upper-right cell in Table 1.1.

These new automata-based architectures have shown only a limited range of applications thus

far, mostly restricted to pattern matching tasks. In this dissertation we argue that this limitation stems from the relative newness of the architecture and the still-fledgling progress in application development, rather than inherent restrictions in the capabilities of the architecture itself. In other words, our work provides new hope that automata-based architectures have a potentially bright future in addressing important problems in big data.

Historically, GPUs required substantial development before they earned their current position as critical subcomponents for supercomputing. In fact, GPUs were initially introduced exclusively to accelerate only specific graphics-related tasks. It was only some time later that researchers discovered that GPUs were able to accelerate much larger categories of problems (e.g. in genomics, machine learning, self-driving cars, etc.), so long as these tasks could be adapted for SIMD -type parallelism. Similarly, automata-based architectures deserve similar diligence in order to identify and investigate the breadth of problems that can be optimized by automata-based MISD -type acceleration.

In Chapter 3 of this dissertation we help to refine the properties of automata-based computation and compare how it measures up to existing parallel processing techniques. Our theoretical analysis reveals an upper bound on the capabilities of the architecture, namely showing that it cannot computationally decide any language that is non-regular. We also show that these architectures are likely to be more efficient for certain problems beyond non-deterministic finite automata (NFA) simulations, due to their more-complex transition behavior.

In Chapter 4 we investigate an important practical application, namely an automata-based pseudorandom number generator, which can be implemented on an automata-based processor such as the Micron AP [8] (this work also resulted in a Patent Application [9] which is included in Appendix D). This result suggests that our current applications for automata-based processors are still incomplete in their exploration and full utilization of the new hardware’s promising capabilities. Interestingly, whereas all previous applications have used automata processing for pattern discovery, our pseudorandom number generator uses automata processing to obscure patterns. This counter-intuitive mismatch should encourage expanding our exploration of applications for automata computing to additional new domains.

1.3 Lessons and Future Directions

This dissertation explores super-scalable big data processing approaches through the design of compression-aware algorithms and hardware-accelerated automata-based implementations. We give several examples of each of these proposed techniques and apply them to particular

application areas. Many additional application domains could benefit from these general algorithmic approaches, and Chapter 5 enumerates some of these. We encourage the reader to use this dissertation as a guide to further investigate the capabilities and limitations of designing future super-scalable algorithms using such techniques.¹

¹The PhD defense presentation video is available at <https://www.youtube.com/watch?v=GP2rm0z3ebI>. The PhD defense presentation slides are available at <http://www.cs.virginia.edu/~njb2b>, and are also reproduced below in Appendix G.

Chapter 2

Compression-Aware Algorithms

While massive datasets are often stored in compressed format, most algorithms are designed to operate on uncompressed data. We address this growing disconnect by developing a framework for compression-aware algorithms that operate directly on compressed datasets. Synergistically, we also propose new algorithmically-aware compression schemes that enable algorithms to efficiently process the compressed data. In particular, we apply this general methodology to geometric / CAD datasets that are ubiquitous in areas such as graphics, VLSI, and geographic information systems. We develop example algorithms and corresponding compression schemes that address different types of datasets, including strings, pointsets and graphs. Our methods are more efficient than their classical counterparts, and they extend to both lossless and lossy compression scenarios. This motivates further investigation of how this approach can enable algorithms to process ever-increasing big data volumes.

2.1 Introduction to Compression-Aware Computation

The potential benefits of the technique are four-fold. First, by operating on compressed data an algorithm may easily take advantage of the data regularity utilized to gain time and space benefits commensurate with the input compression. Second, an algorithm is generally measured by its performance on some worst case or average case analysis. By designing algorithms to run on compressed data we are able to express resource consumption in terms of the size of the input compression, a greater indicator of its success in exploiting the data's regularity. Third, algorithms on compressed data gain benefits from simply dealing with less data. This allows for a direct translation from the space benefits achieved by the compression into time benefits for an algorithm. Fourth, the size of the compression grows much more slowly than the volume of the data it

represents. Therefore as the amount of data to process increases, the trend of benefit gained by operating on compressions outpaces the growth of the data, causing a net decrease in the average amount of computation required per bit of raw data.

When designing compression scheme without consideration for the operations that are expected to run efficiently on the compressed data, the exclusive metrics for success are the compression run time, and the resulting compression ratio. However, when considering compression-aware algorithms, those metrics seem too miopic. As compression schemes aggressively pursue the maximization of compression ratios, this typically results in more cryptic forms for the compressed data (as is intuitively predictable). Unfortunately, such extremely cryptic compressions are typically ill-suited for efficient compression-aware manipulations.

For this reason, maximizing the potential for developing efficient compression-aware solutions requires the *co-design* of compression schemes and the algorithms to be run on the compressed data. The result may be compression schemes which trade-off mild sacrifices in compression ratio in favor of major improvements in terms of algorithmic complexity. In this era of ubiquitous massive data sets which grow faster than the available computing resources, maximizing the usefulness of these large data sets necessitates higher awareness and careful tuning of the trade-offs between compression ratio and overall algorithmic efficiency.

In this work, we devise compression-aware algorithms in numerous classical domains, namely sequence data, geometric data and graph-based data. To further bolster the efficacy of compression-aware algorithms, we also explore the design of algorithmically-aware compression schemes. These schemes are specifically designed to support a broad range of operations on compressed data. In other words, we propose to *co-develop* the compression schemes along with the corresponding algorithms, in order to further enhance the overall performance gains when they are used in tandem.

The intention is to run these compression-aware algorithms on already compressed data. We do not predict speedups by (1) taking raw data, (2) compressing it, then (3) running a compression-aware algorithm relative to running a standard algorithm on the raw data. Getting a benefit in this way would violate a “triangle inequality” of sorts, with this triangle being as in Figure 2.1. If compressing then running the compression-aware algorithm became faster than operating directly on compressed data, then that line from input to output could become the compression-aware approach. Instead, we gain performance benefits when data is *already* compressed. In this case we begin at the top of the triangle (compression), and go directly to output, and observe that this path is often shorter than the input-output path. A traditional approach must decompress the

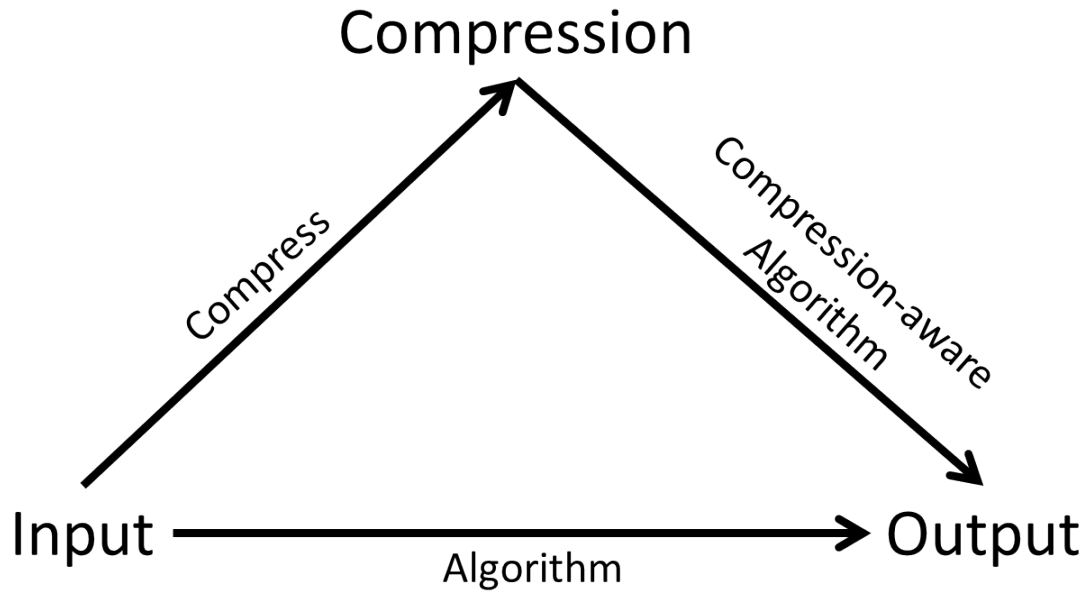


Figure 2.1: Running an algorithm on raw data should always be faster than compressing then running a compression-aware algorithm. We claim performance benefits when data is *already* compressed.

data before running a standard algorithm, thus it must traverse two edges in this triangle, from compression to input to output.

We begin in Section 2.2 discussing algorithms on compressed lists, given both accelerated compression-aware algorithms as well as an impossibility result. Next in Section 2.3 we give an overview of all problems and results considered in this chapter. Section 2.4 discusses prior art of compression-aware algorithms. Section 2.5 presents compression-aware algorithms for geometric data. Sections 2.6-2.8 discuss compression-aware algorithms for graphs. In the interest of readability of this chapter, all Lemmas and Theorems are left to Appendix 2.A, and all algorithm pseudocode is left to Appendix 2.B.

2.2 Case study: Compressed Lists

The prior art most similar to the contributions in this work relate to compression schemes and compression-aware algorithms for string data (these will be discussed in Section 2.4). In this section we explore these pre-existing well-understood compression schemes applied to the best-understood algorithmic domain (list comprehension) in order to demonstrate the cases where time/space benefits are attainable by these techniques, and cases where such gains are impossible. These results illustrate the necessity to wisely select compression schemes based on the algorithms one

wishes to run on the compressed data, and also the importance of compression scheme/algorithm co-design.

The first four schemes presented explore operations on numerical data and consider the classic problem of sorting and statistics. To begin, we consider sorting algorithms under the following four compression schemes: The first scheme represents a sequence in terms of a union of embedded arithmetic sequences, the second is Lempel-Ziv '77 (called LZ77 throughout), the third is Lempel-Ziv '78 (called LZ78 throughout), and the fourth represents an array as a context free grammar (called CFG throughout). In all cases below, the items being compressed are considered atomic, that is their representation cannot be split across terms in the compression. At the end of the section we show that this assumption is necessary to obtain any compression-aware speedup of sorting.

For sorting a list compressed by its arithmetic sequences, we present an algorithm which uses priority queues and runs in $O(n \log C)$ time, where n is the number of points to be sorted, and C is the number of sequences in the compression. This compares with the standard approach which would require decompression in time $O(n)$ and sorting in time $O(n \log n)$. We also present an algorithm which finds the k^{th} order statistic in $O(C \log C)$ time. The classical approach would require at least $O(k)$ to decompress, and $O(k)$ to find the k^{th} order statistic.

For sorting a list compressed by a context-free grammar we present an algorithm which finds the sorted sequence in $O(C \cdot |\Sigma|)$ time. Here, C is the size of the compression, which in this case is the total number of symbols in all of the grammar's substitution rules. This result has the advantage of being independent of the size of the uncompressed list. From here, we can produce a grammar for the sorted list which has size $O(|\Sigma| \log n)$, where n is the length of the decompressed list. The classical approach would require $O(n \log n)$ time to decompress and then sort.

For sorting a LZ77-compressed sequence, we present a sorting algorithm which operates in $O(C + |\Sigma| \log |\Sigma| + n)$. Where C is the compression size, n is the length of the sequence, and Σ is the set of all values in the list. In most instances $C \ll n$, thus our algorithm in practice achieves linear sorting as compared to the classical algorithm's $O(n \log n)$ worst-case performance. Additionally, at no cost to its asymptotic time complexity, the output can be expressed in LZ77-compressed form. We also present a way of indexing into the sequence in $O(C)$ time. By combining these two, we have a method for obtaining the k^{th} order statistic in $O(C + |\Sigma| \log |\Sigma| + n)$ time.

Sorting lists compressed using LZ78 is very similar to sorting context-free grammars. We present an adaptation of those results for LZ78 with the same running time: $O(C \cdot |\Sigma|)$ for the compression-aware algorithm as compared with $O(n \log n)$ for the classical approach.

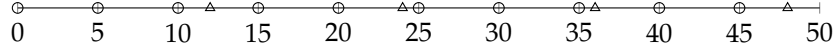


Figure 2.2: An example of a set of arithmetic sequences representing a set of numbers. The circular points represent the arithmetic sequence starting at 0 and extending every 5 points to 50, $\{0, 5, 10, 15, 20, \dots\}$. The triangular points represent the arithmetic sequence starting at 0 and extending every 12 points to 48, $A_2 = \{0, 12, 24, 36, 48, \dots\}$. The combined sequence is the union of the two, $\{0, 5, 10, 12, 15, 20, 24, \dots\}$.

2.2.1 Arithmetic Sequences Compression

Under this model of compression, a set of natural numbers $S \subseteq \mathbb{N}$ is represented by a union of underlying arithmetic sequences. That is, we say that $S = A_1 \cup A_2 \cup \dots \cup A_C$ for some number of arithmetic sequences C . Here, we will assume for the sake of simplicity that all arithmetic sequences start at 0, and subsequent values in arithmetic sequence A_i are all δ_i apart. So, for example, if $C = 2$ we may have arithmetic sequences $A_1 = \{0, 5, 10, 15, 20, \dots\}$ and $A_2 = \{0, 12, 24, 36, 48, \dots\}$ thus giving $S = A_1 \cup A_2 = \{0, 5, 10, 12, 15, 20, 24, \dots\}$, this example is illustrated in Figure 2.2. We will denote an arithmetic sequence with interval δ as $A(\delta)$. Additionally, to simplify notation, we will say that $A_i = A(\delta_i)$.

This representation may find use in representing building schematics. For example, an apartment complex may have in a wall: a stud every 500 centimeters, a window every 2000 centimeters, and a sewage pipe every 5000 centimeters. Rather than explicitly list the location of each of these objects, we may simply maintain a list of the intervals.

Priority Queue Sorting

The method we present for sorting an arithmetic sequences array uses priority queues. This method, called `Arith_sort` shown in Algorithm 1, begins by adding the first element of each sequence to a priority queue. The priority queue is therefore built in $O(C \log C)$ time. We then extract and output the minimum priority element from the queue, call this v .

Since it is possible for two points from the same sequence to be consecutive in the sorted list, we must maintain the invariant that the smallest element from each sequence always be present in the priority queue.

To accomplish this we first check from which sequence the extracted point came. Then, assume v came from $A(\delta')$, we insert the next point from the sequence $(v + \delta')$ into the queue. We repeatedly query the queue n times, until the n smallest elements are found in sorted order. By Theorems 2.A.1 and 2.A.2 this procedure runs in time $O(n \log C)$ time and with $O(n)$ space.

k^{th} Order Statistic

The regularity of the data under arithmetic-sequences compression allows for fast computation of the k^{th} order statistic. The intuition used for performing this calculation is that an arithmetic sequence acts similarly to a Poisson arrival process, which is the approach presented in Algorithm 2 called `Arith_Index`. We describe the action of this algorithm as *guess*, *check*, *count*. To begin, we consider each sequence as a Poisson process with rate $\lambda = \frac{1}{\delta}$, where δ is the interval of the arithmetic sequence. Conceptually, this is a shift from saying that “all points are δ units apart” to “ $\frac{1}{\delta}$ points appear every unit”.

We must now combine all of the processes (sequences) into a single process (sequence). The combined process can be seen as a single process with rate Λ where each event has one of C types. If the probability of a point being of type i is p_i , and $\sum_{i=1}^C p_i = 1$, then the combined process is equivalent to the combination of C slower concurrent processes, each having rate $\Lambda \cdot p_i$. Therefore we say that $\Lambda = \sum_{i=1}^C \lambda_i = \sum_{i=1}^C \frac{1}{\delta_i}$. Conceptually, we are summing together all the sequences’ arrival rates to Λ such that on average points appear every Λ units, or equivalently, on average points are $\frac{1}{\Lambda}$ units apart.

We can now find the location of position k in the combined sequence using this process. The expected arrival time of the k^{th} event is $\frac{k}{\Lambda}$, or conceptually if points are an average of $\frac{1}{\Lambda}$ units apart the k^{th} point is near position $\frac{k}{\Lambda}$. We label this *guess* of $\frac{k}{\Lambda}$ as g . We now must *check* the actual number of points which occur in $[0, g]$, which by Lemma 2.A.1 is between k and $k - C$. An illustration of the guessing is illustrated in Figure 2.3.

From here we use a method similar to `Arith_sort` presented in Section 2.2.1 in order to *count* up the remaining points to k . We add the next element from each sequence into a priority queue. Then we remove the lowest priority point, calculate the next point from that sequence and insert it into the queue, then repeat until we have reached $k - \sum_{i=1}^C \lfloor \frac{g}{\delta_i} \rfloor$ points. Theorem 2.A.3 shows this algorithm’s running time to be $O(C \log C)$, theorem 2.A.4 shows the space complexity to be $O(C)$.

Note that since Poisson processes require that there be 0 events at time 0, we must actually find event $k - C$ in the Poisson process to get the k^{th} point in the sequence. More details on Poisson processes as well as proofs to the claims mentioned can be found in [10].

2.2.2 Context Free Grammar Compression

Context free grammar compression serves as a generalization of many forms of dictionary-based string compression (e.g. LZ77, LZ78) [11], whereby a string is represented by a context free grammar which parses to a language containing exactly one string. A context free grammar (CFG)

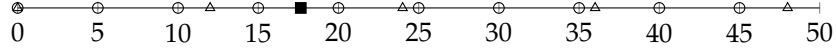


Figure 2.3: An example of a 5th order statistic query on arithmetic sequences representation of a set of numbers. The circular points represent the arithmetic sequence starting at 0 and extending every 5 points to 50, $\{0, 5, 10, 15, 20, \dots\}$. The triangular points represent the arithmetic sequence starting at 0 and extending every 12 points to 48, $A_2 = \{0, 12, 24, 36, 48, \dots\}$. The combined sequence is the union of the two, $\{0, 5, 10, 12, 15, 20, 24, \dots\}$. The combined arrival rate of the two sequences is $\Lambda = \frac{1}{5} + \frac{1}{12} = \frac{17}{60}$, this means the guess point is $\frac{5}{\Lambda} = 5 \cdot \frac{60}{17} = \frac{300}{17} \approx 17.65$, we mark this with a square.

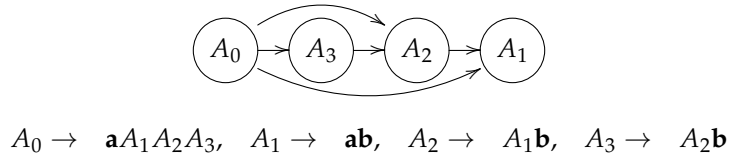


Figure 2.4: The dependency graph for a given context free grammar (this is the same grammar as is given in Section 2.2.2).

is a 4-tuple (Σ, V, S, Δ) , where Σ is a finite set of values called terminals, V is a set of variables, $S \in V$ is a special start variable, and Δ is a set of rules (for our purposes only one rule is permitted per variable, otherwise the language of the CFG will be of size greater than one). A rule is of the form $v \rightarrow s$ where $v \in V$ and $s \in (V \cup \Sigma)^*$ is a sequence of terminals and variables called the definition of v . The grammar is read by iteratively substituting variables for their definitions (starting with S) until only terminals remain. Using a CFG one is able to express a long string as a smaller set of these rules. For example, consider the string **aababbabbb**. This can be translated into the CFG:

$$A_0 \rightarrow \mathbf{a}A_1A_2A_3, \quad A_1 \rightarrow \mathbf{ab}, \quad A_2 \rightarrow A_1\mathbf{b}, \quad A_3 \rightarrow A_2\mathbf{b}$$

Sorting

Similar to LZ77.Sort for sorting LZ77 compressed arrays, Algorithm 5 (CFG.Sort) exploits the fact that all literals in the uncompressed list occur in the compression. Therefore we begin by first finding and sorting Σ . Next we turn the CFG into a dependency graph. For this, we say that if the variable $v_0 \in V$ has in its substitution rule $v_1 \in V$, then v_0 depends on v_1 . The dependency graph for the above context free grammar is shown in Figure 2.4. This graph must be acyclic (since otherwise the grammar would produce more than one string), thus a topological sort exists.

We then consider each literal as a vector of $|\Sigma|$ dimensions such that for the minimal element $\sigma_1 \in \Sigma$, $\sigma_1 \mapsto (1, 0, \dots, 0)$, and the next smallest element $\sigma_2 \mapsto (0, 1, 0, \dots, 0)$, and so on. As a

notational convenience we say that $\langle \sigma \rangle$ refers to the respective vector for symbol σ .

The final step is to follow backwards through the topological sort and sum up each symbol's respective vector upon discovery. In the example given we begin with $\mathbf{a} = (1, 0)$ and $\mathbf{b} = (0, 1)$. We then calculate the vector $\langle A_1 \rangle = \langle \mathbf{a} \rangle + \langle \mathbf{b} \rangle = (1, 0) + (0, 1) = (1, 1)$. We can then calculate $A_2 = (1, 2)$. Eventually we calculate the start symbol $A_0 = (4, 6)$, which says that in the decompressed string there are 4 \mathbf{a} 's and 6 \mathbf{b} 's.

We are now able to return to the user a context free grammar of size $|\Sigma| \log n$. This is done by writing a grammar in which for each value in Σ , we have $\log n$ variables, where each doubles the number of that letter represented. So, for example, if we wanted a grammar which represents the string $\mathbf{a}^8\mathbf{b}^{16}$, our grammar would be:

$$\begin{aligned} S &\rightarrow A_0 B_0 & A_0 &\rightarrow A_1 A_1 & A_1 &\rightarrow A_2 A_2 & A_2 &\rightarrow \mathbf{aa} \\ B_0 &\rightarrow B_1 B_1 & B_1 &\rightarrow B_2 B_2 & B_2 &\rightarrow B_3 B_3 \\ B_3 &\rightarrow \mathbf{bb} \end{aligned}$$

The time complexity of this sorting algorithm is shown by theorem 2.A.9 to be $O(C \cdot |\Sigma|)$. By theorem 2.A.10 the space complexity is $O(C)$.

2.2.3 Lempel-Ziv '77 Compression

With the LZ77 scheme [12], a compression is a sequence of terms each as one of two types: terminals and back pointers. A terminal is a character from the original set of values Σ of the array, and a back pointer is of the form $(back, length)$ where *back* is the index (in the uncompressed array) from which to start a subsequence and *length* is the number of characters to copy starting from *back*. As an example, consider the sequence (with spaces added for clarity): $\mathbf{a} \mathbf{b} (\mathbf{1}, \mathbf{2}) (\mathbf{2}, \mathbf{3}) \mathbf{c} (\mathbf{1}, \mathbf{5})$. The term $(\mathbf{1}, \mathbf{2})$ gives instruction to start at index 1 and copy 2 characters, giving $(\mathbf{1}, \mathbf{2}) = \mathbf{a} \mathbf{b}$. This decompresses into $\mathbf{a} \mathbf{b} \mathbf{ab} \mathbf{bab} \mathbf{c} \mathbf{ababb}$. A back pointer may have *length* larger than the depth of *back* (that is *back* – *current location*). In this case the referenced string is repeated to fill in the gap. For example, if we have the compression $\mathbf{a} \mathbf{b} (\mathbf{1}, \mathbf{6})$, this would decompress into: $\mathbf{a} \mathbf{b} \mathbf{ab} \mathbf{ab} \mathbf{ab}$.

Sorting

The intuition behind Algorithm 3 called LZ77.Sort, which sorts a LZ77 compressed array, is that for any sequence s , where $lz(s)$ is a LZ77-compression of s , the set of terminals present in s is equal to those in $lz(s)$. This means that to sort the decompressed array it suffices to sort all of Σ , then count the number of each character appearing in the decompression. Therefore we begin by first copying all literals to a list and sorting this list.

The next step is to count the number of each type of literal in the decompressed array. To do this, we count characters while mimicking the action of the decompression. First we scan through the compression to find the length of the deepest back pointer (the maximum *back* – *current location*). We then create a circular buffer of this size (call this variable *size*). Now we perform a decompression of the string, except whenever we would append a character to the decompression we instead write that character to the next space in the buffer, and iterate a counter for that character. When reading a back pointer we begin copying from that location in the circular buffer. Since the length of the circular buffer is the depth of the deepest back pointer, we are guaranteed the reference is in the circular buffer. Theorem 2.A.5 shows that the time complexity of this algorithm is $O(C + |\Sigma| \log |\Sigma| + n)$. The circular buffer allows this algorithm be run in space $O(C + \text{size})$, as shown is theorem 2.A.6. Note that in most practical implementations of LZ77 compression, the variable size is a fixed constant.

With the multiplicity of each character we can then return a LZ77 compressed sorted string in time $O(|\Sigma|)$. Assume $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_{|\Sigma|}\}$, and that σ_i has multiplicity m_i in the string. Then the compressed string becomes:

$$\sigma_1 (1, m_1 - 1) \sigma_2 (m_1 + 1, m_2 - 1) \dots \sigma_{|\Sigma|} (n - m_{|\Sigma|}, m_{|\Sigma|}).$$

Indexing

Algorithm 4, called LZ77_Index gives a method for finding the character at index i of a LZ77 compressed array. The algorithm keeps track of two read heads. The one labeled j reads the current location in the compressed list, the one labeled *count* represents the location in the decompressed string if all terms up to j were decompressed.

The first step in the algorithm is to scan the compression until we reach or pass the index i . This is done by advancing *count* by 1 if $LZ[j]$ is a literal (the term at position j in the compression), and by $LZ[j].length$ otherwise. If we reach index i on a literal then we simply output that literal and terminate. If we end on a back pointer then we update the query index i to become $LZ[j].back + LZ[j].length - (count - i) - 1$, then scan backward in the compression to find it. Again, if we end on a literal then we output that literal. Otherwise we repeat until we reach a literal. As an example, consider the compression **a b (2,1) (1,3) (4,3)** at index $i = 8$. The back pointer **(4,3)** occupies positions 7, 8, and 9 in the uncompressed string. Therefore to find position 8 we look backward for position 5 in the uncompressed string. Position 5 is within the **(1,3)** term, which is not a literal. Therefore we again look backward to position 2, which is **b**.

The most difficult step in the algorithm deals with those back pointers where the depth of the reference is less than the number of characters to be copied. This condition is handled in lines 11

and 12 of Algorithm 4. We know that we are this situation if $i > \text{count} - \text{LZ}[j].\text{length}$, as this says that our new index is still within the back pointer referenced by j . If we are in such a situation we first figure out the depth into the copy (given by $i - \text{LZ}[j].\text{back}$). We then must find the length of the string copied (given by $\text{count} - \text{LZ}[j].\text{length} - \text{LZ}[j].\text{back} + 1$). We are then able to figure out how far to go into the copied string by performing

$$(i - \text{LZ}[j].\text{back}) \bmod (\text{count} - \text{LZ}[j].\text{length} - \text{LZ}[j].\text{back} + 1)$$

This new number is the distance we must go from the location of the back reference ($\text{LZ}[j].\text{back}$), thus our new index becomes

$$((i - \text{LZ}[j].\text{back}) \bmod (\text{count} - \text{LZ}[j].\text{length} - \text{LZ}[j].\text{back} + 1)) + \text{LZ}[j].\text{back}$$

Theorems 2.A.7 and 2.A.8 show that the time and space complexity of this algorithm as described are both $O(C)$.

2.2.4 Lempel-Ziv '78 Compression

In addition to the LZ77 compression scheme presented above, Lempel and Ziv in 1978 presented a secondary compression scheme (here on out called LZ78) [13]. Each term in this scheme is a pair of a natural numbers $i \in \mathbb{N}$ and a character $\sigma \in \Sigma$. The rule for decompression is to copy the sublist represented by term i in the compression ($i = 0$ represents the empty string), then append σ . For example **(0,a) (1,b) (0,b) (2,a) (3,a) (2,b)** becomes **a ab b aba ba abb**.

Sorting

Algorithm 6 (LZ78.Sort) acts similarly to CFG.Sort in that we sort the symbols, and then accumulate vectors representing symbol multiplicity. As before we sort Σ . We then define $\langle \sigma_i \rangle$, where σ_i is the i th symbol in sorted order, to be a $|\Sigma|$ -dimensional vector where all terms are 0, save the i th term which is 1. Thus $\langle \sigma_1 \rangle = (1, 0, 0, \dots)$.

LZ78 compression behaves as a context-free grammar compression with the restriction that only a single variable and a single literal be present in each rule. The example above can be expressed as CFG as follows:

$$\begin{aligned}
& A_0 \rightarrow A_1 A_2 A_3 A_4 A_5 A_6 \quad A_1 \rightarrow \mathbf{a} \quad A_2 \rightarrow A_1 \mathbf{b} \\
(0,\mathbf{a}) (1,\mathbf{b}) (0,\mathbf{b}) (2,\mathbf{a}) (3,\mathbf{a}) (2,\mathbf{b}) \equiv & A_3 \rightarrow \mathbf{b} \quad A_4 \rightarrow A_2 \mathbf{a} \quad A_5 \rightarrow A_3 \mathbf{a} \\
& A_6 \rightarrow A_2 \mathbf{b}
\end{aligned}$$

With this construction all variables in this CFG are already in topological-sorted order. Therefore the algorithm for sorting such a compression is a simplified version of `Sort_CFG`.

We first create a second list of terms to parallel the compression. In this list, instead of pairs, we use the $|\Sigma|$ -dimensional vectors as described above. Next we scan through the compressed sequence. For each term we first check if $back = 0$. If so then we know that this term in the compression represents a single character, call this σ , and add into the array at this index the vector $\langle \sigma \rangle$. Otherwise if $back > 1$ we add together $map[back]$, the count for the referenced string, and $\langle \sigma \rangle$, where σ is the character to append for this term. Similar to `CFG_Sort` this algorithm has running time $O(C \cdot |\Sigma|)$ and space $O(C)$, by theorems 2.A.11 and 2.A.12.

2.2.5 Lessons

This case study shows that in certain circumstances we can get asymptotic reductions in run time even for problems with well-known worst case lower bounds, such as sorting and statistics. As can be seen in these cases, however, these bounds are restricted to the case where each term in our list is compressed atomically (meaning a single list item's description is not allowed to be broken for the purposes of compression, so they may be sorted a priori). Without this restriction, dictionary compression schemes cannot attain any asymptotic speedup over the $n \log n$ worst case lower bound for sorting. In other words, it is impossible to improve the worst-case $n \log n$ time complexity of sorting by using a compression-aware sorting algorithm on a string representation of a list that has been compressed by a dictionary compression (such as LZ77, LZ78, or CFG). Comparison-based sorting of a dictionary-compressed list is $\Omega(n \log n)$.

Theorem 2.2.1. *Sorting a CFG-compressed list of length n with non-atomic items requires $n \log n$ time.*

Proof. We will show that the required run time of sorting (in terms of list length) is independent of the size of the list's CFG compression.

Consider the case where we have a start variable $A_0 \rightarrow A_1 \dots A_n$ such that each of the variables A_1, \dots, A_n produces a unique string (no two variables resolve to the same string), and ends with the list delimiter (e.g. $'.'$). This means that any arbitrary reordering of the concatenation of $A_1 \dots A_n$ produces a unique string in the list. Since there are $n!$ different orderings, this means that by a

standard decision tree argument for any algorithm there will be at least one list requiring $\Theta(n \log n)$ time to sort. \square

This theorem provides an important lesson about any attempt to analyze algorithms on compressed data, it is sometimes impossible to realize acceleration using a compression-aware algorithm relative a naive decompress-first approach.

In other words, if one wishes to compress data in a way that allows for compression-aware techniques, the algorithms to be run must be considered when making a choice of compression scheme, as a mismatch may provide no benefit. For this reason, simply using string compression techniques from the prior art and applying them to non-string data (e.g. graphs) is unlikely to provide for speedups, making the prior art (see Section 2.4) unsuitable for compression-aware approaches for non-string data. This is why we consider graph-specific compressions for graph data, and pointset-specific compressions for geometric data. Furthermore, this trend highlights the importance of algorithm-compression co-design, as we demonstrate in our discussion of pointset data.

2.3 Problem Statement and Results

Algorithms for Compressed Graphs We consider the graph problems of topological sort and bipartite assignment performed over compressed graphs. For one scheme (hierarchical compression) we also cover single source shortest path, as this is the only scheme we present which operates on weighted graphs. We consider algorithms under the following four compression schemes: The first is a graph interpretation of the Re-Pair compression scheme, the second is a hierarchical compression scheme, and the third is the compression scheme presented by Boldi and Vigna as part of the WebGraph Framework (called BV throughout).

For a graph compressed using Re-Pair we perform bipartite checking and topological sort in $O(C)$ time (Section 2.6). We use the fact that a Re-Pair compression of a graph can also very easily be conceptualized as a graph and be operated on directly. The improvement we obtain for both algorithms is over the $O(|V| + |E|)$ time for the classical approach.

For a graph compressed using hierarchical compression we perform topological sort and single source shortest path in $O(C + |V|)$ and $O(\sum_{\text{component } c} |V_c|^3 + |V|)$ respectively (Section 2.7). The former is an improvement over the $O(|V| + |E|)$ running time of the classical approach. The latter usually gives a benefit over the classical approach which runs in $O(|E| + |V| \log |V|)$.

We present an algorithm which performs bipartite checking on a BV compressed graph using disjoint set data structures. This algorithm runs in $O(|V| + s)$ where $|V|$ is the number of vertices

in the graph, and s is the total number of sequences given in the compression scheme (Section 2.8). This is a benefit over the running time of the classical approach of $O(|V| + |E|)$.

The performance of the web graph compressions is limited by the inability for the schemes to reduce vertex size. For this reason we believe our results for Re-Pair and BV compression to be near optimal. We believe that there are improvements to be made on the hierarchical compression results.

Algorithms for compressed pointsets Finally, we explore algorithms and compressions over geometric (Cartesian pointset) data. To begin, we present a novel compression scheme which represents points of arbitrary dimension by embedded regular and collinear subsets. We give both lossy as well as lossless algorithms for compressing pointsets in this way. In both cases we provide an algorithm to compute the convex hull of the points in $T(L)$ time where T is the runtime of a traditional convex hull algorithm and L is the number of lines in the compression, perform nearest neighbor queries in $O(L)$ time where L is the number of lines in the compression, perform Manhattan and Euclidean range queries in $O(L \cdot d)$ time where L is the number of lines in the compression and d is the dimensionality of the pointset, and query for polytope (the many-dimensional analog of a polygon) membership in $O(L \cdot F)$ time where L is the number of lines in the compression and F is the number of faces on the polytope. For each algorithm we also provide error bounds in the case the points were lossy compressed.

2.4 Related Work

There has already been limited research on compression aware algorithms, and it is well-known that there is a time and space benefit in the previously studied applications. One of the most well-explored areas is pattern matching on compressed strings, including both exact pattern matching [14, 15, 16, 17, 18], and approximate pattern matching [19, 20, 21, 22, 15, 23, 24]. Others have studied compression-aware edit-distance algorithms [25, 26, 27, 28, 29]. Computational biologists have also written genomics-specific algorithms to run on compressed human genomic data [30]. There have also been algorithms presented which act directly on JPEG-compressed images [31, 32]. Initial steps have been made in compression-aware algorithms on graph compressions by [33, 34].

In this work we seek to show that general-purpose algorithms can receive a benefit when run on compressed data. The following are the original bodies describing the schemes used:

Sequence Compressions Algorithms for detecting hidden arithmetic sequences have been presented in [1, 35, 36, 37]. Lempel-Ziv '77 was first presented in [12]. Lempel-Ziv '78 was first

presented in [13]. The third compression scheme considered, one in which a string is represented as a context free grammar, is presented in [17].

Graph Compressions The first scheme performs Re-Pair on a graph’s adjacency list, an approach first suggested in [38]. We next consider a hierarchical compression scheme similar to those used in VSLI design [39, 40]. Finally, we study a method presented by Boldi and Vigna in The WebGraph Framework [41].

2.5 Set-of-Lines Compression Scheme

We present lossy and lossless versions of the set-of-lines compression scheme for arbitrary-dimensional Cartesian pointsets. Our scheme compresses the data by searching for embedded arithmetic progressions of collinear points (i.e., equally-spaced) within the pointset (we call such a set a *regular line*). The compression algorithm then returns the set of equations of these lines, the distance between points, and the locations of their initial points, as illustrated in Figure 2.5. Alternative representations are possible without loss of efficiency.

We define the first point to be that on the line segment whose coordinates are greatest in lexicographic order. Define points $p_a = (a_1, \dots, a_n)$ and $p_b = (b_1, \dots, b_n)$ where the coordinates are listed based on a fixed arbitrary ranking of the axes. Let $p_a > p_b$ in the lexicographic ordering provided that for $m = \min\{i | a_i \neq b_i\}$ it holds that $a_m > b_m$. Given the equation for the line and the first point lexicographically this definition unambiguously determines the direction the remaining points must progress, as they must go toward $-\infty$ along the most significant axis which is not parallel to the line.

2.5.1 Compression Algorithm

We adapt the set-of-lines compression algorithm from two previously studied problems– the maximal regular subsequences problem [36] for the lossless compression, and the maximal ϵ -regular subsequence problem [1] for the lossy compression. At a high level, both algorithms will exhibit the same procedure. First we find all regular subsequences of the input pointset \mathcal{P} . Second we extend each to a maximal regular subsequence. Finally we select an approximately minimal set of these subsequences which fully cover the point set. The only difference between the lossy and lossless versions of the scheme will be whether the points required to be exactly regular subsequences to be group together, or only must be sufficiently close to being a regular subsequence.

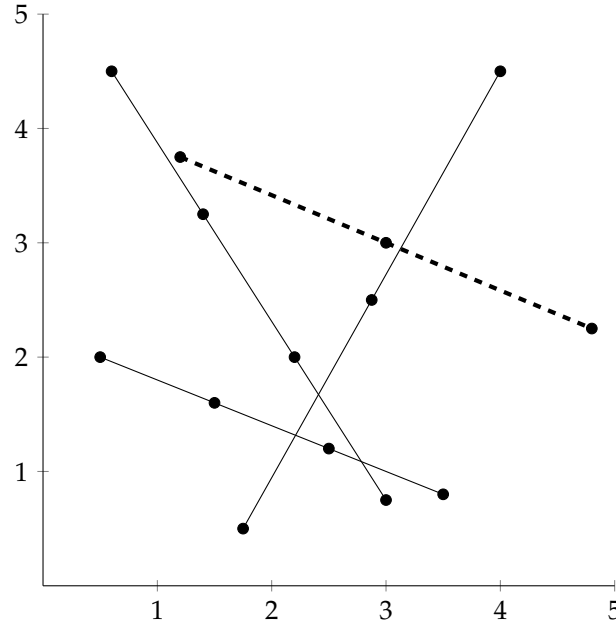


Figure 2.5: An example of a set of lines representing a pointset. The dashed bold line is defined by $y = -\frac{5}{12}x + 4.25$. Each point is ~ 6.34 units apart along this line, with the left-most point occurring at $(1.2, 3.75)$.

Lossless Compression

The basis for this scheme follows from the solving the *all maximal equally spaced collinear subsets problem*, which is solvable in $O(n^2)$ where n is the number of points [36]. The problem states that given a set of d -dimensional Cartesian points \mathcal{P} , find the smallest set of regular collinear subsets which contain all the points. An equally-spaced collinear set is a set of collinear points where each point on the line is distance δ from the point before it, in other words it is an arithmetic progression of points along a line. From an equally-spaced collinear subset, a maximal equally-spaced collinear subset can be constructed by extending the line and gathering all points along that line in the same arithmetic progression.

Finding the Maximal Regular Subsequences We provide a summary of $O(n^2)$ algorithm for solving the *all maximal equally spaced collinear subsets problem* here. First consider the one-dimensional case for this problem, which is to find all maximal arithmetic sequences in a set of numbers $\mathcal{P} = p_1, p_2, \dots, p_n$. Let p_1, p_2, \dots, p_n be sorted. We maintain three pointers into the list, A, B, C , where p_A , for example, represents the value which A references. Assume $A = i$, let $B = i + 1$ and $C = i + 2$. If $p_B - p_A > p_C - p_B$ then increment C , otherwise increment B . After exhausting all choices of B and C , move A one to the right.

If the left hand side and the right hand side above are equal, then the triple (p_A, p_B, p_C) represents an equally-spaced triple. This procedure is repeated for each $1 \leq i \leq n - 2$, and each iteration takes linear time, thus the algorithm runs in $O(n^2)$.

If this algorithm produces a triple (p_i, p_j, p_k) then we say $(p_i, p_j) \sim_{reg} (p_j, p_k)$ ((p_i, p_j) “relates to” (p_j, p_k)). The maximal equally-spaced subsequence containing (p_i, p_j, p_k) is therefore all points in the transitive closure of (p_i, p_j) under relation \sim_{reg} . To generalize this algorithm for collinearity to arbitrary dimension the only addition is to verify the collinearity of (p_A, p_B, p_C) , so we now say that $(p_i, p_j) \sim_{reg, coll} (p_j, p_k)$ provided (p_i, p_j, p_k) are equally spaced and collinear. This gives the final run time as $O(d \cdot n^2)$.

Lossy Compression

Here we present a lossy version of the set of lines compression scheme. This scheme is lossy in the sense that points may be nearly collinear and at nearly regular intervals (with ε Manhattan distance). An ε -regular set is a set of points which are all within ε of a regular line. More formally, an ε -regular subsequence of a point set is a subsequence $P = (p_1, p_2, \dots, p_n)$ of points such that there exists a regular subsequence $R = (r_1, r_2, \dots, r_n)$ of points where $\forall i, \Delta(p_i, r_i) \leq \varepsilon$, with Δ being Manhattan distance. This definition is illustrated in Figure 2.6. A maximal ε -regular subsequence is a regular subsequence such that $\forall p \in P - R$ neither $(p, p_1, p_2, \dots, p_n)$ nor $(p_1, p_2, \dots, p_n, p)$ are ε -regular subsequences. The *all maximal ε -regular collinear subsets problem* states that given a set of d -dimension Cartesian points \mathcal{P} , find the smallest set of ε -regular collinear subsets which contain all the points. This problem is the basis for the lossy compression scheme.

Finding the Maximal Regular Subsequences We provide a summary of a $O(n^{\frac{5}{2}})$ time solution to the *all maximal ε -regular collinear subsets problem* [1]. Note that this solution requires no pair of points be within 8ε distance of each other, otherwise the problem is intractable, as there would be an exponential number of maximal ε -regular collinear subsets. The first step is to generate all pairs of points from the point set, (p_i, p_j) . From here, find an additional point which is ε -regular with a pair of points (p_i, p_j) . Note that given the two points (p_i, p_j) there is a square region of size $8 \cdot \varepsilon$ in which a point may belong.

Thus by partitioning the plane into a grid of size $8 \cdot \varepsilon$ membership of a point in such a region for any pair of lines may be determined in $O(n \log n)$ time. From here, a marching procedure is performed whereby a pair of points is extended to a ε -regular triple to the right, then the leftmost point is dropped to form another pair. This is continued until the sequence cannot be extended further, and then repeated in the opposite direction. All points visited in this procedure define the

maximal ε -regular subsequence containing points p_i and p_j . In total this algorithm is shown to run in $O(n^{\frac{5}{2}})$ time.

From Maximal Regular Subsequences to Set-of-Lines

The output of the algorithms finding maximal (ε -)regular subsequences described above output sets of points representing all maximal (ε -)regular subsequences. Our goal is to use this output to produce a set-of-lines compression of the point set. Doing this will involve two steps: the first is to find an approximately minimal set of the subsequences which cover the point set, then (in the lossy case) we must compute the best fit line for the point set.

Given a set of maximal (ε -)regular subsequences \mathcal{M} , we seek to find a minimal set $\mathcal{C} \subseteq \mathcal{M}$ such that $\bigcup_{C \in \mathcal{C}} C = \mathcal{P}$. Recall the definition of the set cover problem (which is famously NP-Hard): given a set of elements \mathcal{X} and a family of subsets of \mathcal{X} called $\mathcal{F} \subseteq 2^{\mathcal{X}}$, find a minimal set $\mathcal{C} \subseteq \mathcal{F}$ such that $\bigcup_{C \in \mathcal{C}} C = \mathcal{X}$. In order to pick the minimal set of (ε -)regular subsequences, we must solve a version of the set cover problem which has the mild restriction that $\forall C_i, C_j \in \mathcal{C}$ we have that $|C_i \cap C_j| \leq 1$. This restricted problem remains NP-Hard, so we employ a greedy heuristic to approximate (this heuristic is asymptotically optimal provided $P \neq NP$) [42]. The greedy heuristic is to always choose the set of points which contains the largest number of unselected points.

Finding a Fit Line for Lossy Case With a set of ε -regular subsequences covering the whole point set we now wish to find the equation of the line which approximates each set. To accomplish this it suffices to find the line of best fit in the Chebyshev sense, the line which minimizes the maximum distance to each point in a particular dimension (L_∞ -norm of the distance of the line to each point). Consider a two dimensional case where we pick the line which minimizes the maximum distance any point has from the fit line in the y dimension, call this δ . First we consider the point closest to the origin to be at $(0,0)$ and shift all other points accordingly. We then find some line $y = m \cdot x$ which minimizes $\delta = \max_i |y_i - m \cdot x_i|$ for all points (y_i, x_i) . Alternatively, we seek to find the c which yields the minimum δ satisfying $\forall i, \delta \geq |y_i - m \cdot x_i|$. This is solved by the linear program:

$$A^T = \begin{bmatrix} -1 & -1 & \cdots & -1 & -1 \\ -x_1 & x_1 & \cdots & -x_n & x_n \end{bmatrix}, b^T = \begin{bmatrix} -y_1 & y_1 & \cdots & -y_n & y_n \end{bmatrix}, x = \begin{bmatrix} \delta \\ m \end{bmatrix}$$

$$\text{Maximize: } \begin{bmatrix} 1 & 0 \end{bmatrix} \cdot x, \text{ Subject to: } A \cdot x \leq b$$

This linear program is run for each subsequence, and its output will give the equation for the best fit line which is to be used as the lossy compression of that point set. Note that this linear program

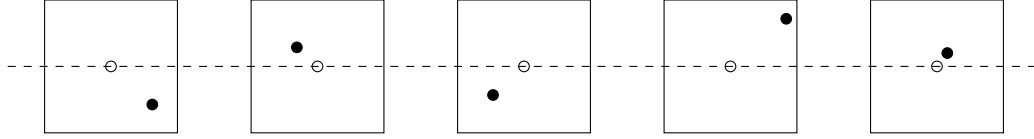


Figure 2.6: An example of an ε -regular point set (solid dots) whose points are within ε of the corresponding points of a regular point set (hollow dots). This figure appears in [1].

could be decomposed into several disjoint linear programs, one for each dimension. This fixes the dimensionality of the matrices, meaning this linear program runs in linear time relative to the number of points in the sequence [43].

2.5.2 Nearest Neighbor Queries

One natural problem to solve over any point set is, given a set of points $P = \{p_1, \dots, p_n\} \subseteq \mathbb{R}^d$ from a d -dimensional Cartesian space and a query point $p_q \in \mathbb{R}^d$, find $p_i \in P$ with minimum Euclidean distance to p_q . In the case that the points are compressed via set-of-lines the input given to the algorithm is a set of lines

$$L = \{(a_1, \dots, a_d, b), (p'_1, \dots, p'_d), n, \delta\} \subset \mathbb{R}^{d+1} \times \mathbb{R}^d \times \mathbb{N} \times \mathbb{R}$$

where (a_1, \dots, a_d, b) represents a line in d dimensions by the equation $\sum_{i=1}^d a_i \cdot x_i = b$, (p'_1, \dots, p'_d) represents the first point on the line, and n represents the number of regular collinear points on the line.

The operation of this algorithm relies on two steps for each line in $l \in L$, both computable in constant time. First, we find the nearest point on the continuous line (a_1, \dots, a_d, b) for l to p_q , next we perform a rounding procedure to the nearest point on l . This will yield the nearest point contained in l to p_q . After repeating this procedure for each $l \in L$ we have a new set of points of size $2 \cdot |L|$ which must contain the nearest point in P to p_q . In the end, we will have an algorithm which runs in time $O(|L|)$.

Without loss of generality we will assume that no line in L is perpendicular to the x_1 axis, as otherwise we can repeat this process according to the x_2 axis (or minutely rotate about the query point). We project each line onto this axis, this will give us a new line equation of $x_1 = 0$ and a new gap δ' , the new first point will be the x_1 coordinate of the original first point. Project the point on (a_1, \dots, a_d, b) which is nearest to p_q onto the x_1 axis, call this p' . We need to find the point on the new line (incident the axis) nearest to p' .

Our first step toward this end is to “left justify” the points on the line. That is, if the first

point on the line occurs at t , then we shift p' to $p' - t$. Now we need to find $m \in \mathbb{N}$ such that $m \cdot \delta \leq p' \leq (m+1)\delta$. This can be done quickly as we get $m \leq \frac{p'}{\delta} \leq m+1$. This means that, going back to the original line l , the nearest point to p_q is either the $\lfloor \frac{p'}{\delta} \rfloor^{\text{th}}$ or $\lceil \frac{p'}{\delta} \rceil^{\text{th}}$ point. We will add both points to a new point set, and run a nearest neighbor search on this new set of size $2 \cdot L$.

Lossy Error

In the case that the data was lossy-compressed, the nearest neighbor algorithm will remain the same, but errors in the results become unavoidable. Here, we look at three ways to quantify the magnitude of this error. The first two measure how “wrong” the response might be: Distance of the actual nearest point from the point returned by the query, and difference in distance from the query point to the returned point and the actual nearest neighbor. In the third case we analyze a modification to the above algorithm which, instead of returning a single point, returns a set of points, guaranteeing that the nearest neighbor is among them.

For this analysis we consider p_q to be the query point p_r to be the (post compression) returned point, and p_n to be the actual nearest neighbor (pre-compression). With δ being the distance from p_q to p_r , and ϵ being the lossy error. In the following discussion we will make heavy use of this lemma:

Lemma 2.5.1. *For a lossy-compressed set-of-lines pointset with error ϵ , a pre-compression point p will be within ϵ of the corresponding post-compression point, p' .*

Proof. This trivially follows from the definition of the compression scheme. If point p was further than ϵ from p' , then p could not have compressed to p' (note that the same bound holds in the case that we compress by Manhattan distance for similar reason, the generalization will extend to all the following results). \square

Distance from p_r to p_n We wish to answer: how far is the actual nearest neighbor (p_n) from the returned point (p_r)?

Theorem 2.5.1. *For a lossy-compressed set-of-lines pointset with error ϵ , a nearest neighbor query on point p_q will return a point p_r that is at most $2\delta + \epsilon$ from the true nearest neighbor p_n , where δ is the distance from p_q to p_r .*

Proof. The post-compression returned point p_r is δ from the query point. This implies that the post-compression distance of the actual nearest neighbor p_n is at least δ from p_q , as otherwise the point p_n would be equal to p_r and the distance would be 0. The point p_n cannot be more than $\delta + \epsilon$ from p_q , as otherwise p_n could not possibly be the nearest neighbor, as any pre-compression for

the point p_r would be closer than p_n . Thus the actual nearest neighbor p_n is at most $2\delta + \varepsilon$ from the returned point p_r . \square

Distance from p_r to p_q vs p_n to p_q We wish to answer: How much farther could the returned point (p_r) be from the query point (p_q) than the actual nearest neighbor (p_n) is from the query point (p_q)?

Theorem 2.5.2. *For a lossy-compressed set-of-lines pointset with error ε , a nearest neighbor query on point p_q will return a point p_r that is at most 2ε farther from p_q than is the true nearest neighbor p_n .*

Proof. Let δ be the distance from p_q to p_r . The the pre-compressed version of returned point p_r is at most $\delta + \varepsilon$ from the query point. The post-compression of the true nearest neighbor p_n must be more than δ from the query point, as otherwise the post-compressed nearest neighbor would have been the returned point). This means that the actual nearest neighbor is at least $\delta - \varepsilon$ from the query point. Thus the maximum difference in their distances is at most $\delta + \varepsilon - (\delta - \varepsilon) = 2\varepsilon$. \square

Returning a set of candidate nearest neighbors In this case we consider a modification to the above algorithm which, instead of returning a single nearest neighbor, it returns a set of candidate nearest neighbor points such that the actual nearest neighbor must be among these points. To achieve this we will return all points within a certain range of the query point p_q , where that range depends on the distance from p_q to the post-compression nearest neighbor (p_r) and the lossy error ε . Once this range is defined we must simply use the range query algorithm presented in the immediately following section (Section 2.5.3).

Theorem 2.5.3. *For a lossy-compressed set-of-lines pointset with error ε , the nearest neighbor p_n to a query point p_q must have a post-compression point that falls within distance $\delta + 2\varepsilon$ of p_q , where δ is the distance from p_q to its post-compression nearest neighbor p_r .*

Proof. In the worst case the post-compression returned point p_r may be ε away from its corresponding pre-compression point. This means that some post-compression point p_0 that is 2ε farther from the query point could have a corresponding pre-compression point that is closer to p_q than the pre-compression point corresponding to p_r , i.e. it is possible for p_0 to be p_n . This could occur in the case that the corresponding pre-compression point for p_0 is ε closer to p_q , while the corresponding pre-compression point for p_r is ε farther from p_q . \square

As mentioned above, finding the optimal compression for a pointset is intractable in the case that there are any points within distance 8ε apart. In the case that the pre-compression pointset

satisfies the requirement that no two points appear within any particular hypersphere of radius 4ϵ we can provide an upper bound on the number of points which belong to the range given in Theorem 2.5.3, in terms of δ , ϵ , and dimensionality.

This upper bound is given by the number of hyperspheres of radius 4ϵ can be packed within the hollowed hypersphere formed between radii $\delta - \epsilon$ and $\delta + 3\epsilon$. Unfortunately, these sphere-packing problems have not been solved in the general case, but only for dimensions 1, 2, 3, 8, and 24 [44]. The upper bound of the number of points, in the limit, will be the ratio of the volume of hollowed hypersphere and that of a $4 - \epsilon$ hypersphere ($V(\delta + 3\epsilon) - V(\delta - \epsilon)$).

2.5.3 Range Queries

Here we present two styles of range queries, Manhattan and Euclidean, on which the set-of-lines compression scheme allows for more efficient computation compared to an explicitly represented set of points. In the case where the data is not compressed each query will take linear time by number of points. Alternatively, in an application where repeated queries will be made a kd-tree data structure may be built on the points and each query will then take nearly linear time for high dimension, specifically it requires $O(n^{1-\frac{1}{d}})$ per query.

Manhattan Range Given a point and a range, we seek to find all points within the given range r_m by Manhattan distance, that is to give all points within a (45-degree axis-misaligned) hypercube of edge length $r = \sqrt{2r_m^2}$.

To begin, for ease of exposition, we shift all points so that this hypercube becomes axis-aligned. From there, this algorithm behaves similarly to the nearest neighbor algorithm. Consider the Manhattan range query on one particular line. As before, our first step will be to project all the points onto an axis x_i and “left-align”. The query hypercube will also be shifted left by the same margin, we will say that post left-alignment the query hypercube contains points within the range $(a, a + 2r)$.

From here we perform nearest neighbor-style queries on a and $a + 2r$ where we return the smallest point larger than a and the largest point smaller than $a + 2r$. This narrows down the range of the line to only that segment which is within the queried range on x_i . We then repeat this process for all lines on this dimension x_i and obtain a new set of “trimmed” lines from L . In the end all the trimmed segments will be completely contained by the query hypercube.

The point projection, left alignment, and nearest neighbor queries on each line can be computed in $O(d)$ time, where d is the number of dimensions of the points. This implies that the time required to trim all the lines for a particular dimension is $O(|L| \cdot d)$. This gives a final run time of $O(|L| \cdot d)$.

Euclidean Range Given a point and a range, we seek to find all points within the given range by Euclidean distance, that is all points within a hypersphere with radius r . For each line we can find which subset of points fit inside the sphere in $O(d)$ time. The size of the range is given by the length of the chord formed by intersecting the line with the sphere. The length of this region is given by $2 \cdot \sqrt{r^2 - \delta^2}$ (where δ is the distance from the center of the query region to the nearest point on the line). With the slope of the line we can then find the radius of the region after projecting onto axis x_i to be $\sqrt{r^2 - \delta^2} \cdot \cos(\theta)$ where θ is the angle of the line with the x_i axis, call this r' . We can then use the Manhattan distance subroutine on this range, as it is only one dimension, and return the trimmed lines. The run time of this algorithm is $O(|L| \cdot d)$.

Convex Polytope Membership The Manhattan distance procedure can be generalized to do inclusion queries on an arbitrary multi-dimensional convex polytope with F faces. We can find membership by, for each line segment in L and for each face of the polytope, projecting the segment onto the line defined by an orthonormal vector to the halfspace of the face, and then performing the Manhattan distance subroutine. This allows for queries to be done in $O(|L| \cdot F)$ time.

Lossy Error

For each of the membership queries above (Manhattan range, Euclidean range, and polytope membership) we modify the algorithm for the lossy case by returning two sets of points rather than one:

- G , a set of post-compression points whose corresponding pre-compression point is guaranteed to be included in the query
- M , a set of post-compression points whose corresponding pre-compression point may be included in the query

We find set G by shrinking the query region to exclude any points whose position allows for the possibility that its corresponding pre-compression point falls outside the query region. We find U by expanding the query region to include any points whose position allows for the possibility that its corresponding pre-compression point falls inside the query region. We now show how to shrink/expand the query region for each of the above algorithms.

Theorem 2.5.4. *For a Manhattan range query of distance δ , the set G of post-compression points whose corresponding pre-compression point is guaranteed to be included in the query is obtained by finding all pre-compression points within radius $\delta - \epsilon$, where ϵ is the lossy error.*

Proof. By Lemma 2.5.1, G is obtained by excluding any point within ε Manhattan distance of a face of the query's hypercube. This implies that we must simply shrink the hypercube by moving every face toward its center by ε . Thus G is obtained by performing a standard Set-of-Lines query for distance $\delta - \varepsilon$. \square

Theorem 2.5.5. *For a Manhattan range query of distance δ , the set M of post-compression points whose corresponding pre-compression point may be included in the query is obtained by finding all pre-compression points within radius $\delta + \varepsilon$ and subtracting off the points in set G , where ε is the lossy error.*

Proof. By Lemma 2.5.1, G is obtained by including any point within ε Manhattan distance of a face of the query's hypercube. This implies that we must simply expand the hypercube by moving every face away from its center by ε . Thus M is obtained by performing a standard Set-of-Lines query for distance $\delta + \varepsilon$ and removing all points found in set U . \square

Theorem 2.5.6. *For a Euclidean range query of distance δ , the set M of post-compression points whose corresponding pre-compression point may be included in the query is obtained by finding all pre-compression points within radius $\delta + \varepsilon$ and subtracting off the points in set G , where ε is the lossy error.*

Proof. Follows from Lemma 2.5.1 similar to the Manhattan case. This time we shrink the radius of the hypersphere defined by the query to $\delta - \varepsilon$. \square

Theorem 2.5.7. *For a Euclidean range query of distance δ , the set G of post-compression points whose corresponding pre-compression point is guaranteed to be included in the query is obtained by finding all pre-compression points of radius $\delta - \varepsilon$, where ε is the lossy error.*

Proof. Follows from Lemma 2.5.1 similar to the Manhattan case. This time we expand the radius of the hypersphere defined by the query to $\delta - \varepsilon$. \square

Theorem 2.5.8. *For a pointset lossy compressed using Set-of-Lines with Euclidean error ε , the set of points G for a Convex polytope membership query may be found by shifting each half space by ε in the direction of its orthonormal vector.*

Proof. Follows from Lemma 2.5.1 similar to the range query cases. This time we shrink the query polytope by moving in each face by ε in order to exclude any post-compression points whose corresponding pre-compression point may fall on the opposite side of the face. \square

Theorem 2.5.9. *For a pointset lossy compressed using Set-of-Lines with Euclidean error ε , the set of points M for a Convex polytope membership query may be found by shifting each half space by ε in the direction opposite its orthonormal vector, then removing all points found in G .*

Proof. Follows from Lemma 2.5.1 similar to the range query cases. This time we expand the query polytope by moving in each face by ϵ in order to include any post-compression points whose corresponding pre-compression point may within the halfspace defined by that face. \square

Theorem 2.5.10. *For a pointset lossy compressed using Set-of-Lines with Manhattan error ϵ , the set of points G for a Convex polytope membership query may be found by shifting each half space, with orthonormal given by $\langle x_1, \dots, x_d \rangle$, by $\frac{\epsilon}{|x_1| + \dots + |x_d|}$ in the direction of its orthonormal vector.*

Proof. We must move each hyperplane defining the half-spaces in the direction parallel to its orthonormal by sufficient distance in order to exclude all points within Manhattan distance ϵ , as per Lemma 2.5.1. To do this, consider a hypercube defining a region of fixed Manhattan distance that is tangential just to the inside of the hyperplane. The hyperplane must be shifted so that the center point of this hypercube now falls exactly on the hyperplane.

Assume, without loss of generality, that the hyperplane's orthonormal vector $\langle x_1, \dots, x_d \rangle$ begins at the origin. We need to find a point (x'_1, \dots, x'_d) which has Manhattan distance ϵ from the origin and lies along the vector $\langle x_1, \dots, x_d \rangle$.

The Manhattan distance for point (x'_1, \dots, x'_d) is given by $|x'_1| + \dots + |x'_d|$. For (x'_1, \dots, x'_d) to fall along vector $\langle x_1, \dots, x_d \rangle$ there must be some constant c such that $(x'_1, \dots, x'_d) = (c \cdot x_1, \dots, c \cdot x_d)$. Thus we must simply solve (note that c must be positive):

$$\begin{aligned} \epsilon &= |x'_1| + \dots + |x'_d| \\ &= |c \cdot x_1| + \dots + |c \cdot x_d| \\ &= c|x_1| + \dots + c|x_d| \\ &= c(|x_1| + \dots + |x_d|) \\ \frac{\epsilon}{(|x_1| + \dots + |x_d|)} &= c \end{aligned}$$

\square

Theorem 2.5.11. *For a pointset lossy compressed using Set-of-Lines with Manhattan error ϵ , the set of points M for a Convex polytope membership query may be found by shifting each half space, with orthonormal given by $\langle x_1, \dots, x_d \rangle$, by $\frac{\epsilon}{|x_1| + \dots + |x_d|}$ in the direction opposite its orthonormal vector, then removing the points found in G .*

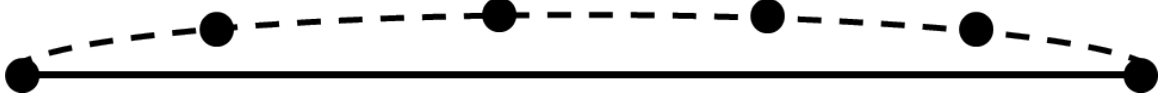


Figure 2.7: If all pre-compression points on one edge of compression-aware convex hull follow an arc extending no more than ε from that edge, the discrepancy in the number of points between the lossy-compressed convex hull result, and the non-compressed result could be arbitrarily large.

Proof. This follows similarly from the case for finding the set G , except now we must include any post-compression point which might fall within Manhattan distance ε of each plane defining a halfspace in the polytope. \square

2.5.4 Convex Hull

The convex hull problem asks that, given a set of n points $\mathcal{P} = \{p_1, \dots, p_n\}$, find the smallest convex polygon which contains all the points. If the point set \mathcal{P} has three or more collinear points then only the extreme points may be vertices on the convex hull. This implies that to find the convex hull when \mathcal{P} is represented in set-of-lines compressed format with L line segments one must simply compute the convex hull on the endpoints of each line segment. Thus if there is a convex hull algorithm used which runs in time $T(n)$ then it may trivially be converted into an algorithm for convex hull on L by extracting the endpoints in worst case $O(d)$ time and then computing the convex hull in time $T(|L|)$.

Lossy Error

In the lossy case, error for Convex Hull may be arbitrarily bad in number of vertices. One could imagine a set of points which follow an arc, all rounding to an ε -regular line, as shown in Figure 2.7.

2.5.5 Experimental Comparison

We implemented set-of-lines Range searches in Python, and evaluate the performance of set-of-lines over a traditional kd-tree approach. Tests were run on a randomly-perturbed grid of points (the grid was set slightly askew from axis-aligned by a random value), generated as a compressed point set. We varied number of lines by changing the number of columns, the number of points per line by varying the number of rows, and the dimensionality by varying the dimensionality of the perturbation. Next we decompressed the point set, constructed a KD-Tree from the resulting set of points, and then ran many range queries of varying diameter.

The tests were run on a 2016 Macbook Pro running MacOS 10.12.5, with a 3.1GHz dual-core Intel Core i5 processor, with 64MB of eDRAM, 8GB of 2133MHz LPDDR3 memory, and a 256GB PCIe-based SSD. We note that the kd-tree approach gains some performance benefit due to the kd-

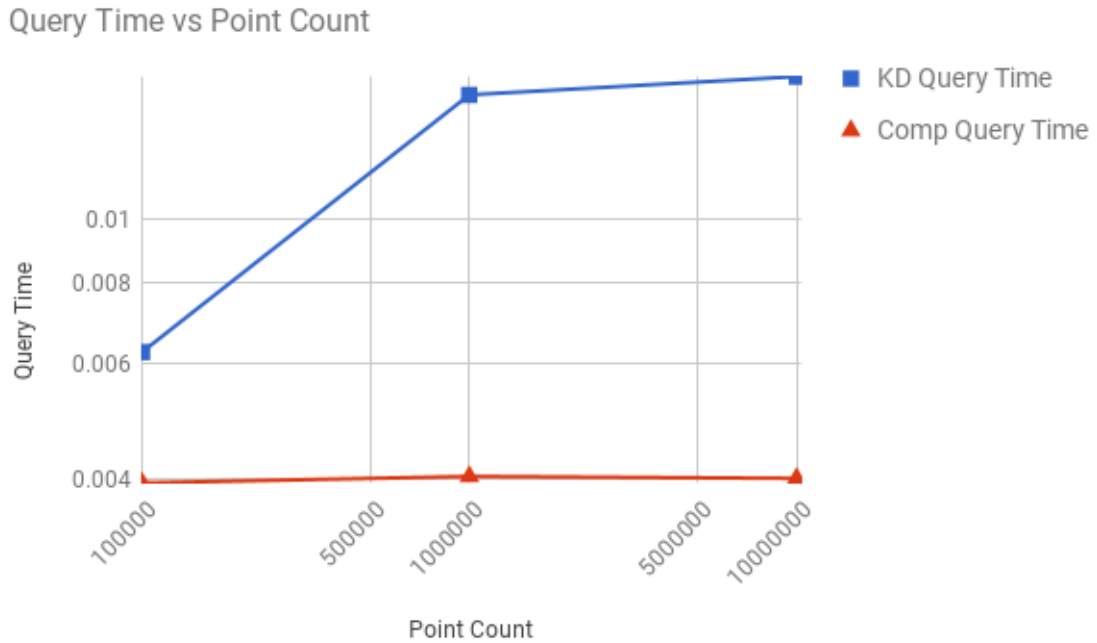


Figure 2.8: Runtime of compression-aware range queries do not depend on point count. Tests were performed on sets of 100 lines in 8 dimensions. Number of points per line was varied to vary total number of points.

tree implementation on Python being a wrapper for compiled C code, while the compression-aware approach is interpreted.

From these results we can see that the runtime of the compression-aware approach is independent of both point count (Figure 2.8) and number of returned points (Figure 2.9), but rather its runtime depends exclusively on number of lines (Figure 2.10).

Also note that we obtain speedups over the KD-tree algorithm in many scenarios, in spite of the KD-tree implementation being compiled and our compression-aware queries implementation being interpreted. This time benefit is especially pronounced when one considers the time required to build the KD-tree, and decompress the data (Figure 2.11)

2.6 Re-Pair for Graphs

Much of the existing work in graph compression schemes has been done on web graphs. In these graphs vertices represent web pages, and edges represent hyperlinks from one page to another. Web graphs are therefore directed graphs with unit-weight edges.

In [45] the authors introduce a graph adaptation of the Re-Pair dictionary compression scheme originally designed for strings [38]. In this scheme (as well as the Boldi-Vigna scheme to be

Query Time by Number of Returned Points

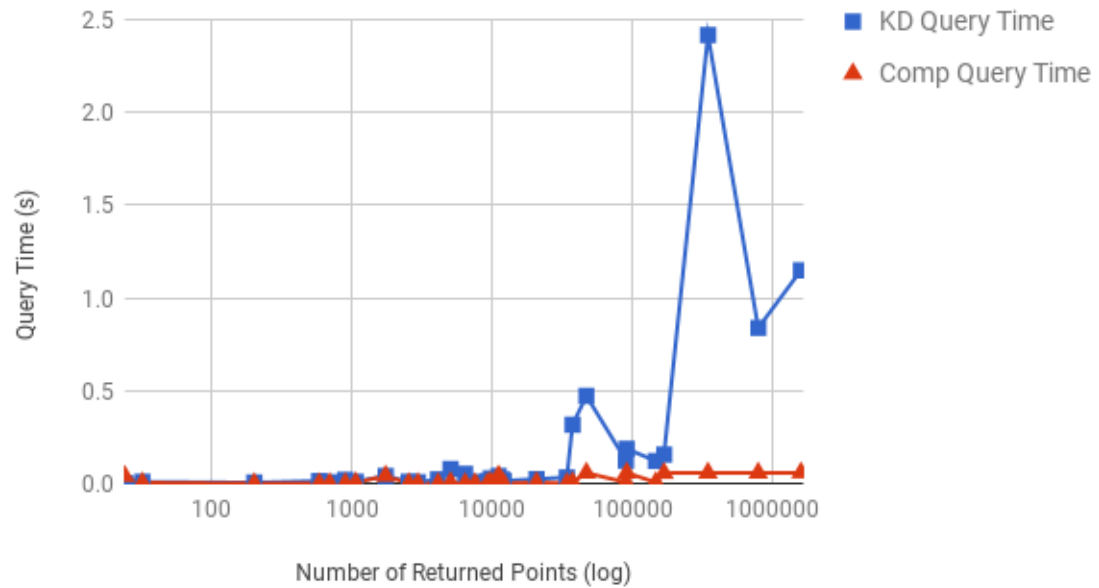


Figure 2.9: Runtime of Compression-aware range queries do not depend on number of points returned. Tests performed on sets of 10,000,000 points in 3 Dimensions. We varied the size of the query range in order to change the number of points returned in each query.

Compressed Query Time vs. Line Count

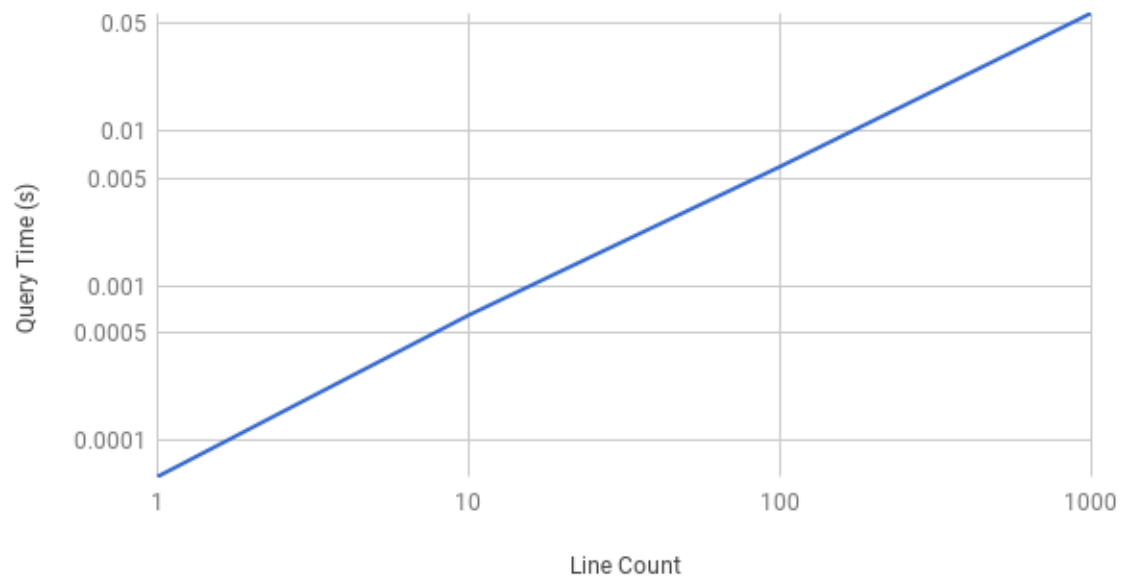


Figure 2.10: The query time for compression-aware range queries is linear by number of lines. Tests performed on sets of 1,000,000 points in 8 Dimensions.

KD-Tree Initialization

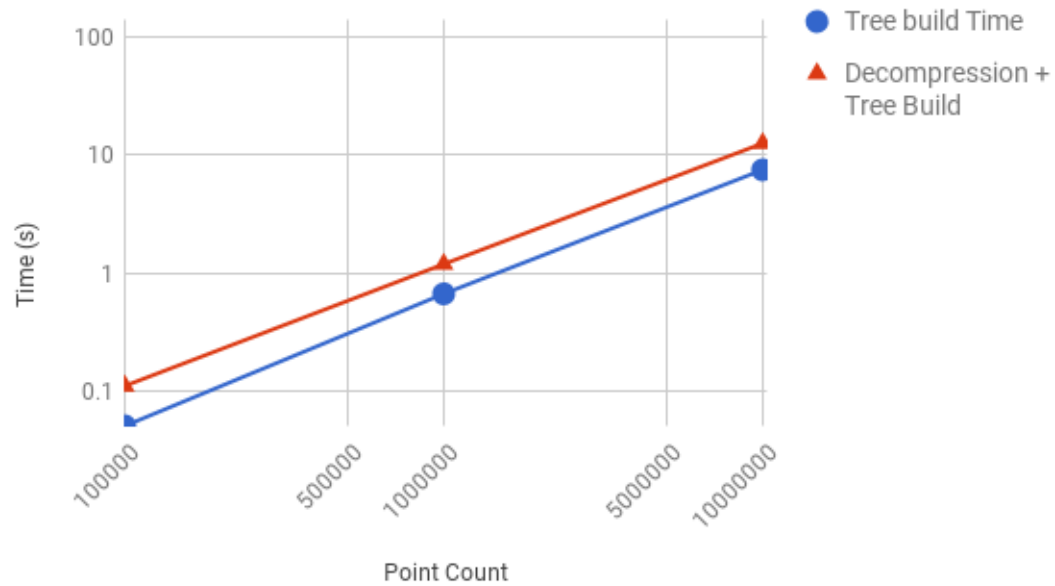


Figure 2.11: Set-up time for traditional range queries. The higher line represents total initialization overhead required (decompression time plus tree-building time). The lower line shows time required for building the KD-tree.

discussed) each vertex is assigned an index $i \in \mathbb{N}$. The graph is then represented in adjacency list form, and is sorted by index. This can be done using a list of lists, where list i contains the destinations of all edges from vertex i . Next the method finds the k most common pairs, and introduces a special symbol to represent each.

Afterwards, each appearance of each pair is replaced by its respective special character. This is repeated to satisfy one of two conditions: each pair in the adjacency list appears exactly once, or some pre-defined number of passes is reached.

The assignment of the indices for the vertices is done in such a way that the resulting graph has high locality and similarity. Locality is the property that edges from a given vertex are likely to end at another with a nearby vertex. Similarity is the property that vertices close to each other will share many vertices in their adjacency lists. This compression scheme is designed to exploit these properties.

Once complete we represent the compressed data itself as a graph. Each dictionary variable appears as a vertex in the graph with an edge to each vertex it represents, an edge whose destination is a dictionary vertex is assigned zero weight. This reveals the interesting advantage of this compression scheme. Since the compressed graph can itself be represented as a graph many

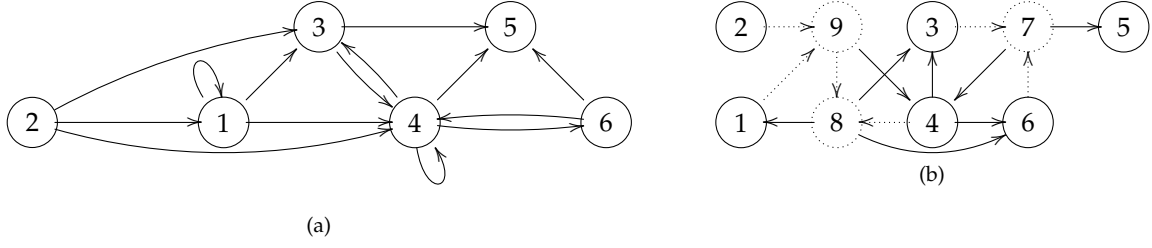


Figure 2.12: A sample graph (a) and its representation as a Re-Pair compressed graph (b). The dot-circled vertices are dictionary keys and the destination their outgoing edges are the edges which they replace in the adjacency lists. The dotted edges have 0 weight.

classical algorithms require only minor adjustment to operate on the compressed data. However, this compression scheme only serves to reduce the number of edges in the compressed graph, the number of vertices is larger than those in the original. For V_C and E_C the sets of vertices and edges in the compressed graph respectively, $|V_C| \geq |V|$ and $|E_C| \ll |E|$. Intuitively, the compression may be seen as a trade-off from a dense graph with fewer vertices in favor of a much more sparse graph with more vertices. Thus there may be no benefit to operating on the compressed data versus the uncompressed if the algorithm requires high time complexity in terms of number of vertices. In general the running time of graph algorithms is expressed in terms of some function of the number of vertices, $|V|$, and the number of edges, $|E|$. An algorithm run on Re-Pair compression should run faster than its uncompressed counterpart for any algorithm A , with running time $f(|V|, |E|)$, where $f(|V|, 1) \in O(f(1, |E|))$. Additionally, being a virtual node compression scheme, algorithms run on the Re-Pair graph compression scheme could utilize the technique shown by Karande, Chellapilla, and Andersen [34] for fast multiplication of adjacency matrices by a vector.

2.6.1 Topological Sort

The method in Algorithm 7 (`RePair_Topological`), given a Re-Pair compressed directed acyclic graph, returns a topological sort of its uncompressed graph. The topological sort of the uncompressed graph is embedded in that of the compressed graph. This follows since all vertices in the uncompressed graph are present in the compressed, and for any two vertices u, v where u appears before v topologically in the compression u must also appear before v topologically in the uncompressed counterpart. Thus we may perform a topological sort on the compressed graph, then remove the dictionary terms from the sort. This procedure takes time $O(|V_C|)$ (by theorem 2.A.13). It requires space $O(C)$ (by theorem 2.A.14).

2.6.2 Bipartite Assignment

Many of the algorithms performed on unweighted directed graphs, as web graphs are, can be written with minor modifications of breadth-first search (BFS) or depth-first search. To demonstrate the ability to perform meaningful applications of breadth first search we present an algorithm which determines whether or not a given compressed graph is bipartite, and if it is, gives the appropriate vertex colors. A bipartite graph is one that can be 2-colored, i.e., there is some way to give each vertex one of two colors such that there is no edge between two nodes of the same color. The classical algorithm for this problem performs a modified BFS which, when a vertex is discovered, assigns that vertex a color opposite of its parent's. If a vertex is already discovered and is the same color as its parent then the procedure terminates and returns false.

We modify this algorithm to work on Re-Pair compressed graphs. A similarly-modified BFS to run on the compressed graph is shown in Algorithm 8 (`RePair_Bipartite`). As before, when a vertex is first discovered it is given the color opposite its parent's. Additionally, we do a check if the discovered vertex is a dictionary variable. If so then the vertex is instead given its parent's color, as the dictionary variable acts as a "stand-in" for the parent vertex. This algorithm is shown in theorem 2.A.15 to have time complexity $O(C)$, and in theorem 2.A.16 to use space $O(C)$.

2.7 Component-Based Compression

Very large-scale integrated circuits (VLSI) are composed of a large number of hierarchically nested components, which may be compressed as a collection of graphs composed of yet smaller subgraphs. This hierarchy continues until some simplest set of graphs is reached (see Figure 2.13). The numerous graphs involved in this compression form a strict partially ordered set, with an irreflexive partial ordering. If a graph G_0 has a supervertex for another graph G_1 then we say that $G_0 > G_1$, or that G_0 is greater in the hierarchy than G_1 . This relation is irreflexive since no graph may have a supervertex representing itself. Similarly the relation is asymmetric; $G_0 > G_1 \wedge G_1 > G_0$ also implies a recursive definition. For a component graph G_c we say its set of edges is E_c and vertices is V_c .

2.7.1 Topological Sort

Our topological sort algorithm utilizes the subgraph-consistency of topological sort for hierarchical compressions. Subgraph consistency results from the observation that if a (super)vertex v_0 in a component graph G_0 appears before v_1 in a topological sort of G_0 , then it follows that all vertices in the graph defined by v_0 must appear before those in v_1 in a topological sort of the uncompressed

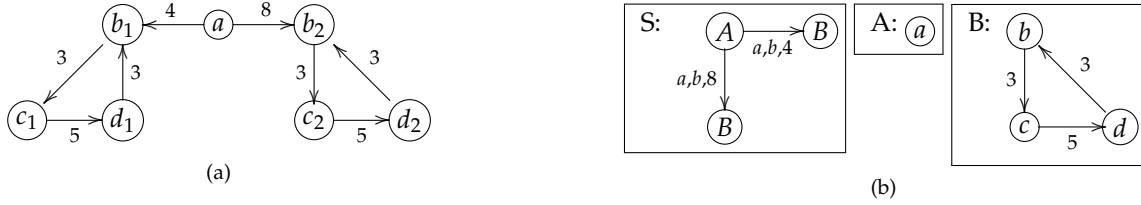


Figure 2.13: Example graph (a) and its hierarchical compression (b). Capital-lettered vertices represent super nodes, lowercase-lettered vertices represent terminal nodes. The graph labeled S is the highest in the hierarchy.

graph.

We begin by doing a topological sort of the graph highest in the hierarchy. For every (super)vertex v of this graph's topological sort we then perform a topological sort on the subgraph it represents. We continue this substitution until we reach only terminals. After we perform a topological sort on each component, the list of sorts behaves as a dictionary compression. Every time a supervertex v appears in a topological sort of a component, we substitute v for the topological sort of the component which it represents. The time and space complexity for this algorithm with this dictionary compression is $O(C)$, and without to be $O(C + |V|)$.

2.7.2 Single Source Shortest Path

The single source shortest paths problem requires, given a graph and a vertex *start*, compute the shortest path from *start* to all other vertices. Our first step is to use an all-pairs shortest path algorithm, like Floyd-Warshall [46], to compute every path (y, x) such that:

- vertex y (we call this an entry vertex) is a vertex within the component which is the destination of an edge whose source (we call this the *input* vertex) is outside of the component. I.e. there is some edge $(input, y)$
- vertex x (we call this an exit vertex) is a vertex which is the source of an edge whose destination is outside of the component (call this the *output* vertex). I.e. there is some edge $(x, output)$.

We then remove all vertices from each component except entry and exit vertices. Now each entry-exit shortest path is represented by a single edge with weight equal that of the shortest path. This repeats on all components, except for the hierarchically topmost component.

Next we use a single source shortest path algorithm, such as Dijkstra's, to find the shortest path lengths from *start* to each of the exit vertices in its component. We then correspondingly substitute this graph for that above it in the hierarchy by substituting the component with *start* and the exit

vertices in *start*'s component. The edges from this component are all those from an exit vertex in *start*'s component to an output vertex in other components. This procedure is repeated, expanding from *start*, up to the top of the hierarchy. The final step is compute the single source shortest path by determining the path length from the start vertex to each of the entry vertices. To determine the distance from *start* to a vertex central in a component (a vertex that is neither an entry vertex nor an exit) we look up the length of the shortest path from the start vertex to an entry vertex of this component, then from the entry vertex to the queried vertex.

The total time complexity of this algorithm is $O(\sum_{\text{component } c} |V_c|^3 + |V|)$. The algorithm requires space $O(\sum_{\text{component } c} |V_c|^2 + |V|)$. The uncompressed approach requires time $O(|E| + |V| \log |V|)$. The compression-aware algorithm performs better when all components are small, and certainly runs faster than the uncompressed version when $\sum_{\text{component } c} |V_c|^3 \leq |E|$. The uncompressed approach will require space $O(|V| + |E|)$ as well.

2.7.3 Minimum Spanning Tree

Given a graph $G = (V, E)$ the minimum spanning tree (MST) of G is a subset of edges $E_{MST} \subseteq E$ such that the graph $G_{MST} = (V, E_{MST})$ is a tree connecting all of V , and the sum of all edge weights $\sum_{e \in E_{MST}} w(e)$ is minimal. We compute the MST on the compressed graph by first computing the MSTs for each component and then combining components by using a merge procedure. First a topological sort will be computed to show the dependency among components. We then compute the MST in topological order with later components added by merging the trees of smaller components. The MST of the largest component is then returned by the algorithm.

In order to find the MST of a graph compressed by its components we will first discuss MST merging: given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and their minimum spanning trees E_{MST1} and E_{MST2} respectively, find the MST of the graph $G^* = (V^*, E^*)$ where $V^* = V_1 \cup V_2$ and $E^* = E_1 \cup E_2 \cup E'$ with $E' \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$. At a high level, this problem asks that given two disjoint graphs G_1 and G_2 , their MSTs, and a set of edges E' which cross these graphs, produce the MST of the new graph G^* which results from combining G_1 , G_2 , and E' .

To solve this we exploit the cut property of minimum spanning trees, this states that the smallest edge which spans any partition of vertices must be included in the minimum spanning tree. More formally, for any partition $V_c \subset V$ of vertices in graph $G = (V, E)$, the minimum edge (v_1, v_2) such that $v_1 \in V_c \Leftrightarrow v_2 \notin V_c$ must be included in G 's minimum spanning tree.

By this property the smallest-weight edge $e_{min} \in E'$ must be in the minimum spanning tree of G^* . Taking $E_{MST1} \cup E_{MST2} \cup \{e_{min}\}$ is sufficient to produce a spanning tree of G^* , it is not sufficient

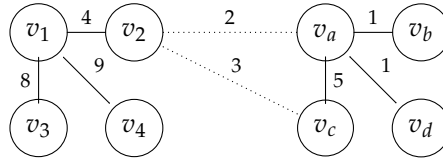


Figure 2.14: This figure shows that to merge two minimum spanning trees, simply applying the cut property between the two subsets of vertices is not sufficient

for a minimum spanning tree. Consider the example demonstrated in Figure 2.14. In this example $G_1 = \{v_1, v_2, v_3, v_4\}$ and $G_2 = \{v_a, v_b, v_c, v_d\}$. The minimum spanning trees for each are shown in solid edges, and the dotted edges give E' . Simply adding the edge (V_2, V_a) to the graph as dictated by the cut property does not give the minimum spanning tree. Instead, the minimum spanning tree contains edge (V_2, V_c) rather than (V_a, V_c) .

To merge the two MSTs we remove, from both, all of the edges which share a vertex with any edge in E' . Call these new graphs G'_1 and G'_2 and the removed edges E'' . We can now view the state of the graph as a partial solution of Kruskal's MST Algorithm where $F = \{G'_1, G'_2\} \cup \bigcup_{(v_a, v_b) \in E'} \{(\{v_a\}, \emptyset), (\{v_b\}, \emptyset)\}$ denotes the forest in this partial solution and $E' \cup E''$ denotes the set of edges. Kruskal's Algorithm can then "resume" from this partial solution to produce the new MST.

The run time of MST Merging will be dominated by the use of Kruskal's Algorithm, which runs in $O(E \log V)$ time (the run time will be $O(E\alpha(V))$ if a disjoint set data structure is used). The cost of Kruskal's can alternatively be written as $O(E \log T)$ (or $O(E\alpha(T))$) where T is the number of trees in the forest, and E is the number of edges which cross trees. This implies that the run time of the partial solution will be $O(|E'| \log |E'|)$ since there are at most $2 \cdot |E'|$ edges crossing the trees G_1 and G_2 , at most $|E'|$ edges removed from G_1 and G_2 to construct G'_1 and G'_2 , and at most $|E'| + 2$ trees in the forest.

2.8 Boldi and Vigna: WebGraph Compression

Boldi and Vigna in [41] presented a compression technique for web graphs as part of their WebGraph Framework. This scheme first requires that all vertices be given a unique index. The graph is represented in its adjacency list form, with the vertices ordered according to their index. The adjacency list is compressed by representing destination vertices in three ways: copy blocks, intervals, and residuals.

Each element in the adjacency list has a reference element. The copy blocks field gives an

encoding of which vertices in the current element's adjacency list are shared with those in the referenced element's. The intervals field gives a list of sequential blocks such that all vertices in the block are in the adjacency list. For example it may state that "all vertices 1-5 and 13-24 are in this element's adjacency list". The residuals are the nodes which cannot be expressed in any of the other two ways. For simplicity we consider residuals as intervals of length 1.

2.8.1 Bipartite Assignment

The advantage of the BV compression comes from the representation of the copy blocks and the sequences, thus any algorithm which effectively utilizes the compression must take advantage of this representation. While performing bipartite checking we know that at a given vertex the color of the back reference must match that of the current node, as otherwise there would be conflict with the destination nodes they share. We exploit this property in order to skip the copy block nodes. At a given node we mark the back reference as the same color, but indicate it is still unvisited to be processed later.

We use a disjoint-set forest data structure like that presented in chapter 21 of [46]. As a sequence is visited we create a set for each element in that sequence then union all together. If there is another set with nonempty intersection with this sequence, we union both sets together. We maintain four instances of this data structure: two for red vertices and two for blue. For each of these we maintain identical copies, with the maximum vertex as the set representative in one instance and the minimum vertex in the other. We call these forests *max_red*, *min_red*, *max_blue*, and *min_blue*. By stating *red* or *blue* we imply that the same operation be done on both *max_red* and *min_red* or *max_blue* and *min_blue*.

Algorithm 11 (BV_Bipartite) for bipartite checking a BV-compressed graph operates as follows. We begin with vertex 1 and color it RED. Note that vertex 1 does not have a back pointer in this form of compression. We then add vertex 1 to a list of visited nodes (called *visited*) and *red*. Next we add each of the sequences in vertex 1's adjacency list to *blue*. We also add these vertices to a buffer.

Next we remove an element from the buffer and add that element to *visited*. We then check the color of this element by doing a FIND operation in *red* and *blue*. After this we check the color of the back reference vertex by doing a find operation in *red* and *blue*. If the color of the back reference does not match that of the current vertex then return false. Otherwise if the back reference vertex is uncolored we add it to either *red* or *blue*, whichever the current vertex is in (assume that the current node is blue). We then process each sequence in the vertex's adjacency list. We traverse

parent vertices in both the *max_blue* and *min_blue* forests to see if any subset has a representative within the sequence. If so then we return false. Otherwise we operate similarly in *red*, and for any sets not disjoint with the sequence we perform a union of all these sets with any additional vertices from the sequence which must be added. We then add all yet unvisited vertices in each sequence to the buffer. We repeat until the buffer is empty.

In total, this algorithm requires time $O((|V| + s) \cdot \alpha(|V|))$ (theorem 2.A.21), and space $O(C)$ (theorem 2.A.22). The uncompressed approach requires $O(|V| + |E|)$ time and space. Since $s \ll |E|$ and for any V of practical size we can consider $\alpha(|V|)$ to be a small constant, this algorithm is much faster than the uncompressed approach, and requires less space.

2.9 Summary

The primary contribution of this chapter is its presentation of various sorting algorithms, graph algorithms, and geometric algorithms which operate on compressed data. Not only does this save the user time from decompressing and then recompressing data, but operating on the compressed data may give a performance benefit over even the traditional raw-data analog. In fact, the algorithms presented guarantee a time complexity improvement over the classical approach whenever $C \ll n$ for numerical sorting, or in most occasions where $C \ll |V| + |E|$ for graph algorithms.

Algorithm 1 (priority queue sorting on data compressed by arithmetic sequences) runs in $O(n \log C)$ time, and are therefore no worse than a decompression followed by a sort, even for inputs with a poor compression ratio. Algorithm 3 (sorting on data compressed by LZ77), by running in $O(C + |\Sigma| \log |\Sigma| + n)$, effectively gives linear sorting time in n . Finally, Algorithm 5 (sorting on data compressed by a context free grammar), by running in $O(C \cdot |\Sigma|)$, gives a sorting time independent of n .

We also present other noteworthy algorithms for indexing and finding k^{th} order statistics. Algorithm 2 provides $O(C \log C)$ running time for computing the k^{th} order statistic of a set of data compressed by arithmetic sequences. Algorithm 4 provides a method for indexing into data compressed by LZ77 in $O(C)$ time. If this indexing is done into a sorted LZ77 compressed data then this gives the k^{th} order statistic. One could use previous work done on random access on sorted grammar-compressed data, such as the method presented in [47], in order to find the k^{th} order statistic for grammar-compressed data.

Next we give a compression scheme for pointset data which is defined for arbitrary dimensionality, and can be used in either a lossy or a lossless way. We give algorithms for nearest neighbor,

range searches, and polytope membership, all of which run faster while requiring less memory than their traditional counterparts. Nearest neighbor and range queries each require $O(L \cdot d)$ time, where L is the number of lines in the compressed pointset. Polytope membership runs in time $O(L \cdot d \cdot F)$, where F is the number of faces on the convex polytope.

Finally we provide fast algorithms on compressed graphs under a Re-Pair compression, hierarchical compression, and BV compression, which each running asymptotically faster than the equivalent algorithms on uncompressed data. We present algorithms for topological sort and bipartite checking on Re-Pair compressed graphs which run in time $O(C)$ rather than $O(|V| + |E|)$ by the uncompressed graph size. Under a hierarchical compression model we can perform topological sort in $O(C + |V|)$ and single source shortest path in $O(\sum_{\text{component } c} |V_c|^3 + |V|)$, compared to $O(|V| + |E|)$ and $O(|E| + |V| \log |V|)$ for the uncompressed approaches respectively. In the BV model we perform bipartite checking in $O(C \cdot (|V|))$, which for all practical purposes is linear in C .

Most importantly this work introduces a novel extension on the aforementioned results on compressed pattern matching and edit-distance: that general-purpose algorithms can be performed on compressed data. We broaden the scope of these results in order to show wide-spread application of these techniques across many problem domains. This demonstrates the necessity of developing new theory and techniques for compression-aware algorithms to unlock this potential.

2.A Lemmas and Theorems

In order to enhance the readability of this dissertation chapter, we collected the proofs of the various lemmas and theorems into a single section below.

Theorem 2.A.1. *When Algorithm 1 (Arith_Sort) is run on an arithmetic-sequences compressed array of compression size C and uncompressed size n , it terminates in time $O(n \log C)$.*

Proof. The algorithm en-queues and de-queues n total points be inserted into and removed from the queue. The invariant requires that exactly one element be in the priority queue at any give time. The time to insert/delete one item into a priority queue is $O(\log m)$ where m is the size of the queue. Due to the invariant $m = C$, the total time taken to perform the n insertions and deletions gives a time complexity of $O(n \log C)$. □

Theorem 2.A.2. *When Algorithm 1 (Arith_Sort) is run on an arithmetic-sequences compressed array of compression size C and uncompressed size n , it requires $O(n)$ space.*

Proof. The only relevant variables are the compression (size C), the output data structure (size n), and the priority queue (which due to the invariant must be size C). It is assumed that in practice $C \ll n$, giving a final space complexity of $O(n)$. \square

Lemma 2.A.1. *The number of points in $[0, g]$, given by $\sum_{i=1}^C \lfloor \frac{g}{\delta_i} \rfloor$ and $g = \frac{k}{\Lambda}$, is between k and $k - C$*

Proof. By construction $\sum_{i=1}^C \frac{g}{\delta_i} = k$. It is clear that for any sum of positive rationals

$$\sum_{i=1}^j \frac{a_i}{b_i} - \sum_{i=1}^j \lfloor \frac{a_i}{b_i} \rfloor \leq j$$

since for any rational

$$0 \leq \frac{a}{b} - \lfloor \frac{a}{b} \rfloor \leq 1$$

This implies that

$$\sum_{i=1}^C \frac{g}{\delta_i} - \sum_{i=1}^C \lfloor \frac{g}{\delta_i} \rfloor = k - \sum_{i=1}^C \lfloor \frac{g}{\delta_i} \rfloor \leq C \Rightarrow k - C \leq \sum_{i=1}^C \lfloor \frac{g}{\delta_i} \rfloor$$

We also know that for any rational

$$\lfloor \frac{a}{b} \rfloor \leq \frac{a}{b} \Rightarrow \sum_{i=1}^C \lfloor \frac{g}{\delta_i} \rfloor \leq \sum_{i=1}^C \frac{g}{\delta_i} = k$$

Thus

$$k - C \leq \sum_{i=1}^C \lfloor \frac{g}{\delta_i} \rfloor \leq k$$

\square

Theorem 2.A.3. *When Algorithm 2 (Index.Arith) is run on an arithmetic-sequences compressed array of compression size C , it terminates in time $O(C \log C)$.*

Proof. We break up the time complexity of this analysis by the *guess*, *check*, *count* processes.

guess:

For this step we must calculate

$$\Lambda = \sum_{i=1}^C \lambda_i = \sum_{i=1}^C \frac{1}{\delta_i}$$

This calculation requires C additions, thus giving time complexity $O(C)$.

Next we find the value of $g = \frac{k}{\Lambda}$, which can be done in constant time.

check:

For this step we check the number of points in $[0, g]$, given by

$$\sum_{i=1}^C \lfloor \frac{g}{\delta_i} \rfloor$$

This calculation requires C additions, thus giving time complexity $O(C)$.

count:

For this step we must count the remaining points in the sequence, the number of these is given by

$$\sum_{i=1}^C \frac{g}{\delta_i} - \sum_{i=1}^C \lfloor \frac{g}{\delta_i} \rfloor = m$$

By Lemma 2.A.1 this value is no greater than C .

We then find the smallest element in the sequence after g , given by (for sequence $A(\delta_i)$)

$$\delta_i + g - (g \bmod \delta_i)$$

This can be calculated in constant time.

These smallest elements in each sequence are then put into a priority queue, and then the remaining $m \leq C$ points are found as in `Arith.sort`. This gives a final running time of this step to be $O(C \log C)$, which is of higher complexity than the *guess* and *check* steps.

□

Theorem 2.A.4. When Algorithm 2 (`Index.Arith`) is run on an arithmetic-sequences compressed array of compression size C , it requires $O(C)$ space.

Proof. In this algorithm there is only one variable-sized data structure used: the priority queue. Similar to `Arith.sort` this priority queue has a data structure that restricts its size complexity to be no more than C . Therefore the total space used by this algorithm is $O(C)$. □

Theorem 2.A.5. When Algorithm 3 (`LZ77.Sort`) is run a LZ77 compressed array of compression size C , uncompressed size n , and a set of values Σ , it terminates in time $O(C + |\Sigma| \log |\Sigma| + n)$.

Proof. The action of this algorithm can be broken into two components: *scan* and *simulated decompression*.

scan:

This step does a search through the compression to build up Σ , then sorts the characters in Σ . To scan through the compression requires $O(C)$ time. Whenever a terminal is encountered it can

then be added to a priority queue, if repeated the terminal can be removed. The cost to maintain the priority queue is $O(|\Sigma| \log |\Sigma|)$. This gives a final time complexity of $O(C + |\Sigma| \log |\Sigma|)$.

simulated decompression:

This step acts similarly to a decompression of the LZ77 compressed array, which runs in time $O(n)$. The only difference between this step and a true decompression is that this step must keep count of the multiplicity of each element in Σ , and it uses a circular buffer rather than an array. Since we have a static sorted list of elements in Σ we can iterate the counter for any character in constant time. A circular buffer, in this case, can be implemented as an array where every query to index i of the array is actually an index to $i \bmod \text{size}$, where size is the capacity of the circular buffer, which still allows for constant access time into the array. Therefore the simulated decompression takes no more time than actual decompression, so this step takes time $O(n)$.

This gives a final time complexity of $O(C + |\Sigma| \log |\Sigma| + n)$

□

Theorem 2.A.6. *When Algorithm 3 (LZ77_Sort) is run on a LZ77 compressed array of compression size C with maximum back pointer length size , it requires $O(C + \text{size})$ space.*

Proof. There are three data structures at use in this algorithm: the original compression, Σ , and the circular buffer. The size of the compression is defined to be C , and we know that $C \leq |\Sigma|$. The size of the circular buffer has been defined as the depth of the deepest back pointer in the compression, which is called size . This implies that our final time complexity is $O(C + \text{size})$. □

Theorem 2.A.7. *When Algorithm 4 (LZ77_Index) is run on a LZ77 compressed array of compression size C it terminates in time $O(C)$*

Proof. This algorithm requires two scans through the compression: One in the forward direction, one in the backward. The forward scan iterates through each element in the compression to find which one contains the query index i . In this scan each element in the compression is viewed no more than once, giving time complexity $O(C)$. If this element is not a terminal the value of i is updated by a constant time calculation to a new $i' < i$.

Next the backward scan begins, searching for the element which contains i' , if this again is not a terminal we perform a constant time update of i' and continue scanning backwards. In the backwards scan we never index any element in the compression more than once, thus giving running time $O(C)$. □

Theorem 2.A.8. *When Algorithm 4 (LZ77_Index) is run on a LZ77 compressed array of compression size C it requires $O(C)$ space.*

Proof. The only data required for this algorithm is the original compression, the index i , and some additional counters. Therefore the asymptotic space complexity is given by $O(C)$. \square

Theorem 2.A.9. *When Algorithm 5 (CFG_Sort) is run on a context free grammar compression of size C with a set of values Σ , it terminates in time $O(C \cdot |\Sigma|)$*

Proof. The first step of this algorithm behaves much the same as the first step of LZ77_Sort, there is a scan through the compression which sorts the characters. This has time complexity $O(C + |\Sigma| \log |\Sigma|)$.

The dependency graph construction step takes time $O(C)$, as by simply removing terminals from the substitution rules the grammar resembles an adjacency list. This resulting graph has a vertex for every variable in the grammar, and an edge for every term in the compression. This means that a topological sort of the graph can be completed in time $O(C)$.

The final step of summing the vectors for each term in total takes $O(C)$ vector additions. Each addition takes $O(|\Sigma|)$ time, giving a time complexity of $O(C \cdot |\Sigma|)$ for this step.

Since $C > |\Sigma| > \log |\Sigma|$ the dominating term is $O(C \cdot |\Sigma|)$. \square

Theorem 2.A.10. *When Algorithm 5 (CFG_Sort) is run on a context free grammar compression of size C with a set of values Σ , it requires $O(C \cdot |\Sigma|)$ space.*

Proof. The data structures used in this algorithm are the original compression, the dependency graph representation of this compression (with its topological sorted list of vertices), and the list of vectors used during the final summation. The original compression has size C , and its graph representation also has size C since by removing terminals from the compression we create the graph's adjacency list representation. The list of vectors has size $O((|V| + |\Sigma|) \cdot |\Sigma|)$, where V is the set of variables in the grammar, since each term in the grammar requires a vector of size $|\Sigma|$. Since $|V| + |\Sigma| \leq C$ we simplify this by giving space complexity $O(C \cdot |\Sigma|)$. \square

Theorem 2.A.11. *When Algorithm 6 (LZ78_Sort) is run on a LZ78 compression of size C with a set of values Σ , it terminates in time $O(C \cdot |\Sigma|)$.*

Proof. The proof for this statement is similar to that for CFG_Sort. Additionally, a LZ78-compressed array can be converted to a CFG-compressed array in time $O(C)$. Thus CFG_Sort can be run with no asymptotic increase in time complexity. \square

Theorem 2.A.12. *When Algorithm 6 (LZ78.Sort) is run on a LZ78 compression of size C with a set of values Σ , it requires $O(C \cdot |\Sigma|)$ space.*

Proof. The proof for this statement is similar to that for CFG.Sort. Additionally, a LZ78-compressed array can be converted to a CFG-compressed array with space $O(C)$. Thus Sort_CFG can be run with no asymptotic increase in space complexity. \square

Theorem 2.A.13. *When Algorithm 7 (RePair_Topological) is run on a Re-Pair compressed graph of size C , it terminates in time $O(C)$.*

Proof. The Re-Pair compression is itself a graph of size $|V_C| + |E_C| = C$ by definition, where V_C is the set of vertices in this graph, and E_C is the set of edges. A topological sort on this graph requires time $O(|V_C| + |E_C|) = O(C)$.

Once the topological sort is done the dictionary vertices are removed, taking time $O(|V_C|)$. This gives a final time complexity of $O(C)$. \square

Theorem 2.A.14. *When Algorithm 7 (RePair_Topological) is run on a Re-Pair compressed graph of size C , it requires $O(C)$ space.*

Proof. The only data structures involved with this algorithm are the original compression, and its topological sort. The compression is defined to have size C , and the topological sort has size $|V_C| \leq C$. Thus the asymptotic space complexity of the algorithm is given by $O(C)$. \square

Theorem 2.A.15. *When Algorithm 8 (RePair_Bipartite) is run on a Re-Pair compressed graph of size C , it terminates in time $O(C)$.*

Proof. This algorithm is a modification of a breadth first search. The modification made is simply the procedure for color checking done while processing each node, which for each node is a constant time overhead. This means that there is no asymptotic time penalty for running this algorithm over breadth first search, giving the time complexity to be $O(|V_C| + |E_C|) = O(C)$. \square

Theorem 2.A.16. *When Algorithm 8 (RePair_Bipartite) is run on a Re-Pair compressed graph of size C , it terminates in time $O(C)$.*

Proof. The only data structure needed for this algorithm is the original compression plus auxiliary data to keep track of vertex color. This auxiliary information comes at no additional asymptotic space cost since it requires only constant size additional information for each vertex, therefore the space complexity is $O(C)$. \square

Theorem 2.A.17. *When Algorithm 9 (Hierarchical.Topological) is run on a hierarchy compressed graph of compression size C , and V vertices uncompressed, it terminates in time $O(C + |V|)$.*

Proof. The first step in this algorithm is to perform a topological sort of each component. For each component with $|V|$ vertices and $|E|$ edges this takes $O(|V| + |E|)$. Therefore the total time complexity of this step is (where k is the number of components in the compression)

$$\sum_{i=0}^{k-1} O(|V_i| + |E_i|) = O(C)$$

Left as-is the output of these topological sorts is a dictionary-compressed topological sort of the original uncompressed graph. However, this can be expanded into the uncompressed topological sort in $O(|V|)$ time, where V is the set of vertices in the uncompressed graph. Thus the total time complexity is $O(|V| + C)$. \square

Theorem 2.A.18. *When Algorithm 9 (Hierarchical.Topological) is run on a hierarchy compressed graph of compression size C , and V vertices uncompressed, it requires $O(C + |V|)$ space.*

Proof. The data structures used in this algorithm are the compression (which is defined to have size C), and the topological sorts. Similar to the time complexity, if the topological sorts of each component are left in a dictionary compressed form we achieve space

$$O\left(\sum_{i=0}^{k-1} O(|V_i|)\right) \leq O(C)$$

for the topological sorts. If these are decompressed we use space $O(|V|)$. This gives an overall complexity of $O(C + |V|)$. \square

Theorem 2.A.19. *When Algorithm 10 (Hierarchical.SSSP) is run on a hierarchy compressed graph where a component c has vertices V_c and the uncompressed graph has vertices V , it terminates in time $O(\sum_{\text{component } c} |V_c|^3 + |V|)$.*

Proof. The first step of this algorithm is to perform Floyd-Warshall on each component, this gives overall time complexity

$$O\left(\sum_{\text{component } c} |V_c|^3\right)$$

The time to run Dijkstra's algorithm on the component with $start$ is

$$O(|E_{start}| + |V_{start}| \log |V_{start}|) < O(|V_{start}|^3)$$

The final step requires Dijkstra's algorithm be run on the graph atop the hierarchy (call this G_0), which takes time

$$O(|E_0| + |V_0| \log |V_0|) < O(|V_0|^3)$$

The output path table takes time $O(|V|)$ to compute because the shortest path from *start* to any other vertex v can be computed in constant time once the distance from *start* to the entry vertex in v 's component is known. Therefore the final running time is $O(\sum_{\text{component } c} |V_c|^3 + |V|)$. \square

Theorem 2.A.20. *When Algorithm 10 (Hierarchical_SSSP) is run on a hierarchy compressed graph where a component c has vertices V_c and the uncompressed graph has vertices V , it requires $O(\sum_{\text{component } c} |V_c|^2 + |V|)$ space.*

Proof. The space required for all of the all-pairs shortest path tables is

$$O\left(\sum_{\text{component } c} |V_c|^2\right)$$

The space required to maintain the single-source shortest path table is $O(|V|)$. These combined give the overall asymptotic space complexity to be

$$O\left(\sum_{\text{component } c} |V_c|^2 + |V|\right)$$

\square

Theorem 2.A.21. *When Algorithm 11 (BV.Bipartite) is run on a Boldi-Vigna compressed graph with vertices V and s sequences, it terminates in time $O((|V| + s) \cdot \alpha(|V|))$ where $\alpha(|V|)$ is the inverse Ackermann function.*

Proof. We assume constant time insertion and deletion from *visited* and the buffer. We never do more than $|V|$ inserts and deletes from each one, so the time used to maintain these structures across the whole algorithm is $O(|V|)$.

In [46] the authors state that a sequence of m MAKE – SET, FIND, and UNION operations where n of them are MAKE – SET takes time $O(m \cdot \alpha(n))$ where α is the very slow growing inverse Ackermann function. This gives us the running time for maintaining the forests to be $O((|V| + s) \cdot \alpha(|V|)) \leq O(C \cdot \alpha(|V|))$, which is also the dominant term and thus final running time. \square

Theorem 2.A.22. *When Algorithm 11 (BV.Bipartite) is run on a Boldi-Vigna compressed graph with vertices V and compression size C , it requires $O(C + |V|)$ space.*

Proof. The data structures used in this algorithm are the original compression, the array *visited*, the buffer, and the forests. The original compression is defined to be size C , and each of the other structures require no more than $|V|$ space. Since there are a constant number of forests we obtain time complexity $O(C + |V|)$. \square

2.B Algorithms Pseudocode

In order to enhance the readability of this dissertation chapter, we collected the pseudocode for the various algorithms into a single section below.

Algorithm 1: Arith.Sort-A method for sorting arithmetic sequences using a priority queue. Here, *pair* contains an element v which is the value of a point, and δ which is the interval for its source arithmetic sequence.

Input: set of C arithmetic sequences $\mathcal{A} = \{A_1, \dots, A_C\}$, the number of values to sort n

Output: set of n ordered smallest values

```

1 initialize priority queue pq;
2 initialize an array of size  $n$  sorted;
3 foreach  $A_i \in \mathcal{A}$  do
4    $pq.insert(pair(0, \delta_i))$ ;
5 for  $i = 0; i < n; ++i$  do
6    $pair\ p = pq.poll()$ ;
7    $sorted[i] = p.v$ ;
8    $pq.insert(pair(p.v + p.\delta, p.\delta))$ ;
9 return sorted;

```

Algorithm 2: *Arith_Index-Finds* the k^{th} element in the combined sequence. Here, *pair* contains an element v which is the value of a point, and δ which is the interval for its source arithmetic sequence. The method *next_mult*(a, b) finds the next multiple of b which is greater than a .

Input: set of C Arithmetic Sequences $\mathcal{A} = \{A_i, \dots, A_C\}$, the index queried k

Output: value of the k^{th} smallest element

```

1  $k = k - C$ ;
2 initialize priority queue  $pq$ ;
3  $\Lambda = 0$ ;
4 foreach  $i < C$  do
5    $\Lambda += \frac{1}{\delta_i}$ ;
6  $d = \frac{k}{\Lambda}$ ;
7  $count = 0$ ;
8 foreach  $i < C$  do
9    $count += \lfloor \frac{d}{\delta_i} \rfloor$ ;
10   $pq.add(pair(next\_mult(d, \delta_i), \delta_i))$ ;
11 Initialize  $value = 0$ 
12 for  $i = 0; i < k - count; ++i$  do
13    $pair\ p = pq.poll()$ ;
14    $value = p.v$ ;
15    $pq.insert(pair(p.v + p.\delta, p.\delta))$ ;
16 return  $value$ ;

```

Algorithm 3: *LZ77_Sort-A* method for sorting a LZ77 compressed string. Here *back* is the location of the back pointer index, and *length* is the number of characters to copy. It is assumed that if an index is not in Σ then it is a back pointer.

Input: A LZ77-compressed list LZ

Output: A LZ77-compressed sorted list

```

1 initialize a list  $Lit$ ;
2 initialize a table  $map$  where  $key \in \Sigma$ ,  $value = 0$ ;
3 initialize a circular buffer  $b$  with  $size = \text{length of longest back reference}$ ;
4 foreach  $\alpha \in LZ$  do
5   if  $\alpha \in \Sigma \wedge \alpha \notin Lit$  then
6      $Lit.insert(\alpha)$ ;
7  $Lit.sort()$ ;
8 initialize  $j = 0$ ;
9 for  $i = 0; i < C$  do
10  if  $LZ[i] \in \Sigma$  then
11     $b[j \bmod size] = LZ[i]$ ;
12     $++map.value(LZ[i])$ ;
13     $++j$ ;
14  else
15    for  $m = LZ[i].back; m < (LZ[i].back + LZ[i].length)$  do
16       $b[j \bmod size] = b[m \bmod size]$ ;
17       $++map.value(b[m \bmod size])$ ;
18       $++j$ ;
19 return  $map$ ;

```

Algorithm 4: LZ77_Index-A method for indexing a LZ77 compressed string. Here *back* is the location of the back pointer index, and *length* is the number of characters to copy. It is assumed that if an index is not in Σ then it is a back pointer.

Input: A LZ77-compressed list *LZ*, a query index *i*

Output: The *i*th element of a decompressed *LZ*

```

1 initialize count = 1;
2 initialize j = 1;
3 while count < i do
4   ++ j;
5   if  $LZ[j] \in \Sigma$  then
6     ++ count;
7   else
8     count += LZ[j].length;
9 while  $LZ[j] \notin \Sigma$  do
10  i = LZ[j].back + LZ[j].length - (count - i) - 1;
11  if i > count - LZ[j].length then
12    i = ((i - LZ[j].back) mod (count - LZ[j].length - LZ[j].back + 1)) + LZ[j].back;
13  while count ≥ i do
14    if  $LZ[j] \in \Sigma$  then
15      count = count - 1;
16      j = j - 1;
17    else
18      count = count - LZ[j].length;
19      j = j - 1;
20  ++ j;
21  if  $LZ[j] \notin \Sigma$  then
22    count = count + LZ[j].length;
23  else
24    ++ count;
25 return LZ[j];

```

Algorithm 5: CFG_Sort-Sorts a context free grammar. Here, Σ is the set of values in the sequence represented by the grammar, and *V* is the set of variables.

Input: Context free grammar *CFG*

Output: The sorted string represented by *CFG* with start variable *A*₀

```

1 Convert the Variables in CFG into a dependency graph G;
2 perform a topological sort on G;
3 reorder rules in CFG to be the reverse of G;
4 Sort  $\Sigma$ ;
5 transform each literal into a  $|\Sigma|$ -dimensional vector;
6 foreach rule r ∈ CFG do
7   initialize a vector sum = ⟨0⟩;
8   foreach Symbol S ∈  $\Sigma \cup V$  listed in r do
9     sum += ⟨S⟩;
10   $\Sigma = \Sigma \cup \{A + r\}$ , where Ar is the variable associated with rule r;
11  ⟨Ar⟩ = sum;
12 return ⟨A0⟩;

```

Algorithm 6: LZ78.Sort-A method for sorting a LZ78 compressed string. Here *back* is the location of the back pointer index, and σ is the symbol to append.

Input: A LZ78-compressed list *LZ*

Output: A LZ78-compressed sorted list

```

1 initialize an array map where elements are in  $\mathbb{N}^{|\Sigma|}$ ;
2 initialize a  $|\Sigma|$ -dimensional vector sum =  $\langle 0 \rangle$ ;
3 for  $i = 1; i < |LZ|; i++$  do
4     if  $LZ[i].back == 0$  then
5          $map[i] = \langle LZ[i].\sigma \rangle$ ;
6     else
7          $map[i] = map[LZ[i].back] + \langle LZ[i].\sigma \rangle$ ;
8      $sum += LS[i]$ ;
9 return sum;
```

Algorithm 7: RePair.Topological-A method for performing a topological sort on a Re-Pair compressed graph. Here *V* is the set of vertices from the original uncompressed graph *G*.

Input: A Re-Pair compressed graph G_C

Output: A list of vertices from the original graph *G* in topologically-sorted order.

```

1 initialize a list TS;
2  $TS_C = \text{topological\_sort}(G_C)$ ;
3 foreach  $v \in TS$  do
4     if  $v \in V$  then
5          $TS.append(v)$ ;
6 return TS;
```

Algorithm 8: RePair_Bipartite-A method for bipartite assignment on a Re-Pair compressed graph. Here V is the set of vertices from the original uncompressed graph G , and V_C is the set of vertices from the compressed graph G_C . Each vertex object is associated with a color called *color*, a predecessor called π , and an adjacency list *adj*. the color WHITE identifies unvisited vertices, GRAY is discovered but not processed, and RED and BLUE are used as the two colors for the bipartite testing and signify that the vertex has been completely processed.

Input: A Re-Pair compressed graph G_C

Output: V if the uncompressed graph G is bipartite, FALSE otherwise.

```

1 initialize FIFO queue  $Q$ ;
2 set  $D = V_C \setminus V$ ;
3 foreach  $v \in V_C \setminus \{s\}$  do
4    $v.color = WHITE$ ;
5    $v.\pi = NULL$ ;
6  $s.color = GRAY$ ;
7  $s.\pi = NULL$ ;
8 ENQUEUE( $Q, s$ );
9 while  $Q \neq \emptyset$  do
10    $u = DEQUEUE(Q)$ ;
11   foreach  $v \in u.adj$  do
12     if  $v.color == WHITE$  then
13        $v.color = GRAY$ ;
14        $v.\pi = u$ ;
15       ENQUEUE( $Q, v$ );
16     else if  $v.color == u.color \wedge u \notin D \vee v.color \neq u.color \wedge u \in D$  then
17       return FALSE;
18   if  $((u.\pi).color == RED \wedge u \notin D) \vee ((u.\pi).color == BLUE \wedge u \in D)$  then
19      $u.color = BLUE$ ;
20   else
21      $u.color = RED$ ;
22 return  $V$ ;

```

Algorithm 9: Hierarchical_Topological-A method for performing a topological sort on a hierarchy compressed graph.

Input: A list of components L for a hierarchical-compressed graph.

Output: A list of vertices from the original graph G in topologically-sorted order.

```

1 initialize a 2-dimensional list  $TS_C$ ;
2 initialize a list  $TS = \emptyset$ ;
3 foreach  $c \in L$  do
4    $TS_C[c] = \text{topological\_sort}(c)$ ;
5 while  $TS_C[0] \neq \emptyset$  do
6    $v = TS_C[0].remove\_head()$ ;
7   if  $TS_C[0][v] \in L$  then
8     INSERT( $TS[0][v], TS[v]$ );
9   else
10     $TS.append(v)$ ;
11 return  $TS$ ;

```

Algorithm 10: Hierarchical_SSSP-A method for performing single-source shortest path on a hierarchy compressed graph. We assume that in L the components are listed from the bottom of the hierarchy to the top.

Input: A list of components L for a hierarchical-compressed graph G . A start vertex $start$.

Output: A table containing the shortest paths from $start$.

```

1 initialize a list  $all\_pairs = \emptyset$ ;
2 define  $G_{start}$  as the component containing  $start$ ;
3 define  $G_0$  as the top component in the hierarchy;
4 define  $paths$  to be the shortest distances from  $start$  to each  $v$ ;
5 foreach component  $c \in L$  do
6   foreach vertex  $v \in V_c$  do
7     if  $v \notin V$  then
8        $\lfloor$  replace  $v$  with all  $(entry, exit)$  paths in its representative component;
9    $\lfloor all\_pairs[c] = \text{Floyd} - \text{Warshall}(c)$ ;
10 Find shortest paths of type  $(start, entry)$ ;
11 foreach vertex  $v \in V$  do
12    $\lfloor paths(v) = \min((start, entry) + (entry, v))$ ;
13 return  $v$ ;
```

Algorithm 11: BV.Bipartite-A method for performing a bipartite assignment on a BV compressed graph. Every time we do MAKE – SET into $red_min, red_max, blue_min, blue_max$ we also do a FIND for one more than and one less than that value. If either exist we then do a UNION with those sets. Where it states red and $blue$, assume the same operation is done on red_min, red_max or $blue_min, blue_max$. $adj(v)$ refers to the compressed adjacency list of v . When we do a FIND on a sequence we receive all sets incident with that sequence.

Input: A BV-compressed graph G .

Output: red_min and $blue_min$ if the uncompressed graph G is bipartite, FALSE otherwise.

```

1 initialize  $red\_min, red\_max, blue\_min, blue\_max$ ;
2 initialize  $visited = \emptyset$ ; initialize  $buffer = \emptyset$ ;
3  $red.MAKE - SET(1)$ ;
4  $visited.insert(1)$ ;
5 foreach vertex  $v \in adj(1)$  do
6    $buffer.insert(v)$ ;  $blue.MAKE - SET(1)$ ;
7 while  $buffer.notEmpty()$  do
8    $v = buffer.remove()$ ;  $visited.insert(v)$ ;
9    $color = RED$  if  $red.FIND(v)$  else  $BLUE$ 
10  if  $blue.FIND(back(v)) \wedge color == RED \vee red.FIND(back(v)) \wedge color == BLUE$  then
11    return FALSE;
12  else if  $color == RED$  then
13     $red.MAKE - SET(back(v))$ ;
14  else
15     $blue.MAKE - SET(back(v))$ 
16  foreach sequence  $s \in adj(v)$  do
17    if  $color == RED$  then
18      if  $red.FIND(s)$  then
19        return FALSE;
20      foreach vertex  $u \notin blue.FIND(s)$  do
21         $blue.MAKE - SET(u)$ ;  $buffer.insert(u)$ ;
22    else if  $color == BLUE$  then
23      if  $blue.FIND(s)$  then
24        return FALSE;
25      foreach vertex  $u \notin red.FIND(s)$  do
26         $red.MAKE - SET(u)$ ;  $buffer.insert(u)$ ;
27 return ( $red\_min, blue\_min$ );

```

Chapter 3

Overview and Complexity-Theoretic Analysis of Automata Processing

The ubiquity of web connectivity has caused runaway increases in the size and number of massive datasets as well as unprecedented volumes of web traffic. These problems are exacerbated by Moore’s Law’s inability to keep pace with this trend. In response, the architecture community is looking beyond the traditional von Neumann CPU designs, towards adopting heterogeneous computing models, using a variety architectures as accelerators for performance-critical tasks. SIMD processors, such as graphics processing units (GPUs), can perform vector-parallel floating point arithmetic to yield sizable speedups on appropriately parallelizable tasks. Many-core CPU designs, such as Xeon Phi, utilize multiple parallel tandem CPUs to achieve performance benefits in a MIMD manner. Field programmable gates arrays (FPGAs) and application-specific integrated circuits (ASICs) enable highly specialized hardware designs to optimize domain-specific tasks.

We are currently seeing an increasing interest in automata-based architecture designs as a new addition to the pantheon of specialized co-processors. The promise of these architectures lies in their ability to efficiently simulate in hardware non-deterministic computations (in the form of non-deterministic finite automata), thus providing acceleration in a MISD way.

Investigators have developed fairly robust theory to support the value of many other computing tools. Some problems were long known to be more efficiently solvable by parallel computers [48], which provides a guide for identifying problems well-suited for implementation on e.g., many-core CPUs. Previous research on the complexity of Boolean circuits [49] can guide the development of circuit-based hardware such as ASICs and FPGAs. In this section we discuss the nature of

problems that are amenable to efficient parallelization via automata computing, and also relate these observations to other well-established results.

3.1 Finite State Automata

A finite state automaton (FA) is a simple classical computation model consisting of transitions among a fixed set of states [50]. More formally, a finite state automaton is defined by a 5-tuple $(Q, q_0, \Sigma, \delta, F)$ where Q is a finite set of states, q_0 is a unique initial state, Σ is an alphabet of symbols, $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function for the machine (δ maps an input state-symbol pair to a destination state), and F is a set of final accepting states. A finite automaton begins at its initial state q_0 and reads the input string one symbol at a time, while changing states for each processed symbol according to its transition function. If, after reading the entire input string, the FA halts in a final state (one belonging to F), we say that the machine accepts the input string, otherwise it rejects the input string.

Note that in the interest of succinctness we can generalize transition function δ to map sets of symbols onto new states, rather than just individual symbols (i.e. $\delta : Q \times 2^\Sigma \rightarrow Q$). This does not change the behavior of the machines, as the generalized transition function can always be converted to the former simpler definition by adding separate state-symbol pair transitions for each state-symbol-set pair transition.

A finite automaton is a *deterministic* finite automaton (DFA) if its transition mapping δ always has exactly one next state for every symbol-state input pair. If for some transitions there is more than one next state (that is, δ maps onto *sets* of states rather than individual single states), the automaton is a nondeterministic finite automaton (NFA). For an NFA, at every step a subset of its states may be active (compared to a DFA, where exactly one state may be active at any particular time), and the transition function is applied to each state in its active state-set in turn, with the resulting next-state *set* being the union of the resulting individual transitions. Formally, an NFA is defined by the same 5-tuple $(Q, q_0, \Sigma, \delta, F)$, but with a generalized transition function which maps states to sets of states, i.e. $\delta : Q \times \Sigma \rightarrow 2^Q$. If the set of states Q_a is active, and the machine receives input b , then the next set of active states is given by $\bigcup_{q \in Q_a} \delta(q, b)$. Thus, determinism is a special case of non-determinism.

While NFAs are more general than DFAs, it can be proven that they have no greater computing power. Using a technique called a *powerset construction*, any NFA can be converted to an equivalent DFA, at the potential cost of an exponential expansion in the number of states. Since there may be an arbitrary set of active states in the NFA, we build a DFA where each of its states represent a set

of states in the NFA. Every set of active states in the NFA therefore corresponds to a single state in this DFA. This implies that for some NFA with k states, there is always an equivalent DFA with no more than 2^k state. That is, eliminating non-determinism from an NFA can result in an exponential increase in the state set, and indeed this is sometimes unavoidable [50].

3.2 Micron's Automata Processor

A specific architecture that we will focus on in our discussion and experimentation is the *Micron Automata Processor* (AP) [7], which is a reconfigurable hardware accelerator for non-deterministic finite automata (NFA) simulation. This automata processor was built by the Micron Technology corporation as a hardware implementation on a circuit board that can be plugged into commodity PCs (see Figure 3.1 for a photo of this prototype). Leveraging existing VLSI memory technology, Micron's Automata Processor uses memory reads and writes along with wide logic gates to implement bit-parallel execution of NFAs in hardware.

The AP's usage of finite automata does not consider whether a particular input string is accepted/rejected, but rather gives an output signal whenever one of its pre-defined "reporting states" is active, thereafter continuing to process the input string. Such a machine is a variation of a finite automaton similar to a "Moore machine" in that every transition can also result in an outputted symbol, depending on the current active state(s) [51].

For our theoretical analysis of the AP, we wish to simplify its behavior, which will hopefully enable us to characterize the formal language classes computed (or recognized) by such automata. To do this, we must consider only the traditional automata variants, i.e. those that accept/reject input strings, rather than ones that also output symbols like in the Moore machine model.

3.2.1 Homogeneous Finite Automata

The Micron AP acts as a hardware implementation of a bit-parallel NFA simulator. The bit-parallel NFA simulation algorithm requires the automaton be homogeneous, meaning that all incoming transitions for every state must match on the same character (this could be generalized to a set of characters). Since all the incoming transitions for each state match on the same symbol, homogeneous NFA matching is a property of *states* rather than transitions. This allows for the machine topology to be defined independently from the matching behavior. The result is that each state determines its activation in isolation from the other states, thus enabling MISD parallelism.

More formally, a finite state automaton is homogeneous if for every state $q \in Q$ such that $\exists q_1, q_2 \in Q$ and $\sigma_1, \sigma_2 \in \Sigma$ where $\delta(q_1, \sigma_1) = \delta(q_2, \sigma_2) = q$ then it must be that $\sigma_1 = \sigma_2$. To generalize this definition to transitions on sets of states, the automaton is homogeneous if for every

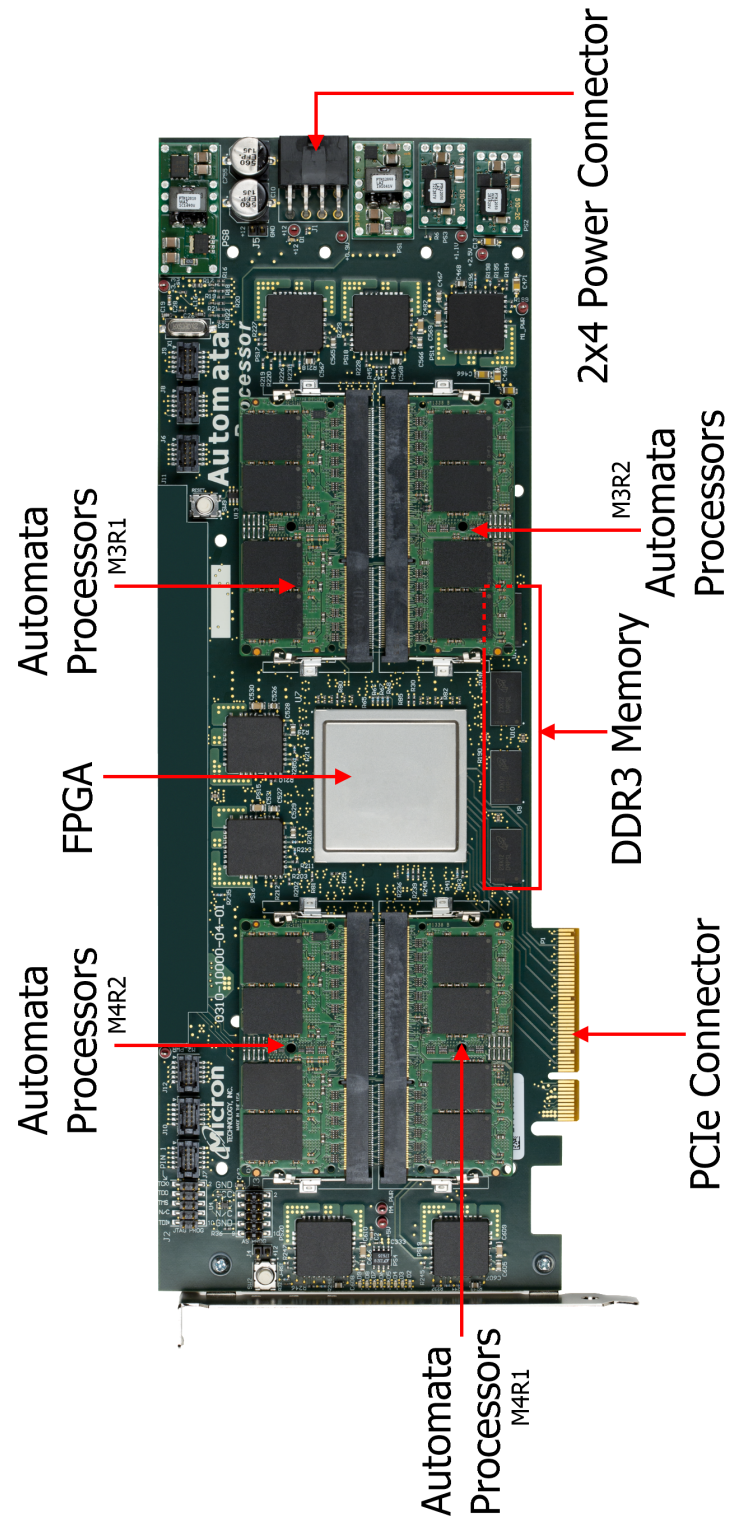


Figure 3.1: A photograph of the Micron Automata Processor (AP) hardware, manufactured by the Micron Technology corporation in 2016, which can be plugged into commodity PCs.

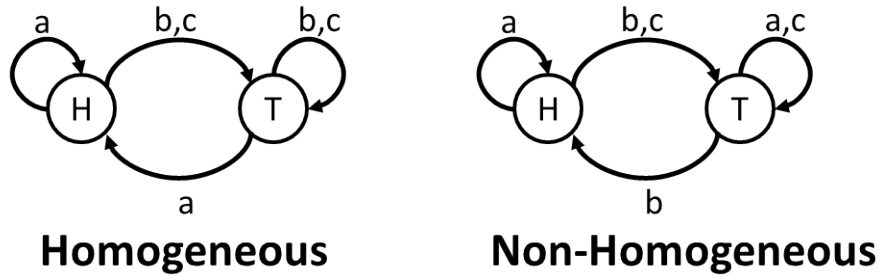


Figure 3.2: An example of homogeneous vs. non-homogeneous finite automata. In the automaton on the left all incoming transitions of all states match on the same symbol set, making it homogeneous. In the automaton on the right there is an incoming transition for state “H” which matches on the symbol ‘a’, and another that matches on the symbol ‘b’, thus this automaton is not homogeneous.

state $q \in Q$ such that $\exists q_1, q_2 \in Q$ and $\Sigma_1, \Sigma_2 \subseteq \Sigma$ where $\delta(q_1, \Sigma_1) = \delta(q_2, \Sigma_2) = q$ then it must be that $\Sigma_1 = \Sigma_2$.

Figure 3.2 illustrates the latter distinction. The bit-parallel algorithm requires this restriction as symbol matching in homogeneous automata is a property of states rather than of the transitions, i.e. *states* match on sets of symbols rather than transitions matching on sets of symbols. For a homogeneous automaton, a state q which matches on symbol σ (or set of symbols Σ') becomes active in step i if there is some state which was active in step $i - 1$ which has a transition to state q , and the i^{th} symbol in the input string is σ (or belongs to Σ').

Non-homogeneous to Homogeneous Finite Automata

Homogeneous finite-state automata are provably no less powerful than non-homogeneous automata. To establish this equivalence, we convert a given non-homogeneous automaton to an equivalent homogeneous automaton with a state expansion of $|\Sigma|$ where Σ is the machine's alphabet. To do this, consider a state q which has one incoming transition on the symbol a , and another on symbol b . To make this homogeneous we split this state into two versions, one which receives a -transitions, the other receiving b -transitions. The output transitions on both states will be the same as the outgoing transitions on the state q . This must be done at most once for each symbol and automaton state. Figure 3.3 illustrates this construction.

In the case that each state matches on a set of symbols, this construction may require $|Q| \cdot 2^{|\Sigma|}$ states in the worst case. In practice, when eliminating non-homogeneity from finite automata, the expansion of the state set seems to be modest.

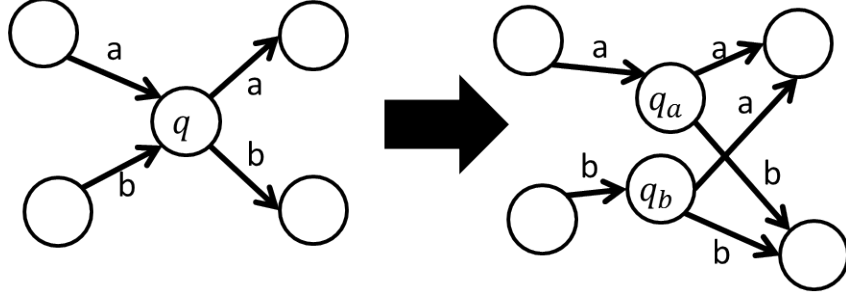


Figure 3.3: Eliminating non-homogeneity in automata by splitting states.

3.2.2 Bit-parallel Algorithm

The bit parallel technique [52] implemented by the Micron AP maintains a bit string representing the set of active states, and manipulates this string using Boolean logic over bitstrings that are stored in large tables. We begin with a bit string v_a which maintains the set of active states, where character $v_a[i] = 1$ if state i is currently active and 0 otherwise. Additionally we maintain two 2-dimensional tables of Boolean values, a routing table and a match table, whose values are derived from the machine definition. The routing table t_r , which has dimension $|Q| \times |Q|$, represents how states interconnect with each other. We say that $t_r[i][j] = 1$ if and only if there is a transition from state i to state j . The match table t_m , which has dimension $|\Sigma| \times |Q|$, represents the character sets that each state matches upon in a homogeneous way. We say $t_m[x][i] = 1$ if and only if state i matches on the symbol σ indexed by x .

To update the active set v_a on input character σ , indexed by x , the AP first computes the set of possible next states (those with an incoming transition from an active state) $v_n[j] = \bigvee_{i=0}^{|Q|} t_r[i][j] \wedge v_a[i]$. Next, the AP computes the new set of states as $v_n \wedge t_m[x]$, as this represents the set of states that both receive an incoming transition from at least one active state and also match on σ .

3.2.3 Hardware Specifications

The current generation of Micron's Automata Processor implements states (called State Transition Elements or STEs, shown in Figure 3.4) as 256-bit memory columns representing the set of 8-bit symbols upon which that state matches. When given an 8-bit symbol, an 8-to-256-bit decoder serves to access a row in this memory column (which will contain a 1 if that state matches on that symbol, and 0 otherwise). The result of this access is then passed into an AND gate with an "enable" signal, that has the value 1 if there is some adjacent state which was active in the previous cycle. In the first (prototype) generation, 256 STEs are arranged into blocks, and 96 blocks are arranged into a core, thus providing up to 24,576 states per core. Each AP chip contains 2

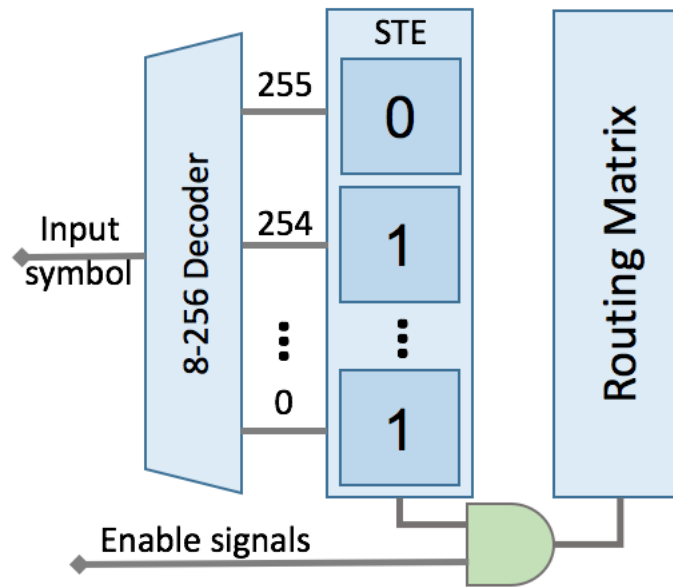


Figure 3.4: A simplified model of an STE's construction. The STE decodes each 8-bit symbol to perform a row lookup. If the lookup returns a 1, and the enable signal is active, then the STE propagates its enable signal to the routing matrix.

disjoint cores, with no transitions allowed to go between them. An AP board contains 32 chips. The hardware is designed to operate at 133MHz, consuming 1 symbol every 7.5ns, thus consuming input at a rate of 133MB/s. It is projected to operate with a TDP (thermal design power) of 4W per chip (128W per board).

Of the 256 STEs in a block, 32 are able to “report” off-board (i.e. provide output in the style of a Moore machine). To report, the hardware produces a “report vector” every cycle, which is a bit-vector representation of the active status of all reporting STEs. Each current-generation AP chip contains 6 reporting regions capable of exporting 1,024-bit output vectors within 1.8 milliseconds. Therefore a best-case upper-bound for the full AP output throughput is roughly 436.9MB/s per chip.

Each AP chip provides Boolean logic elements, which allow for a STE's enable signal to be defined by a boolean function of other states. Each gate is reconfigurable to one of 9 choices of Boolean operations: OR, AND, NOT, NOR, NAND, POS (product-of-sums), SOP (sum-of-products), NPOS (NOT POS), and NSOP (NOT SOP). The former 5 (OR through NAND) behave in the expected manner. Product-of-sums breaks up the inputs into pairs, finds the OR of each pair, then returns the AND of all results. For example, for Boolean inputs a, b, c, d, e, f $POS(a, b, c, d, e, f) = (a \text{ OR } b) \text{ AND } (c \text{ OR } d) \text{ AND } (e \text{ OR } f)$. Sum-of-products switches ANDs with ORs compared to

product-of-sums, i.e. $SOP(a, b, c, d, e, f) = (a \text{ AND } b) \text{ OR } (c \text{ AND } d) \text{ OR } (e \text{ AND } f)$. NPOS and NSOP are the negations of POS and SOP, respectively.

Finally, the Automata Processor includes counter elements. These elements have an input port and an output activation. During every cycle in which they receive an enable signal on the input port, an internal count increments. When their internal count exceeds a reconfigurable threshold, they emit an output signal, which behaves as any other automata transition.

3.3 Characterizing the Computational Power of the AP

If the Automata Processor were a simple NFA emulation engine, its computational power would be well-understood, since non-deterministic finite state automata accept exactly the class of regular languages [50]. Intuitively, this language class represents the set of functions computable by a Turing Machine using finite memory. The inclusion of the Boolean elements and counter elements in the AP architecture makes the characterization the computing capabilities of the Automata Processor less obvious.

Toward identifying the computational power of the AP, constructions for Turing-complete cellular automata were shown to be implementable on the AP [53]. This result would seem to imply that the Automata Processor is itself Turing-complete, i.e. that it can compute any function which is computable by a Turing machine, as cellular automata are Turing complete [54]. While this result certainly speaks to the possibility of the Automata Processor being able to serve purposes beyond the computation of only regular languages, the conclusion that the AP is Turing-complete is flawed.

The problem of identifying the computability classes of a particular architecture must be tackled with great care. In order to answer in a way that is faithful to practice, one must be very careful to identify and justify proper parameterization of all of the computing resources. For example, the typical computer scientist finds little issue with considering modern CPUs to be Turing-complete. In reality, however, every real-world machine has a physical memory with only a finite capacity (and the sizes of hard drives and other storage devices are similarly finitely bounded). For this reason, the class of languages computable by any real-world computer that ever existed coincides exactly with the class of regular languages, i.e. is the set of languages decidable with finite memory. We excuse this apparent inconsistency between the theory and practice of computing by considering the resources available to our computing devices.

Due to the design of von Neumann architectures, even though memory is finite, it is rarely the most scarce resource in any particular computation (though sometimes that is certainly the

case). The actuality of a limitation on memory is left unapparent in practice because, almost always, computing *time* is considered the most scarce resource. In other words, most practical computations tend to run out of time before they run out of storage space. Thus we declare that a CPU is Turing-complete if its instruction set is Turing-complete (with an underlying assumption that memory and its addressability are unbounded), and this theoretical conclusion most closely aligns with common practice.

Thus, when one states that a cellular automaton is Turing-complete, this means that the *activation rules* are Turing-complete [54]. In the same way that declaring CPUs to be Turing complete requires an assumption of unbounded memory, concluding that cellular automata are Turing complete presumes an unbounded number of automata cells. In order to justify that the AP's capability to simulate cellular automata also demonstrates its Turing-completeness, we must first justify the assumption that AP STEs (the analog of the cellular automata's cells) and Boolean gates (needed to define a cellular automaton's activation rules) are both apparently unbounded resources (at least in practice). However, the relatively low counts of STEs (approximately 50,000 per chip) and Boolean gates (about 2,000 per chip) render the assumption of "unboundedness" indefensible, since many applications, when run on realistically-sized problems, quickly exceed the AP's available hardware resources. For this reason, we must consider these components as bounded resources, and explore the AP's computing capabilities in light of the AP's hardware resources being finitely bounded.

3.3.1 Alternating Finite Automata

Our analysis of the AP's computing power relies on a comparison to a generalization of NFAs called Alternating Finite Automata (AFAs) [55]. Recall that a deterministic finite state automaton (DFA) accepts its input string when its unique single active state after processing the input happens to belong to its set of accepting states. A non-deterministic finite state automaton (NFA) is able to transition to multiple states on each input symbol, meaning that there is no *unique* single active state at the end of input, but rather a (possibly large) set of active states. An NFA therefore accepts in the case that any state among its set of active states happens to belong to its set of final states. In other words, this requirement is by nature an existential quantification. We say that a string s belongs to the language of some NFA M provided that, after consuming all of the input, *there exists* an active state which is an accepting state. Formally:

$$s \in L(M) \Leftrightarrow \exists q \in F | q \in \delta^*(q_0, s)$$

where δ^* is an extension of δ which consumes an entire string rather than a single character:

$$\delta^*(q, s) = \begin{cases} q & s = \varepsilon \\ \delta^*(\delta(q, a), s') & s = a \cdot s' \end{cases}$$

An alternating finite state automaton further generalizes NFAs. Like NFAs, AFAs allow for multiple valid transitions on any state-symbol pair. When an NFA branches to a multiplicity of states, the condition for acceptance is “if any paths from here accept, the machine accepts” (again, an existentially-quantified requirement). For an AFA, the acceptance condition may be either existential or universal, the choice of which is defined individually for each state. For some states the acceptance condition matches that of NFAs, others have a universally-quantified requirement: “if *all* paths from here accept, the machine accepts”. Figure 3.5 illustrates the different acceptance conditions of DFAs, NFAs, and AFAs.

Formally defining the acceptance condition for AFAs is similar to NFAs. As with NFAs and DFAs, AFAs are also defined by a state set Q , alphabet Σ , start state $q_0 \in Q$, and a set of final states $F \subseteq Q$. The transition function must be modified in order to represent the quantifiers in the machine. This is done using a function $g : Q \times \Sigma \times \{0, 1\}^{|Q|} \rightarrow \{0, 1\}$. The function g takes as input a state, a symbol from the alphabet, and a bit vector of length $|Q|$ and gives a bit as output. From the perspective of a state q , we say g_q (which defined by $g_q(\cdot, \cdot) = g(q, \cdot, \cdot)$) maps a symbol to a Boolean expression over the state set. If this state is an existential state, that Boolean expression will be the OR of all states it transitions to on the given input (all others are “don’t cares”). If it is a universal state, this Boolean expression will be the AND of all states it transitions to on that input (again, all others are “don’t cares”).

Alternating finite state automata acceptance is defined to be the truth value of a Boolean expression generated by iterative application of this function g on the input set:

$$g^*(q, s) = \begin{cases} 1 & s = \varepsilon \wedge q \in F \\ g_q(a)(g^*(q_0, s'), \dots, g^*(q_{|Q|}, s')) & s = a \cdot s' \end{cases}$$

We restate the behavior of g^* algorithmically:

1. g^* takes as input a state q and a string s , initially q is the start state q_0
2. g^* removes the first character from s , call this a

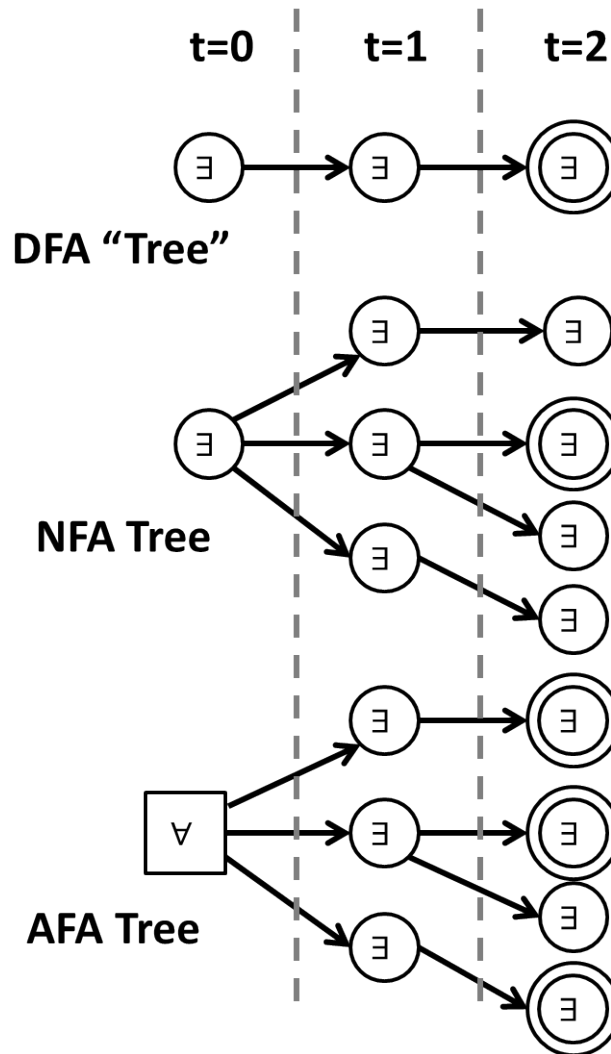


Figure 3.5: Examples of accepting DFA, NFA, and AFA paths. Each circle represents an active existential state, each square represents an active universal state. They are arranged left-to-right to demonstrate a sequence of 2 transitions. Note that the DFA is only in one state at a time, whereas NFAs and AFAs may be in multiple

3. Look up the Boolean expression associated with input a for its current state q
4. Apply g^* to the remainder of s (which we call s') and all states in the automaton, where the response of each state's result is used to evaluate the Boolean expression from the previous step
5. After all paths reach the base-case where s is empty, the terminating states "return" 1 if they are accepting and 0 otherwise
6. These 1s and 0s are propagated up the "call stack" until we reach the Boolean expression from the original call of $g^*(q_0, s)$

| Resource | Constrained? |
|---------------|--------------|
| States | Y |
| Boolean gates | Y |
| Counters | Y |
| Gate Fan-in | N |
| Active set | N |
| Time | N |

Table 3.1: Constrained vs. Unconstrained resources available on the AP

7. Accept if this first Boolean expression returns 1

Just as converting a nondeterministic finite automaton of k states into an equivalent deterministic finite automaton requires up to 2^k states, we incur the same exponential expansion in state-count when converting an alternating finite automaton into a non-deterministic one. Thus, converting an AFA of k states to an equivalent NFA requires up to 2^k states, and further converting it into an equivalent DFA requires up to 2^{2^k} states [55] (in some worst-case example these upper bounds are actually unavoidable). This ability to convert an AFA to an equivalent NFA shows that the class of languages accepted by an AFA is exactly that of an NFA, i.e. alternating finite state automata accept exactly the regular languages.

3.4 Micron's AP Accepts the Regular Languages

In this section we will show that proper parameterization of the automata processor implies that the AP's computability class is exactly the set of regular languages. To do this, we model the AP as a non-deterministic finite state automaton with added Boolean logic gates and counter elements. In Table 3.1 we identify all resources available in defining an AP-style machine. Each resource is identified as either "constrained" or "unconstrained". A constrained resource is one which is likely to bottleneck a typical application, and therefore we consider it to be a finite parameter. An unconstrained resource rarely becomes a bottleneck, and therefore we consider it to be effectively infinite.

States, Boolean gates, and counters are the clear choices for constrained resources, as the AP is by its nature a spacial computing architecture, and these are the elements which occupy the vast majority of the silicon space. We consider the fan-in of the Boolean Gates to be unconstrained without loss of generality of our results, as one could build arbitrary fan-in gates from several limited fan-in gates (with a commensurate propagation delay, but we assume time to be an unconstrained resource here). We consider the active set (maximum possible number of active states) to be unconstrained because the automata processor hardware allows this to be as large

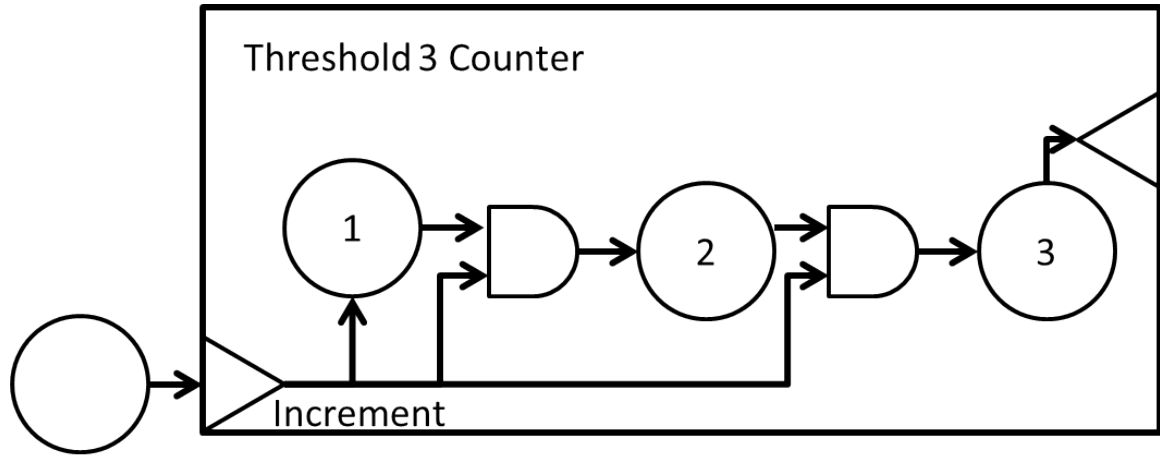


Figure 3.6: Converting a counter with threshold 3 to states and AND gates. All states match on all inputs in this construction.

as the state set. We consider time to be unbounded because the AP, as with any finite automata, makes exactly one transition for each input symbol, and thus always execute within linear time.

3.4.1 Eliminating Counter Elements

We give a construction that converts an AP-style automaton with counters to an equivalent automaton without counters. Recall that counter elements on the Automata processor only become active after their internal counters exceed some reconfigurable threshold. Since this threshold is fixed per automaton, its size is finite in terms of the size of the input string; in other words, this threshold is a fixed “compile time” parameter. For this reason we can eliminate counters using a combination of AND gates and automata states.

In the most simple construction, a counter with threshold t requires t states (label these q_1, \dots, q_t) and $t - 1$ AND gates. Let the state q_i represent the condition in which the counter’s internal count is currently at i . The counter advances its count from i to $i + 1$ only when a state connected to its increment port is active. To simulate this, state q_{i+1} will become active if on any cycle q_i is active as well as some “incrementing” state (one which was connected to the would-be counter’s increment port). An illustration of this construction for a counter with threshold 3 is shown in Figure 3.6.

A more efficient construction would use each state to represent a bit in a binary representation of the current counter threshold. Each state’s activation would then be determined by a ripple-carry adder each cycle, adding 1 when an incrementing state is active, and 0 otherwise. This would only require $\log_2 t$ states to build, as well as about $5 \log_2 t$ gates.

These constructions show that counter elements, in their current form, contribute nothing to the computability powers of AP automata beyond the functionality of states and gates. However,

we now have a resource equivalency between counter elements and Boolean gates. Counters on the AP have a maximum value of 2^{12} , so one counter serves to replace 12 states and 60 gates. While it is difficult to estimate the relative silicon space required for counters, gates, and states without careful examination of the AP chip's layout (which is typically proprietary information), one would expect counter elements to be substantially more space efficient than their state-gate replacements.

Alternative Counter Designs

The threshold counters of the AP do not contribute to its computational ability due to their finiteness relative to the length of the automaton's input string (being a "compile time" fixed parameter). A construction similar to the above will not be possible if the range of values which the counter can represent depends on the length of the automata input (making it a "run time" parameter).

For example, consider an alternative counter design which has two input ports, increment and decrement, and an output port which indicates whether the internal count is 0. Assuming that the count capacity of the counter is an unconstrained resource (and therefore can count arbitrarily high), this new counter simulates a single-character stack. Automata with access to multiple stacks are known to be more powerful than NFAs [50]. Therefore, an AP design with enhanced counters may be able to accept a larger class of languages than only the regular languages.

3.4.2 Eliminating Boolean Gates

Now that we have shown counter elements do not give AP-style automata additional computing powers, the only behavior they have over NFAs is the Boolean gates. Recall the definition of AFAs in Section 3.3.1. The function g defines acceptance by constructing a large Boolean expression (consisting of only ANDs and ORs) from the given string, then evaluating that expression substituting 1 for all final states and 0 for all non-final states. If we remove the "don't care" terms from this substitution, we can see that AP-style automata can be simulated by AFAs.

Consider an AP-style machine in which the state q_i becomes active after input a provided both states q_j, q_k were active in the previous cycle. This transition is therefore defined as $g(q_i, a) = q_j \wedge q_k$. The transition corresponds to an equivalent Boolean expression in an AFA. Thus we can construct an AFA from a AP-style automaton by introducing a new existential start state with transitions to all initial states in the AP-style automaton.

This construction has the issue that AFAs require only ORs and ANDs (from existential states and universal states respectively), whereas AP automata may have negations. A construction in [56] shows that an AFA with negations may be converted to one without, resulting in no more than a

doubling of the number of states.

This construction is the same as a construction from a Boolean automaton to an AFA [57], and is shown in [58].

3.5 Comparison to Circuit Complexity

Now that we have seen that an AP automaton always accepts some regular language, we can explore how the MISD parallelism of the AP might compare to the capabilities of other hardware co-processors. To do this, we relate the theoretical performance of the AP to FPGA- or ASIC-like co-processors by way of circuit complexity.

3.5.1 Circuit Complexity

A circuit is a directed acyclic graph where each node is labelled with an input variable, 0, 1, or a Boolean operator. One unique node is identified as being the output node, and it has no outgoing edges. Each input variable represents a particular bit in a binary input.

Each node in the circuit observes the status of each node connected via an incoming edge. It then performs its prescribed Boolean operator, identifying its status as 1 if that operation resolves to true, and 0 otherwise. The language computed by such a circuit is given by the set of binary input strings for which the output node resolves to 1. The size of a circuit is the number of Boolean operators, and the depth of the circuit is the longest path from an input variable node to the output.

Automata-based models of computation (i.e. finite state automata and Turing Machines) are uniform, as the same automaton computes inputs of any length. By contrast, Boolean circuits are a non-uniform model of computation, meaning that inputs of different lengths will be computed by different circuits. We say a function is computed by a particular automaton, as the same automaton suffices for any input length. In a circuit model of computation, we must say that a function is computed by a *family* of circuits, where a family of circuits is a set of circuits $\{C_0, \dots, C_n, \dots\}$ where C_i computes the function when the input size is exactly i .

Typically, these circuits are given a uniformity restriction. This gives a Turing machine complexity bound on the computation of a circuit C_n for a given input length n . For example, the $DTIME(n^2)$ -uniform size- n circuits would be the class of languages computable by a family of circuits, each with a linear number of gates relative to the input size, and computable in quadratic time (again, relative to the input size).

3.5.2 Nick's Class

The complexity class NC, for Nick's Class, represents the class of problems solvable by a family of circuits of polylogarithmic depth ($O(\log^p n)$ for some choice of p) and polynomially many gates, or equivalently the class of problems solvable in polylogarithmic time by a polynomial number of parallel Turing Machines. The class was named after Nick Pippenger by Stephen Cook in honor of his research on such circuits.

This complexity class serves the role of describing a class of efficiently-parallelizable problems [48]. Within NC there is an (conjectured) infinite hierarchy parameterized by the degree of the exponent in the polylogarithmic run time. We say that the class NC^i is the class of problems solvable by a family of circuits of depth $O(\log^i n)$, so NC^0 circuits have constant depth, NC^1 circuits have log depth, etc.

3.5.3 Circuit Complexity of the AP

It is well-known that the regular languages belong to the class NC^1 [59]. This means that any language which is computable on an AP automaton can be computed by a Boolean circuit (and therefore an FPGA) in logarithmic time using a polynomial number of resources. Since the AP always requires linear time to process its input, other hardware co-processors see asymptotic time speedups over the AP for *every* application.

This asymptotic time speedup can be seen as a time/space tradeoff, however. The method of converting finite state automata to these NC^1 circuits requires a syntactic monoid. This is a representation of the automaton as an associative algebra such that each character and each state maps to one term in the algebra. Machine acceptance on that string could then be determined by:

1. begin with the term representing the start state
2. multiply this term by the first symbol's algebraic term
3. repeat until out of input
4. accept if the resulting term belongs to an accepting set of terms

Since this algebra is associative, the multiplications in step 2 can be done out-of-order, allowing for parallel execution.

The space required to represent these circuits is dependent on the size of the monoid required. While it is difficult to find tight bounds on this size (their size depends in part on the size of the input alphabet), in general $|Q|^{|Q|}$ monoid terms are required to simulate a deterministic finite state automaton with state set Q [60].

Combining this with the above results, if we represent an AP machine as an AFA, converting a k -state AFA to a DFA will result in 2^{2^k} states, meaning that the size of the monoid would be:

$$(2^{2^k})^{2^{2^k}}$$

However, [55] states that reversing an AFA (that is modifying it to accept a string s if and only if it would originally accept the reverse of s) allows for conversion to a DFA using only 2^k states, giving the monoid size to be:

$$(2^k)^{2^k}$$

In either of these two cases, a circuit-based model of computation would require at least a double exponential factor more than the equivalent automaton. This suggests that for many applications, an automata-based model will be more feasible than a circuit-based model.

3.6 Summary

In this chapter we discussed various different types of finite state automata as well as the Micron Automata Processor. Through this discussion we showed the major claim of the chapter: the Micron Automata Processor can compute no more than the regular languages.

We additionally showed that with a small modification to the AP, a new counter design, we could enable the AP to compute a large class of languages. This motivates future investigation into the hardware requirements of making such a change, and a search for new applications which the change would enable.

Finally we gave a complexity-theoretic comparison between the Automata Processor and circuit-based processors (like FPGAs). The conclusion we can make from this discussion is that many applications are unlikely to be accelerated via a circuit-based computation due to the immense space requirements needed. This result only extends up to the current knowledge of DFA to circuit conversions, and motivates finding techniques that are more efficient in gate count relative to automata size.

Chapter 4

Pseudorandom Number Generation using Parallel Automata

Automata-based computing has seen promise as a potential new component for heterogeneous computing. Micron’s Automata Processor has been shown to provide substantial speedups for pattern-matching type problems over von Neumann executions by serving as a simulator for nondeterministic finite automata.

We show that automata-based computing has an unexpected breadth of applications with our surprising pattern-obscuring problem: pseudorandom number generation, i.e. using a small amount of input randomness to produce a large amount of output which behaves similarly to random. This result demonstrates that automata-based hardware accelerators, like the Automata Processor, show potential in providing high-throughput, low-power pseudo-random number generation. The current specifications of the Automata Processor suggest that our approach can create 4.1GB/s of pseudorandom data of the highest quality per each 4W chip (131.2GB/s per board). Future manifestations of the hardware on more state-of-the-art memory technology should enable throughput of at least 40.5GB/s per chip (1.296TB/s per board), with $6.8\times$ better power efficiency than state-of-the-art GPU designs.

We perform a theoretical analysis of our pseudorandom number generator in terms of its algorithmic complexity and cryptographic security. This shows that our pseudorandom number generator is cryptographically secure, belongs to the class NC^0 , and has linear stretch under the assumed hardness of an automata learning problem. We also analyze our algorithm’s behavior in the case that it is given non-random input.

Finally we present an alternative approach to Bloom filtering through execution of parallel Markov chains, whose performance behaviors only hold given the pseudorandomness property of APPRNG. This application shows that APPRNG, in addition to being a valid method for pseudorandom number generation, is also a necessary first step in opening up a new domain for automata-based computing: probabilistic algorithms from parallel Markov chains.¹

4.1 Motivation

While the class of problems shown to benefit from the MISD parallelism of automata processors is steadily growing, the efficacy of these architectures is still restricted to only a few applications, such as network intrusion detection [61], association rule mining [62], natural language processing [63], entity resolution [64], and machine classification [65], all of which are intuitively analogous to pattern recognition problems. On the other hand, though originally designed to accelerate graphics-related tasks, GPUs found applications across numerous problem domains and are now recognized as fundamental to heterogeneous computing. Automata architectures require similar diligence and research to determine their extensibility and applicability to additional non-obvious domains beyond the above examples.

We present such a non-obvious (and even counter-intuitive) application of automata computing: pseudorandom number generation (PRNG). A PRNG is a deterministic algorithm which takes a small amount of source randomness as input and produces a large amount of output which behaves similar to randomness. PRNGs are fundamental in a multitude of important domains, including cryptography [66], algorithm derandomization [67], and Monte Carlo methods [68]. This new application is therefore an exciting and necessary first step in evaluating the potential application of automata processing in each of these domains. This application is also fundamentally different from any previous use for automata-based computation. While all results above are by nature “pattern finding” problems, pseudorandom number generation is intuitively opposite, being a “pattern obscuring” problem.

We begin by explaining pseudorandom number generation in Section 4.2. In Section 4.3 we present our algorithm for automata-based pseudorandom number generation. Section 4.4 describes our hardness assumption for security of our PRNG. In Section 4.6 we give a construction of APPRNG on the Micron Automata Processor, doing a performance evaluation on current and future hardware implementations. We discuss modifications to our PRNG for weakly random input in Section 4.7.

¹The work in this chapter was done (in part) collaboratively in [8]. The areas which are principally my contribution are the general APPRNG algorithm (Section 4.3), the characterization of the hardness assumption (Section 4.4), the theoretical performance evaluation (Section 4.5), the adaptation for weak randomness (Section 4.7), and automata-based Bloom Filtering (Section 4.8).

Finally, in Section 4.8 we present automata-based Bloom filtering as an application enabled by APPRNG.

4.2 Pseudorandom Number Generation

A pseudorandom number generator (PRNG) is a function that takes as input a short random bit string and yields a larger output that “seems” random. More formally, a function $f : \{0,1\}^n \rightarrow \{0,1\}^m$ is a PRNG provided that $m > n$ and there is no polynomial time probabilistic algorithm which can, with probability greater than $1/n^p$ for some p , distinguish the distribution of the range of f from the uniform distribution U_m of bit strings of length m . Intuitively, the purpose of a PRNG is to take a small amount of hard-to-produce truly random bits, and through a deterministic method produce a much greater amount of output bits which can be used as if they were generated randomly. The number of pseudorandom bits gained relative to the random bits provided (i.e. $n - m$) is called the stretch of the PRNG.

The existence of PRNGs is currently an open problem. Their existence implies a separation of P and NP , and their non-existence would imply the impossibility of many cryptography primitives. This means there is no known process for vetting candidate PRNGs by direct means. Instead, we employ two indirect strategies, depending on the PRNG’s target application. In the case of cryptographic use of PRNGs, vetting requires a hardness reduction, i.e., a demonstration that the existence of an algorithm which distinguishes between the range of the generator from a uniform distribution would imply the existence of a polynomial-time solution for some problem which is widely believed to require superpolynomial time. For example, the classic Blum-Blum-Shub [69] PRNG was reduced to the quadratic residuosity problem, which is conjectured to be not polynomial time solvable.

While the cryptographic requirement for vetting a PRNG is certainly the more robust of the two, it is an exceedingly difficult standard to meet. Because of this, few PRNGs satisfying the cryptographic requirement are particularly performant. In applications which require large volumes of random numbers as input (e.g., Monte Carlo simulations), the throughput of the randomness generator is often the performance bottleneck. Therefore, PRNGs targeted for such applications are vetted empirically using statistical tests designed to measure randomness.

To evaluate the quality of a PRNG, these test suites play the role of the polynomial time distinguisher mentioned in the definition of a PRNG. By seeking patterns that are either overrepresented or underrepresented, these tests produce a likelihood that a given (very long) sequence was generated randomly. If the entire battery of tests reports high confidence that a candidate

PRNG's output sequence was random, then that algorithm is determined to be a PRNG. The downside of this approach for evaluating pseudorandomness, as compared to the cryptographic approach, is that the test battery must evaluate *arbitrary* input sequences. Since these tests are unaware of the method for generating the input, passing the tests does not forbid the existence of a polynomial time algorithm "custom designed" for a particular PRNG. By way of example, the sequence of digits of π pass many statistical tests, but could easily be checked for by a custom test looking specifically for the digits of π . A reduction provides more convincing evidence of pseudorandomness, as the failure of all previous solution attempts by many acclaimed pursuers serves as its vetting.

We seek to evaluate the AP-PRNG algorithm using both of the above approaches. The cryptographic vetting gives implications for developing further cryptographic primitives natively in automata processors, and also provides stronger evidence of the quality of the APPRNG algorithm for producing pseudorandomness. The statistical tests allow for the performance evaluation of the AP-PRNG algorithm in terms of throughput and power efficiency on actual hardware, namely the Micron Automata Processor (AP).

4.2.1 Previous Work on PRNGs

The storied uses of pseudorandomness for cryptography and computation has inspired the development of quite sophisticated PRNGs. The Mersenne Twister algorithm [70] is the most widely-used PRNG, and serves as the randomness source for a plethora of programming languages. The Blum-Blum-Schub algorithm [69] is useful as a very simple yet cryptographically secure PRNG, but is outdated by the current dialog in the area. Currently the focus of theoretical PRNG research revolves around the achievable stretch for a PRNG of NC^0 complexity (an NC^0 function can be computed in constant time by a polynomial number of parallel machines). Specifically, this research seeks a PRNG in NC^0 with at least asymptotically linear stretch $O(n)$ (there is evidence favoring one with stretch of $O(n^{1-\epsilon})$) [71]. As for high-throughput PRNGs, the Philox algorithm [72] is currently considered the state-of-the-art. It is specifically designed for massive parallelism by reducing the amount state required per parallel thread, the result is portability to SIMD GPU hardware, achieving a throughput of 145GB/s on an NVidia GTX580.

To experimentally evaluate our PRNG, we subject it to the most comprehensive and stringent collection of tests available, namely the BigCrush test battery from the TestU01 suite [73], which evaluates the randomness of a 2^{43} -bit input. This suite includes all tests from Knuth's Art of Computer Programming Volume 2 on Seminumerical algorithms [74], DIEHARD [75], and the

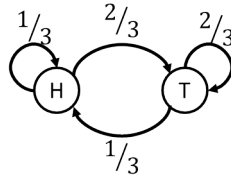


Figure 4.1: An example Markov Chain of a biased coin which lands heads with probability $\frac{1}{3}$ and tails with probability $\frac{2}{3}$

NIST statistical test suite [76]. Surprisingly (given its ubiquity), Mersenne Twister does *not* pass the BigCrush randomness test suite. To our knowledge the only currently available, high-throughput PRNG which passes all BigCrush tests is Philox, and we therefore use this as the state-of-the-art comparison standard.

4.2.2 Markov Chains as Automata

While the AP is designed to be a hardware simulator of finite state automata, APPRNG leverages the AP as a Markov chain simulator (this construction was first presented by Wadden et al. in [77]). A Markov chain is a discrete time, discrete space stochastic process, and behaves much like a finite state automaton where the transitions are dependent on the value of a random variable (whereas an automaton transitions based on an input character). One can emulate this behavior using a Moore machine by randomly choosing the match set of each transition, and then streaming random input into the machine. In this case the probability of a particular transition matching on each cycle is exactly the proportion of the total alphabet which appears in its match set. That is, the probability that a transition which matches on a set of randomly selected characters M is given by $\frac{|M|}{|\Sigma|}$, where Σ is the alphabet size.

Figure 4.1 shows an example of a Markov chain which simulates biased coin flips. In this case the coin lands heads with probability $\frac{1}{3}$ and tails with probability $\frac{2}{3}$. The above construction is then applied to this Markov Chain, which results in an equivalent automaton shown in Figure 4.2.

4.3 AP-PRNG Algorithm

The most important kernel of APPRNG performs a construction to convert Markov chains (which emulate the behavior of fair s -sided dice) into finite automata, in which each state is given a unique label that is emitted as output every time the machine enters that state. An s -sided die

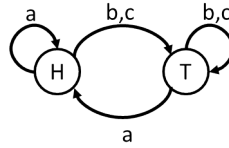


Figure 4.2: An example construction of an automaton which, when given a sequence of random input characters, emulates the behavior of the Markov Chain shown in Figure 4.1.

has s different configurations, one for each side, and is a memoryless process since each roll's value is statistically independent of all previous rolls. A Markov chain which emulates such a die will appear as a complete directed graph of s nodes (including self-looping edges), where each transition occurs with probability $\frac{1}{s}$.

Any automaton constructed to emulate an s -sided die Markov chain, when given random input from an alphabet of size s , produces an output distribution statistically identical to the input distribution. In order to “boost” the length of the output to exceed that of the input, and therefore satisfy that requirement of a PRNG, we run many machines in parallel on the same input. Since each machine produces its own output separate from the others, but they all share input, the expansion of the output size from the input size is multiplicative by the number of parallel machines. For sufficiently many input symbols the amount of output expanded by the parallel machines will exceed the fixed quantity of randomness needed to build the machines.

Since the chains themselves are deterministic, the output's entropy can at best match that of the input. Because the output of the automaton is larger than the random input there must be some correlation among the values of the output bits. The success or failure of APPRNG hinges on how difficult it is for a distinguisher to “discover” this correlation. In order to obscure any patterns, we use some of the random input to build each machine. The symbols upon which each transition matches will be chosen completely at random, the adjacency list of each state generated being a random permutation of the symbol set. This procedure allows for the construction to adapt relative to several parameters:

- The number of machines, call this M
- The number of states in each machine, call this s
- The number of symbols in the input alphabet, call this k
- The number of symbols to be given as input r

For our theoretical analysis, we will say that $k = s$, although for practical situations we can achieve better throughput in the case that $k > s$ (our experimental analysis shows for current-generation automata-based hardware we obtain optimal performance when $s = 8$ and $k = 256$). In this case one can apply the striding construction presented in Section 4.7 to model its behavior. Much of this work studies the properties of the function $G_{M,s,r} : \{0,1\}^n \rightarrow \{0,1\}^m$ where the amount of input randomness is given by $n = M \cdot s^2 \log s + r \cdot \log s$, and the amount of output pseudorandomness is given by $m = M \cdot r \log s$.

To set up the machine, we require $\log s!$ bits of randomness for each of s states in each Markov chain, thus for M Markov chains $M \cdot s \log s! \approx M \cdot s^2 \log s$ bits are required (see Theorem 4.3.1). In total, running AP-PRNG on an input of k symbols requires $M \cdot s^2 \log s + r \cdot \log s$ input bits and produces $M \cdot r \log s$ bits of output. Thus this algorithm is a PRNG if after $r > \frac{M \cdot s^2}{M \cdot r - 1}$ transitions the output is indistinguishable from a random sequence. An overview of the deployment of the AP-PRNG algorithm is shown in Figure 4.3. An example execution is shown in Figure 4.4.

Theorem 4.3.1. *Parallel automata construction for configuration $G_{M,s,r} : \{0,1\}^n \rightarrow \{0,1\}^m$ of APPRNG requires $M \cdot s \log s! = \Theta(M \cdot s^2 \log s)$ bits of random input.*

Proof. To construct the Parallel automata for APPRNG, a random permutation of the symbol set Σ , where $|\Sigma| = s$ uniquely defines each state's outgoing transitions. This procedure thus acts independently on each state in each machine. To count the number of random bits needed to build all machines, we must simply count the number of random bits needed for each state, then multiply by the total number of states across all machines, $M \cdot s$. There are $s!$ ways to permute a list of length s , thus requiring $\log s!$ random bits to sample one such permutation uniformly at random. $\log s! \in \Theta(s \log s)$, therefore the total number of random bits needed for automata construction of $G_{M,s,r}$ is $M \cdot s \cdot \log(s!) \in \Theta(M \cdot s^2 \log s)$. \square

4.4 Hardness Assumption

As mentioned above, at the time of writing no provably secure pseudorandom generator has been discovered, as an existence proof for a pseudorandom generator that has no polynomial time adversary would imply the non-equivalence of P and NP. Instead of disproving the existence of a polynomial time adversary, the security quality of a pseudorandom generator is measured by relating the existence of a distinguishing adversary to the existence of a problem with strong evidence of being “hard”. Ideally, for a particular pseudorandom generator P a polynomial time distinguisher between the distribution of the codomain of P and a uniform distribution over the

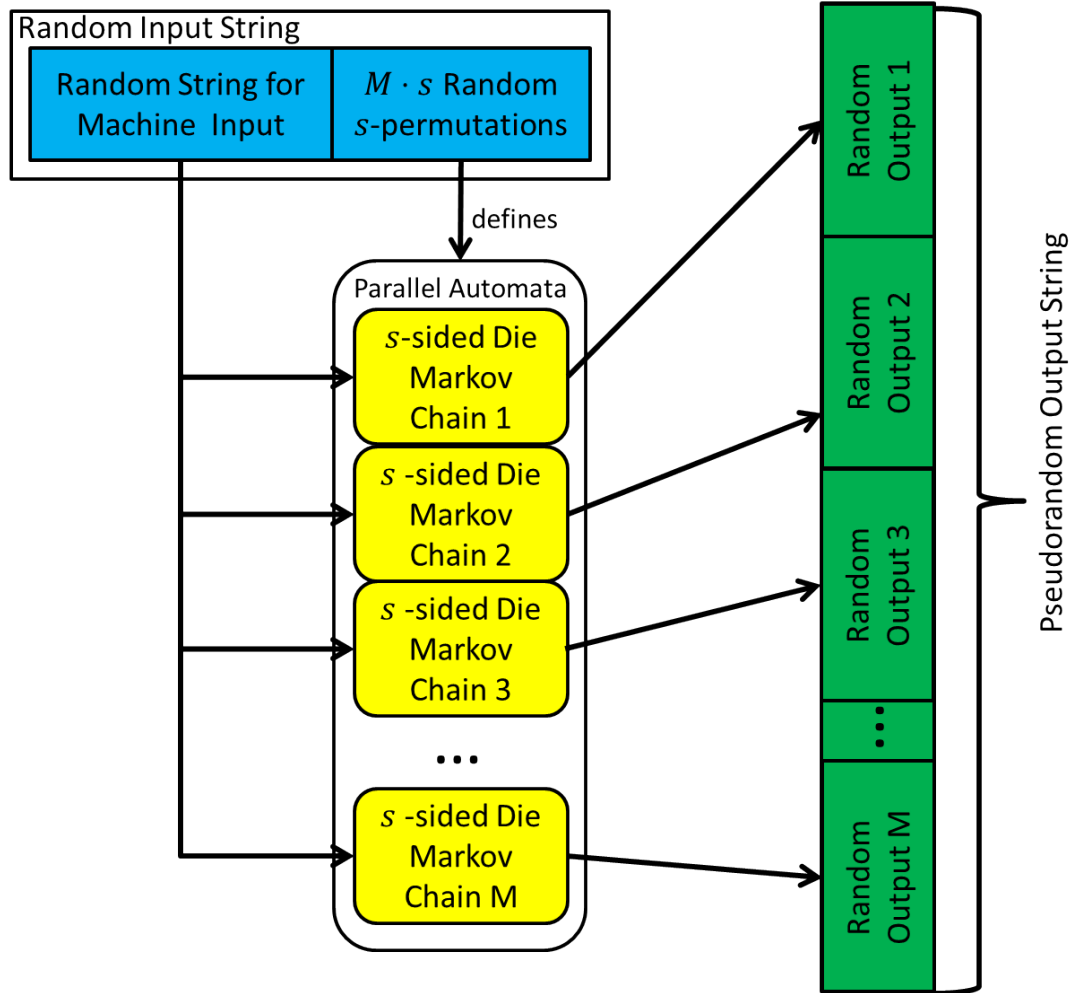


Figure 4.3: Overview of the AP-PRNG algorithm for M total machines with N states each. Each machine simulates a Markov chain representing an N -sided fair die. The N -permutations are used to define the matching symbols on each of the outgoing transitions on each state and requires $N \log N!$ bits of randomness. The random input string requires $\log N$ random bits per symbol. The output will contain M pseudorandom bits per each input bit.

range of P will be shown capable of efficiently solving a well-studied problem which has no known efficient (polynomial time) solutions.

In some cases, a slightly weaker standard for security is acceptable. For example, the famous RSA encryption scheme is widely known to be insecure in the case that prime factorization of integers is not polynomial time solvable, however it has not been shown that the ability to efficiently factor integers is *required* for breaking RSA [78]. In other words, RSA may be easier than factoring integers. The complexity/cryptography community still largely accepts that RSA is as secure as factoring integers, as it is difficult to imagine a way of efficiently solving the problem set forth in the RSA assumption that cannot be used to efficiently factor integers. For APPRNG we provide an

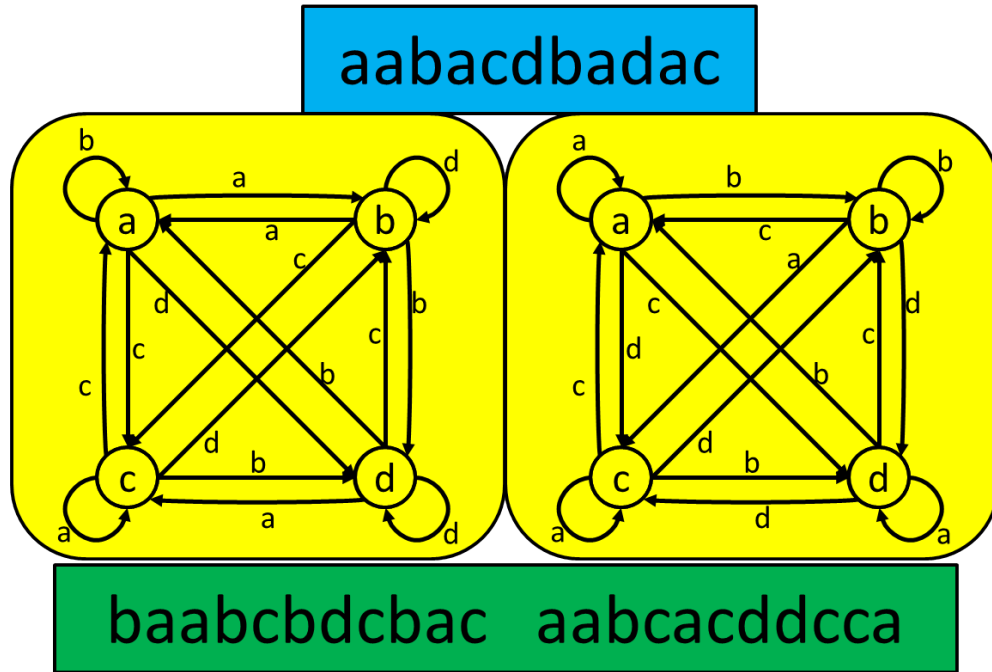


Figure 4.4: Example execution of an instance of APPRNG for 4 states, 2 machines, and 11 input symbols.

argument similar to the justification of the security of RSA. We relate APPRNG to the problem of passively learning hidden finite state automata, an old and well-studied problem which provides no prior art capable of breaking APPRNG.

4.4.1 Hardness Problem Statement

Any adversary for APPRNG must be able to distinguish a random walk through a large machine (one with many states) from a random walk through a small machine (one with few states) by viewing only a concatenated sequence of state labels. To demonstrate this we show that the output to APPRNG and a random string yield the same distribution as a random walk over a large machine and small machine respectively.

Consider a particular configuration of APPRNG $G_{M,s,r}$, where M is the number of parallel machines, s is the number of states in each machine as well as the size of the alphabet, and r is the number of random symbols upon which the machines will transition. In this case the length of the output will be $M \cdot r \cdot \log s$, as there will be one output symbol per input symbol per machine. Instead of representing the transition behavior of APPRNG as that of parallel machines transitioning on shared input, we can represent this behavior as that of a singular machine over a random input through repeated application of a standard cross-product construction on finite

automata [50].

One uses the cross product construction to build a single finite automaton that simulates the behavior of two parallel finite state automata operating in lock-step over the same input. Consider two finite automata, M_1, M_2 given by

$$M_1 = (Q_1, q_{01}, \Sigma, \delta_1, \ell_1)$$

$$M_2 = (Q_2, q_{02}, \Sigma, \delta_2, \ell_2)$$

Let Q_i represent a set of automata states, q_{0i} represent a particular start state, Σ represent the alphabet of input symbols, $\delta_i : Q_i \times \Sigma \rightarrow Q_i$ represent the automata transitions, and $\ell : Q_i \rightarrow \Sigma^*$ represent a labelling of the states in Q_i .

We can simulate the operation of these two automata transitioning in parallel over the same input using the machine

$$M_{1 \times 2} = (Q_1 \times Q_2, (q_{01}, q_{02}), \Sigma, \delta_{1 \times 2}, \ell_{1.2})$$

For $q_1 \in Q_1, q_2 \in Q_2$ and $\sigma \in \Sigma$ we define

$$\delta_{1 \times 2}((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$$

The transition function defines a single transition from the pair of states q_1 and q_2 of the embedded parallel machines corresponding to how each would transition independently over the particular input σ . For $q_1 \in Q_1$ and $q_2 \in Q_2$ we define

$$\ell_{1.2}((q_1, q_2)) = \ell_1(q_1) \cdot \ell_2(q_2)$$

The new labelling function labels the states in the larger machine with the concatenation of labels from the corresponding states of the embedded machines.

Using the cross-product construction described, we are able to combine all M machines in APPRNG into a single machine, call it A_π , with equivalent behavior to all M machines running in parallel. Since each of the M machines has s states, the resulting super machine will contain s^M states. Each of these states will have a label that is $M \cdot \log s$ bits long, thus it produces $M \cdot \log s$ bits of output, as expected. Since the behavior of this machine is equivalent to the behavior of the

other machines operating in parallel, the output of APPRNG will be identically distributed to the sequence of labels given by a random walk over A_π .

The truly random distribution of the same length as APPRNG's output can be modelled by a random walk over a smaller machine, which we will call A_\S . Since each transition emits an output that is $M \log s$ bits long, a random sequence of length $r \cdot M \cdot \log s$ will be distributed identically to a random walk of length r over an automaton with $M \log s$ states, each with a unique label of length $M \log s$. Because a random walk through this machine serves as a random permutation of the input, it does not alter the entropy of the input distribution at all.

Note that $m \log s < s^m$ whenever $s^m > 1$, which will hold for any choice of $m > 1$ and $s > 1$, which any manifestation of APPRNG must satisfy. This means that if there exists a polynomial time algorithm which, when given two distributions of r state labels from random walks (one over A_π and the other over A_\S), distinguishes which sequence came from which machine, then that same algorithm must be able to observe a random walk over an arbitrary machine and determine whether the number of states it contains is above or below some threshold between $m \log s$ and s^m .

Assumption 4.4.1. *There is no polynomial time algorithm which, when given as input a sequence of state labels from a random walk over a DFA, can determine whether the number of states in that DFA is above or below some threshold between $m \log s$ and s^m for some choice of $s, m \geq 2$.*

To the authors' knowledge, in spite of a large amount of related results, there is no prior art which efficiently learns hidden finite state automata by looking at a random walk alone, thus providing evidence of the hardness of APPRNG.

4.4.2 Prior Art in Automata Learning

Prior work on learning automata has focused on efficiently finding a minimal (or as small as possible) finite automaton which computes a hidden regular language through viewing examples of strings which do vs. do not belong to the language. Among the earliest impossibility results relating to this problem was Dana Angluin's 1978 result showing that the problem of finding the minimal automaton consistent with example sets of accepted and rejected strings is, in general, NP-Complete [79]. This problem was later shown to not even be approximately solvable by Pitt and Warmuth in 1993 [80]. In 1994 Kearns and Valiant (a Turing Award winner) showed that the weaker problem of learning acyclic finite state automata consistent with a set of examples is of equivalent difficulty to several cryptographic assumptions, e.g. RSA and factoring of Blum integers, in the case that the learning hypothesis must be a finite automaton [81].

While these impossibility results may make one pessimistic for the future of automata learning (or optimistic for the security of APPRNG), there have been several algorithms which allow for efficient learning in certain situations. Dana Angluin's 1987 L^* algorithm provides an efficient (polynomial time) way of learning an automaton given the opportunity to select query input for the black-box machine, rather than relying on a random sampling of examples [82]. This approach had the limitation of requiring a "reset" mechanism, by which the hidden automaton could be directed to revert back to its start state. This requirement was removed by Rivest (another Turing Award winner) and Schapire in 1989 through the use of homing sequences, which are short sequences guaranteed to explore the neighborhood of a particular state [83]. In 1993 Freund et al. provide a way to learn automata by observing a random walk over its states in a passive way (without being allowed to make its own choice of input) [84], however this result still requires that the algorithm be able to observe the input to the automaton. Angluin in 2015 showed the ability to efficiently learn random DFAs through observing random strings and state information [85].

The above results share one fundamental shortcoming, which prevents them from serving as a distinguisher for APPRNG. They all require the ability to observe the (possibly random) input used to drive the automata, which is forbidden for an adversary of a pseudorandom generator. The above research dialogue demonstrates a clear trend of achieving increasingly general results by diminishing the ability of the learning algorithms to select their own input. In spite of results making progressively weaker assumptions about the input, there has yet to be any results regarding learning algorithms which do not require observing the input to the machine. In summary, through nearly 40 years of research performed by many impressive computer scientists and mathematicians (including two Turing Award winners) studying adjacent results trending toward increasing generality, there has yet to be a result general enough to serve as a distinguisher for APPRNG.

4.5 Theoretical Performance analysis

While for many practical purposes, such as Monte Carlo simulations, the most important metric for a PRNG is the throughput of pseudorandomness produced (this analysis is done in Section 4.6.3), from a theoretical perspective the most important performance metrics are the stretch of the pseudorandom generator (the size of the output pseudorandomness compared to the input randomness) and the computational complexity class (the asymptotic run time of the algorithm). The current state-of-the-art results are those which consider pseudorandom number generators with at least linear stretch, and belong to the complexity class NC^0 [71].

4.5.1 Stretch

The stretch of a pseudorandom number generator $G : \{0,1\}^n \rightarrow \{0,1\}^m$ is the expansion of the number of output bits m relative the number of input bits n , and is given by $m - n$. APPRNG can be configured relative to many different parameters, so here we analyze the stretch of APPRNG relative to each given parameter.

Recall that a particular instance of APPRNG is given by $G_{M,s,r} : \{0,1\}^n \rightarrow \{0,1\}^m$ where M is the number of parallel automata, s is the number of states in each automaton as well as the size of the input alphabet, and r is the number of symbols which will be fed as input into each of the parallel automata. To configure the automata we must compute a random permutation of the alphabet of size s for each of the s states in each of the M machines. Each permutation requires $\log s! \in \Theta(s \log s)$ bits to define, thus with $M \cdot s$ total states the machine configuration requires $\Theta(M \cdot s^2 \log s)$ bits. Then for the r random input symbols, each of which is s bits long, we require $r \log s$ bits. Therefore the total amount of input randomness is given by $n = M \cdot s^2 \log s + r \cdot \log s$.

Each of the M machines will produce one s -bit symbol of output for each symbol of input received, thus for each of the r input symbols given to the automata we will receive $M \log s$ bits of output. This gives the total amount of output pseudorandomness to be $m = M \cdot r \log s$.

The total stretch of $G_{M,s,r}$ is therefore given by

$$\begin{aligned}
 m - n &= (M \cdot r \log s) - (M \cdot s^2 \log s + r \cdot \log s) \\
 &= M \cdot r \log s - (M \cdot s^2 + r) \log s \\
 &= (M \cdot r - M \cdot s^2 - r) \log s \\
 &= (r(M - 1) - M \cdot s^2) \log s
 \end{aligned}$$

Due to the presence of the negative s^2 term, as well as certain practical problems which arise from large machines (namely routability when implemented in hardware), we will consider s to be a small constant (our experiments use $s = 8$). This means that the stretch for $G_{M,s,r}$ belongs to $\Theta(r \cdot M)$. If we vary only the number of symbols used as input or the number of parallel machines used the stretch will be linear. If we vary both at the same rate the stretch will be quadratic.

This analysis of the stretch of APPRNG will only hold if arbitrarily large choices of r and M allow for APPRNG to be a pseudorandom number generator for a fixed choice of s . This is certainly not the case when r alone varies. As the number of input symbols given to each machine increases

the inherent correlation in the machines' behavior will become more pronounced, i.e. the output will become easier to distinguish from random. This behavior is borne out in our experimental analysis shown in Section 4.6.3. While we see that APPRNG demonstrates different quality for different choices for machine count, this does not appear to be related to the size of this count, but rather seems to be attributable to certain number theoretic behaviors. For APPRNG to remain secure for an arbitrary choice of M , we must strengthen our hardness assumption 4.4.1:

Assumption 4.5.1. *There is no polynomial time algorithm which can, when given a sequence of state labels from a random walk over a DFA as input, can determine whether the number of states in that DFA is above or below any threshold.*

We again emphasize that there has been no prior art which provides an efficient solution to this problem.

4.5.2 Complexity

The current trend in pseudorandom number generator research is the exploration of generators belonging to the class NC^0 . These are of particular interest because they represent problems computable in constant time in parallel, and have the property of each output bit depending on only a constant number of input bits.

Under the assumption that APPRNG remains cryptographically secure for arbitrarily large M , APPRNG belongs to the class NC^0 . For constant s and r , if we have M Turing machines operating in parallel we can compute APPRNG in constant time. Each of the M machines will simulate a single DFA. It will receive as input the constant number $(s^2 \log s)$ of random bits needed to configure its particular automaton, which are specific per automaton. Each Turing machine will then receive a copy of the $r \log s$ bits of random input for the machines, which is also a constant value in this scenario. The time required for each machine to process its input is therefore constant.

In summary, in the case that Assumption 4.5.1 holds, APPRNG is a cryptographically secure pseudorandom number generator with linear stretch belonging to NC^0 . Should Assumption 4.5.1 fail to hold, but Assumption 4.4.1 hold, then we could vary M and r together, causing the complexity of APPRNG to belong to $NC^{\frac{1}{2}}$.

4.6 AP-PRNG in Practice

In the above sections we discussed the theoretical properties of APPRNG, including the effect of the input parameters on the quality of the asymptotic complexity of the algorithm, and the quality of the pseudorandom output. We now seek to demonstrate APPRNG's applicability as a

high-throughput algorithm for producing high-quality pseudorandom numbers. To do this, we modify the execution model of APPRNG to produce arbitrarily large output. First, We build M machines with s states each over a b -bit alphabet. Next, we run the machines over r random input symbols of b bits each. After these r inputs we rebuild all machines using new randomly-generated transitions. This continues until we produce the desired number of output symbols.

In order to evaluate the practical value of APPRNG to applications requiring high-throughput randomness, we must answer the following questions:

1. How does the automata state-count effect the quality of output? Reconfiguration of automata matching on the Automata Processor is expensive, so we would like to choose the number of states which allows for the largest value of r , thus prolonging machine reconstruction as long as possible
2. How does the number of parallel automata effect the quality of output? If increasing the number of automata we run in parallel limits the size of r , then by using more on-chip resources may actually harm the effective throughput by requiring more frequent machine reconstruction.
3. What is the maximum value of r , i.e. the longest we can delay automata reconstruction, based on the best choices of parameters in the previous questions, while still preserving pseudorandom behavior?
4. What minor hardware adjustments could be made to the Automata Processor to increase the value of r , and therefore our throughput?
5. The current configuration of the Automata Processor uses DRAM built using a 40nm process (state-of-the-art in ca. 2009 [86]). How would updating the AP architecture to current-generation transistor technology and memory specification impact the throughput of AP-PRNG?

4.6.1 Hardware Constraints

The first step in constructing APPRNG is to build a set of automata emulating fair Markov chains. The number of states in each chain should be a power of two in order to ensure a uniform distribution of 0s and 1s in the output. After each transition, each chain reports its current state, thus emitting $\log_2(s)$ bits of output for each input.

By building more parallel automata, and interleaving their output, we linearly increase the amount of pseudorandom output relative to the number of inputs. If we have a single 2-state

Markov chain, it will only emit 1 bit of output. If we have a b -bit alphabet, b 2-state Markov chains will create as much output as the input given.

Only 32 STEs out of 256 in an AP block can report in the current architecture. This means that either total number of STEs or number of reporting element could manifest as the bottleneck for number of machines. An N -state Markov chain requires N reporting STEs, thus we are limited to 16, 8, and 4 chains per block for 2-, 4-, and 8-state chains respectively by reporting capacity. Each also requires $N^2 + N$ STEs, thus we are limited to 42, 12, and 3 chains per block for 2-, 4-, and 8-state chains respectively by STE capacity.

Assuming we can use all the reporting elements to draw output, that there are no bottlenecks in the reporting architecture, and we never need to reconfigure the machines, the Automata Processor can create 51 GB/s of pseudorandom output.

4.6.2 Sensitivity Analyses

To test APPRNG performance and quality we ran an implementation through the test batteries in the TestU01 statistical test suite [73] to assess quality of random output. Because fully-operational APPRNG hardware is not yet available, this is a simulation of APPRNG behavior on conventional hardware, rather than on an automata-based accelerator.

TestU01 consists of three main test batteries: SmallCrush, Crush, and BigCrush. SmallCrush contains 10 statistical tests, and is meant to check obvious statistical patterns in a random sequence. Crush applies 96 statistical tests (144 total test statistics), and BigCrush applies 106 tests (160 test statistics). Because we're emulating APPRNG without the AP as a hardware accelerator the BigCrush test battery can take 3-7 days to complete on a single core CPU. We therefore use SmallCrush and Crush for course-grained refinement of APPRNG parameters to quickly identify trends. Once we have identified a promising set of parameters with these weaker tests, we use BigCrush, the most comprehensive test suite available [87, 73], to verify that the choice configuration provides good random quality. We then use that choice of parameters to derive performance on AP hardware and compare to state-of-the-art high-throughput PRNGs.

APPRNG, as with any pseudorandom number generator, requires a source of random bits to construct machines and produce the input string. For this evaluation we use Philox32x4_10 [72] to provide all random input. As is expected for any PRNG, lower quality random sources result in lower quality output from APPRNG. Philox was a natural choice for high-quality input as it is the most performant generator available which passes all of the BigCrush tests. We performed all evaluations on a cluster of Intel i7-4820k CPUs operating at 3.7 GHz with 64 logical cores.

| Number of Markov Chain States | 2 | 4 | 8 |
|-------------------------------|-----|---|---|
| Average Number of Failures | 5.5 | 2 | 0 |
| Distinct Number of Failures | 6 | 2 | 0 |

Table 4.1: It is statistically harder to identify correlation between chains with more states.

Effect of Automata Size

We configured APPRNG to have a reconfiguration threshold of $r = 50,000$ input symbols, 384 parallel chains, each with 2, 4, or 8 states. We ran 16 trials over SmallCrush, the results shown in Table 4.1.

We see that state count in the automata profoundly impacts the pseudorandom quality of the output. The 2-state automata fail an average of 5.5 tests, failing 6 of the 10 tests in SmallCrush. The 4-state automata fail 2 of the 10 tests. SmallCrush was unable to distinguish the 8-state case from random, having passed all tests.

This trend should be expected. The more states in each automaton, the more complex their transition behavior, and the greater the number of bits required to construct the machines. This means that two small machines should reveal their correlation after fewer transitions than larger machines. This effect is compounded when considering large sets of many machines, allowing for more automata transitions before their correlations become apparant, requiring reconfiguration.

Effect of Automata Count

Next we explore the impact that number of parallel 8-state automata has on the quality of pseudorandom output. We explore performance with from 32 to 768 parallel automata operating in parallel over shared random input, using Crush to evaluate random quality. The results of this experiment are shown in Figure 4.5. With the exception of the obvious outliers at 352 and 768 machines, we saw no trends in the relationship between the number of parallel automata and the quality of random output. This suggests that increasing the number of parallel automata does not decrease the quality of pseudorandom behavior.

We hypothesize that the outliers occurred due to a number-theoretic property that the output exhibits. Many of the Crush tests operate by considering blocks of 32 bits as integers. Our tests concatenate machine outputs from every 8-state chain together in a round-robin fashion. Thus in the case of the outliers at $352 (= 32 \times 11)$ and $704 (= 64 \times 11)$ each machine contributes to the same bits in each 32-bit integer (we have no explanation for the co-occurrence at multiples of 11). To test this hypothesis we altered the manner in which we concatenate the output so that each machine produced 32 bits before performing the round-robin interleaving. The results of this experiment

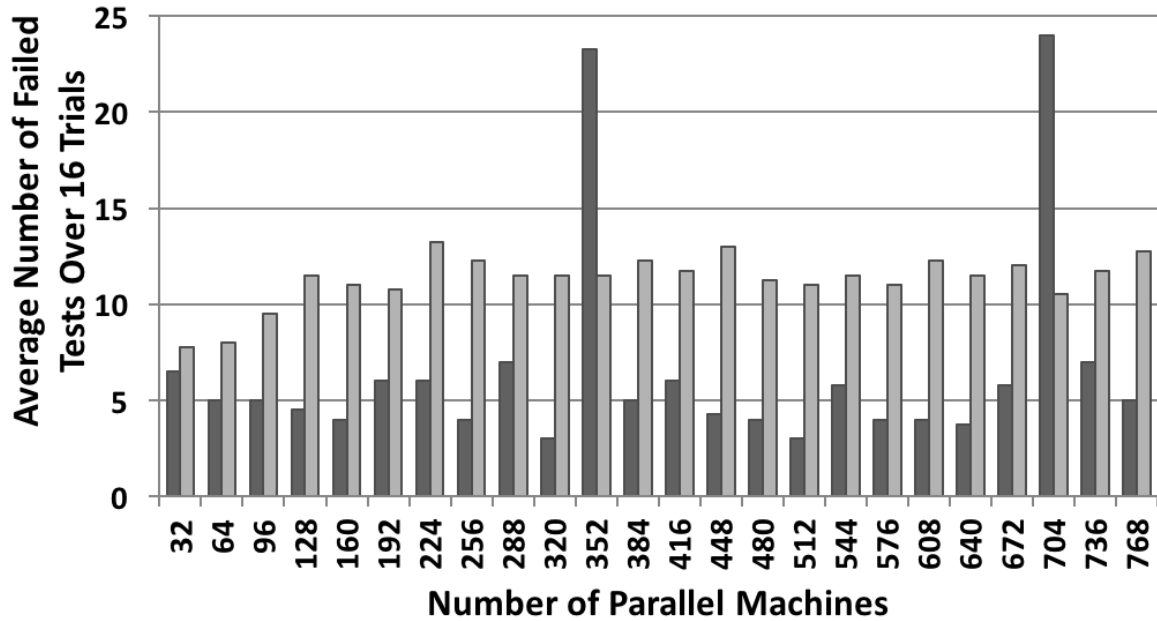


Figure 4.5: Each bar represents the average number of Crush failures over four trials for parallel 8-state Markov chains with a reconfiguration threshold of 200,000. The darker bars represent failure rates when interleaving output bits. Spikes in failure rates occur when the same Markov chains always contribute to the same bits in output integers. The lighter bars represent failure rates when successive output from a single Markov chain contributes to a single output integer. This eliminates the spike in failures, but reduced overall performance.

are also shown in Figure 4.5.

The spikes in occurrence of test failures disappear after this modification. To avoid further cases of such number-theoretic attacks on APPRNG we choose a prime number of parallel automata. This minimizes the chance any machine from contributing to the same location when the output is broken into blocks. The maximum number of 8-state automata which can fit onto an Automata Processor chip is 576, therefore we use 571 parallel machines (the largest prime number less than 576). We use this configuration of 571 parallel 8-state machines for our final performance analysis.

Effect of Input Size

To show the impact increasing time between automata reconfigurations has on the output quality, we ran four trials of 726 8-state automata through the Crush test suite varying the number of inputs from 20,000 to 90,000. It should be the case that increasing the number of inputs given before reconfiguration, the more likely the output would appear non-random. Decreasing the number of inputs given before reconfiguration, however, reduces the performance of APPRNG, as each reconfiguration incurs a substantial time penalty. We seek to find a choice of reconfiguration threshold to optimize a performance/quality tradeoff. Our experimental results are shown in Figure 4.6.

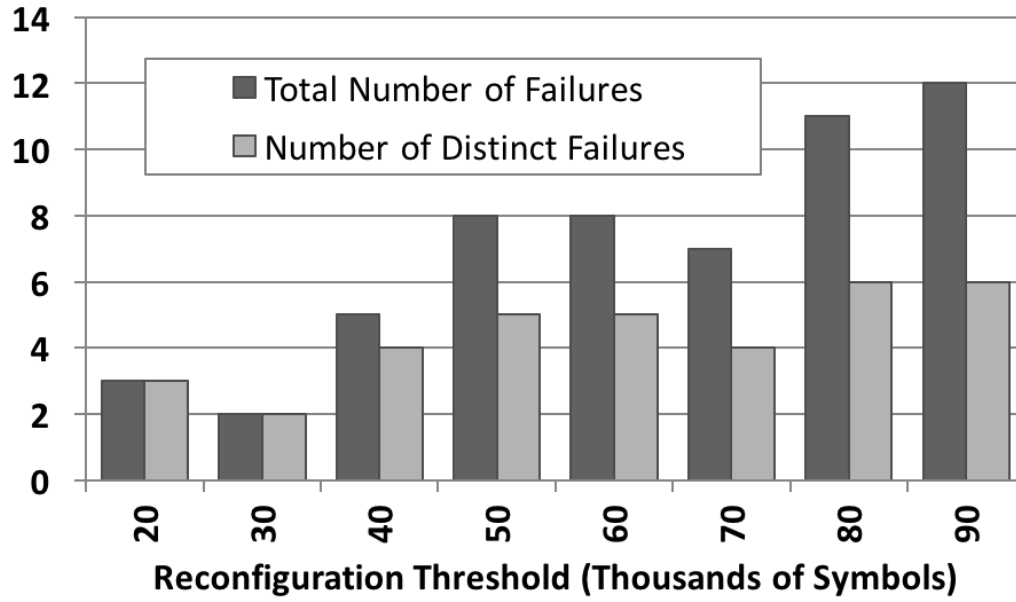


Figure 4.6: As the reconfiguration threshold increases, it becomes easier for statistical tests to identify non-random behavior.

Figure 4.6 shows that the quality of random output is inversely correlated with the length of the reconfiguration threshold. Concerningly, even a threshold of 20,000 input symbols fails to consistently pass all Crush tests. In exploratory tests done (not shown in the figures), some BigCrush tests failed for thresholds as low as 10,00, requiring shorter thresholds to match the quality of Philox. As the Automata Processor only requires 7.5ns to consume a symbol, but 45ms to reconfigure, a reconfiguration threshold of 10,000 cases the AP to spend 99.83% of its time reconfiguring, resulting in 48MB/s of output.

We have found that the reconfiguration threshold can be dramatically increased if we can reduce the impact of neighbor dependence. The algorithm constructs input by interleaving bits in a fixed order, requiring nearby machines to have uncorrelated behavior. To mitigate this effect we investigated using support hardware or software to reorder output bits of a group of automata.

We ran 4 trials of BigCrush on the output of 571 8-state automata, randomly permuting positions of every group of 32 automata after every 1,000 symbols. The results of this experiment are shown in Figure 4.7. Without the permutation step APPRNG failed the Crush tests with a reconfiguration threshold of 20,000 symbols. With the reconfiguration APPRNG passes all BigCrush tests with a reconfiguration threshold of 1,000,000. While we currently implement this step in software, we hypothesize that the permutation could be done by a support ASIC or support processor.

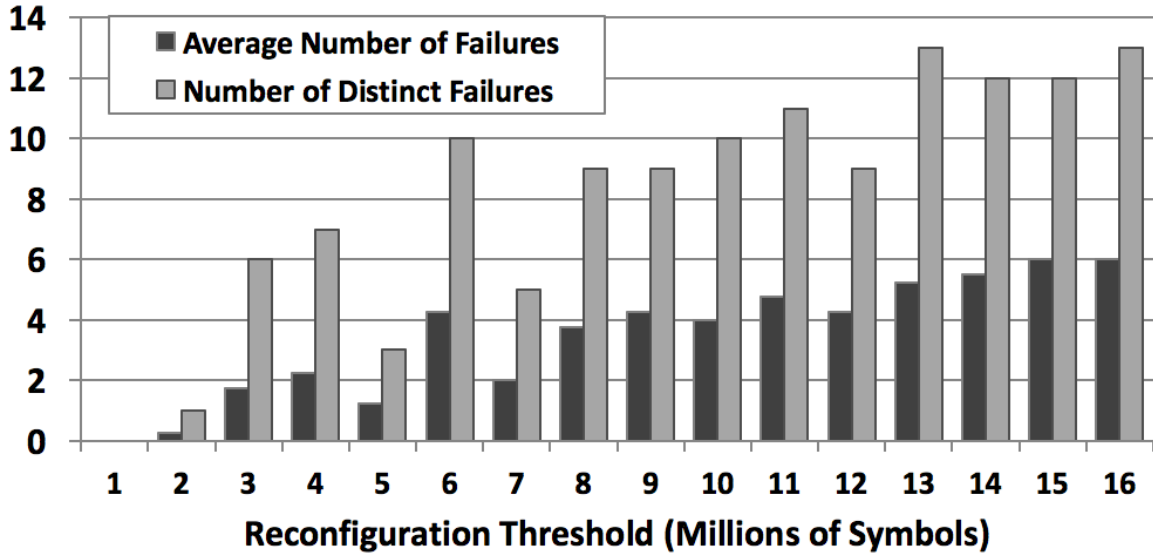


Figure 4.7: Output quality of AP PRNG with output permutation greatly increases quality of random output. AP PRNG passes all tests in BigCrush with a reconfiguration threshold of at least 1,000,000, and at most 2,000,000

4.6.3 AP-PRNG Performance Model

Based on the above sensitivity analyses, we evaluate the performance of APPRNG on first-generation Automata Processor architecture configured for 571 chains, 8-states, a reconfiguration threshold of 1,000,000, and a permutation threshold of 1,000. The Automata Processor operates at 133MHz, consuming 1 8-bit symbol every 7.5ns, regardless of the number of automata operating in parallel.

Figure 4.8 shows the predicted APPRNG throughput as we vary reconfiguration threshold. When $r = 1,000,000$ APPRNG produces 4.1GB/s of pseudorandom output per each proposed AP chip, while consuming 200.7MB/s of random input per chip.

A clear advantage APPRNG has over other pseudorandom generators is its flexibility. A user is easily able to make tradeoffs between pseudorandom output quality and throughput. This is useful in the case that a user has a power- or performance-bound application or simulation in which weak randomness suffices. This user can greatly increase the reconfiguration threshold r to gain throughput. Our model shows that for $r = 10,000,000$ a single first-generation Automata Processor chip can produce 17.8GB/s of output. This allows for users to increase performance (by increasing the reconfiguration threshold) or decrease power requirements (by using fewer chips) if weak randomness is acceptable.

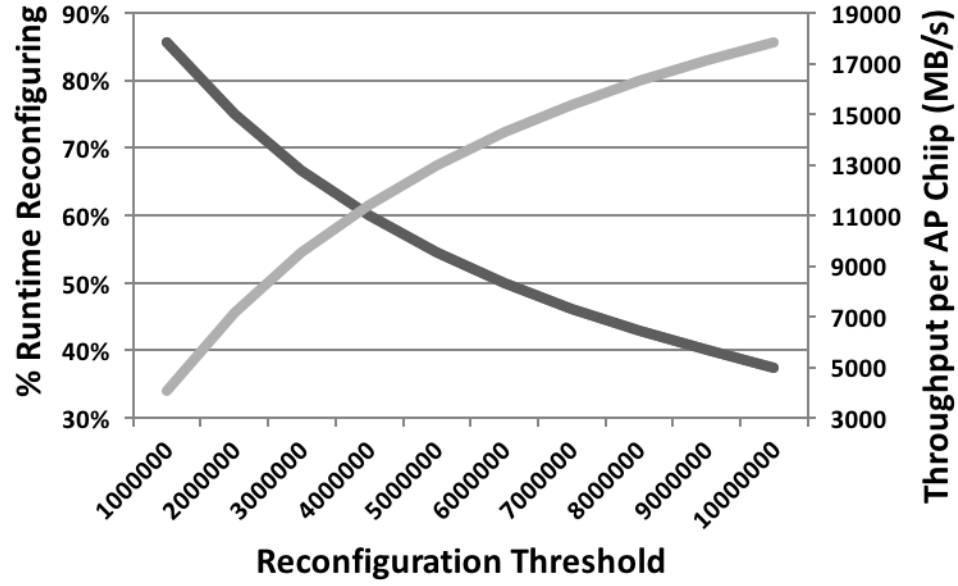


Figure 4.8: Percentage of runtime spent reconfiguring vs. AP PRNG throughput with different reconfiguration thresholds. Performance increases dramatically if AP PRNG is able to reconfigure less frequently.

| | |
|--|----------|
| Frequency | 133MHz |
| Cycle Time (T_c) | 7.5ns |
| STE Size | 256 bits |
| Random State per Chip ($ChipState$) | 1.17MB |
| Est. AP Reconfiguration Time (T_r) | 45ms |

Table 4.2: First Generation AP Architectural Parameters

| | |
|---|-----------|
| States per Markov chain (s) | 8 |
| Markov chains per AP Chip (M) | 571 |
| Input Reconfiguration Threshold (r) | 1,000,000 |
| Permutation Width (P_W) | 32 |
| Permutation Reconfiguration Threshold (P_R) | 1,000 |

Table 4.3: AP PRNG Parameters

| | |
|--|-----------------------|
| Chip Output per Input Symbol (O) | $\log_2(s) * M$ |
| Random Generation Time (T_R) | $r * T_c$ |
| Runs per second ($Runs$) | $1 / (T_{Run} + T_r)$ |
| AP PRNG Throughput (P) | $Runs * O$ |
| Random Input Stream Rate (In_s) | $Runs * r$ |
| Random Input Required for Reconfiguration (In_r) | $Runs * ChipState$ |
| Random Input Required per Permutation (In_p) | $P_W \log_2(P_W)$ |

Table 4.4: AP PRNG Performance Model

| Memory Technology | DDR3 | DDR4 | HMC 2.0 |
|--|------|------|---------|
| Peak Throughput (GB/s) | 12.8 | 17.0 | 320 |
| T_r (μ s) | 91.4 | 68.8 | 7.3 |
| AP Chip Output (GB/s) | 28.2 | 28.3 | 28.5 |
| Throughput Limited AP Chip Output (GB/s) | 12.8 | 17.0 | 28.5 |

Table 4.5: AP PRNG performance modeled on different memory technologies. AP PRNG throughput is limited by peak memory throughput for DDR3 and DDR4 technologies.

Future AP Hardware

First generation AP hardware is projected to have an output throughput of 436.9MB/s, serving as an upper bound on the throughput of off-chip pseudorandomness for APPRNG. Reconfiguration time, which we have shown to be especially burdensome for APPRNG, is projected to be 45ms. These limitations, however, are limitations resulting from AP hardware design decisions, rather than fundamental limitations of APPRNG. The STEs in the AP are implemented using DRAM, and the 45ms figure is much larger than what one would expect for native memory I/O speeds.

We hypothesize the optimal performance of APPRNG assuming that automata reconfiguration can be done at native data rates of DDR3, DDR4, and Hybrid Memory Cube (HMC [88]) throughputs. These projections are shown in Table 4.5.

In order to reconfigure 571 8-state automata we only need to reconfigure the matching symbols of each transition (64 total transitions per automaton), as the topology of the automata remains the same. For a single chip, this requires 1.17MB of data be changed per reconfiguration. If we are able to reconfigure at native DDR3 speeds, our model in Table 4.5 predicts 28.2GB/s of on-chip pseudorandomness. The peak output throughput of DDR3 is only 12.8GB/s, however, which serves as the off-chip performance bottleneck.

Using state-of-the art high-throughput memory technologies, such as HMC, our performance model predicts APPRNG can produce 28.33GB/s of on-chip pseudorandomness. HMC provides much higher output data rates, and is capable of matching the APPRNG throughput.

Future implementations of the AP architecture implemented on a more modern transistor process should also have increased capacity for more STEs, translating to more parallel automata in APPRNG. If we adjust our model for a $1.41\times$ increase in capacity per AP core, APPRNG can produce 40.5GB/s of pseudorandomness per chip.

APPRNG Power Requirements

While the overall power performance on APPRNG will certainly depend on the deployment scenario, we project that APPRNG on the Micron Automata Processor should be much more

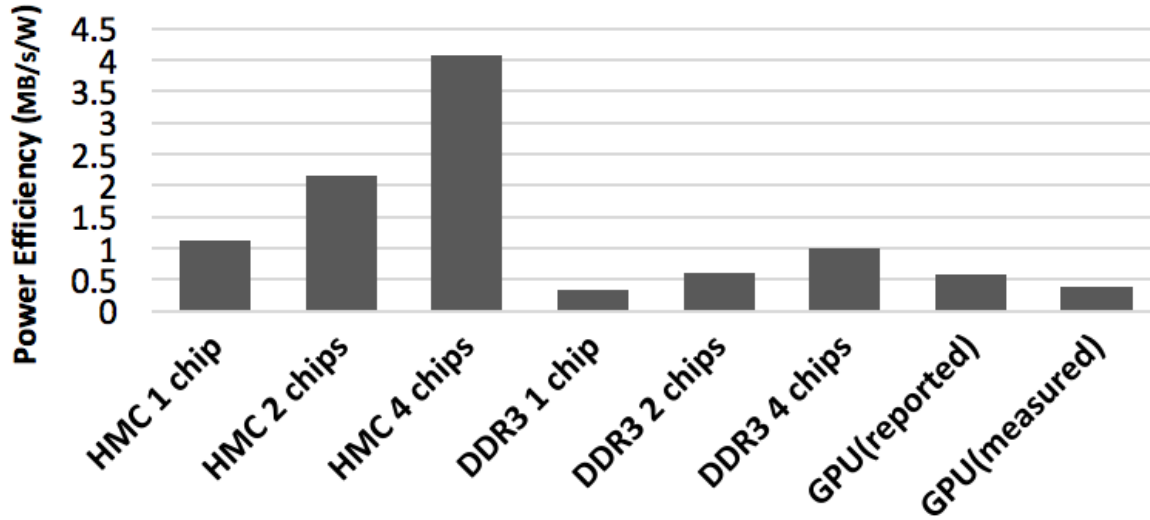


Figure 4.9: AP PRNG is up to $6.8\times$ more power efficient than the highest-throughput reported GPU PRNG depending on the deployment scenario.

efficient than implementation on other architectures. The GTX 580 GPU used in Salmon et.al [72] has a TDP of 244W, while each DDR3-based AP chip has a projected TDP of 4W, and stacked HMC-based memories are projected to use 70% less energy than DDR3. Figure 4.9 shows the PRNG efficiency of a few different realistic AP PRNG deployment scenarios. All AP deployment scenarios require a support processor to generate random input and configure the AP. We assume that the support processor consumes 35W, a reasonable assumption for a single CPU core. The configuration with 4 AP chips implemented in an HMC technology produces 4MB/s/W, $6.8\times$ more power efficient than the best performing GPU PRNG reported in the literature [72], and $10.8\times$ more power efficient than our measured experiments using the current library implementation of *Philox32x4_10* on an NVidia K20C GPU.

Disregarding support processor power consumption, and conservatively assuming a 4W TDP per AP chip, AP chips are $6.8\times$ more power efficient than the reported GPU implementation.

4.7 Sensitivity to Weakly Random Input

By definition, the input to a pseudorandom number generator is required to be truly random. In the case that non-random input is given to a pseudorandom number generator, no guarantees can be made about the pseudorandomness of the output data.

In practice, randomness is typically derived by a computer in any of a number of ways, for example using insignificant bits from the computer's clock, or using biometrics such as the pattern of user keystrokes. These sources are chosen under the assumption that there is some inherent

randomness to the sampling of these processes, however these sources are certainly not truly random. True randomness is very hard to come by, with the only well-respected truly random sources coming from measuring quantum effects, such as measuring the decay of radioactive isotopes.

Since true randomness is expensive, and more easily-obtained sources are only weakly random, a pseudorandom generator could easily be bottlenecked by its ability to obtain its random input. To mitigate this bottleneck we demonstrate how APPRNG could be adapted to produce pseudorandom input even in the case that the random source is only weakly random, thereby broadening the circumstances in which APPRNG provides high-throughput pseudorandomness.

4.7.1 Entropy Extractors

Prior work on using a weakly random source to drive a process requiring true randomness primarily relies on the use of entropy extractors. An Entropy extractor is an algorithm which takes as input a large sequence of n bits from a weakly random (i.e. somewhat biased) source, as well as a small sequence of d bits from a truly random source, and mitigates the bias by producing a sequence of m output bits that are substantially closer to being uniformly distributed. We emphasize that the definition makes a *statistical* requirement on the output of an entropy extractor rather than a computational complexity requirement. In other words, the output must *actually* be statistically close to random rather than being computationally indistinguishable from random. Note that this definition still requires using some true randomness, which is unavoidable. An entropy extractor requires a small number of truly random bits in order to provide even one bit of near-random output [89].

One could use a composition of an entropy extractor and a pseudorandom generator to construct a new pseudorandom generator which operates using weak randomness. Through this technique the entropy extractor will “condense” the randomness from the weakly random source, then provide its output as the source randomness expected by the pseudorandom generator. For APPRNG we directly adapt the algorithm so that it will improve the quality of pseudorandom output, thereby serving the role of this composition. This method does not serve as a stand-alone entropy extractor.

It is known that pseudorandom generators can serve as entropy extractors under certain assumptions relating to the size of circuits computing the function with that of its inverse [90]. Determining whether APPRNG satisfies this assumption is left to future research.

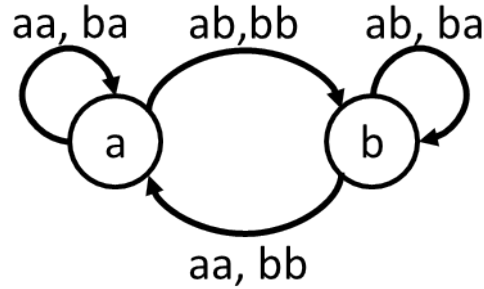


Figure 4.10: An example of a 2-state automaton that is 2-strided.

4.7.2 Min-Entropy

Intuitively, less randomness contained in a source necessarily reduces the amount of entropy that can be extracted from that source. This means that the weaker a random source is, the less efficient APPRNG must be in producing pseudorandom output from this source. For this reason we need a means of expressing the quality of a pseudorandom source. The metric settled upon is that of min-entropy.

The min-entropy of an n bit distribution X is the largest value p such that for every $x \in \{0, 1\}^n$ we have that $P[X = x] < 2^{-p}$. We say that X is an (n, p) -distribution. In other words, the min-entropy of a distribution is proportional to the log of the inverse of the probability of the most likely event. The smaller the min-entropy, the weaker the distribution. For example, if we have a uniform distribution over 4 bits, then the probability of any particular bit string being sampled would be $\frac{1}{2^4}$, thus the min-entropy would be 4, and this would be a $(4, 4)$ distribution. If, however, the distribution was strongly biased such that there was a particular bit string that was sampled with probability $\frac{1}{2}$, then the min-entropy would be 1, and this would be a $(4, 1)$ distribution.

4.7.3 Striding APPRNG

In order to use APPRNG over weakly random input we transform the input distribution to one with a higher min-entropy. We use a technique called striding, whereby an automaton transitions simultaneously on multiple characters at once rather than on a single character. For example, if an automaton is 3-strided then it will consume 3 input characters per transition. An example strided automaton is shown in Figure 4.10.

Consider that we have an (n, p) distribution. If we convert the machines to k -stride its input then the input distribution to the automata is $(n * k, p^k)$. Let's assume the the most probable bitstring from the input distribution is 0^n , which occurs with probability $\frac{1}{2^p}$. This implies that the

most probable bitstring in the k -strided distribution must be $0^{k \cdot n}$, which occurs with probability $\frac{1}{2^p} \cdot \frac{1}{2^p} \cdots \frac{1}{2^p} = \frac{1}{2^{kp}}$. Since the min-entropy of the distribution increases exponentially with the stride length, the min-entropy of the distribution can be made arbitrarily close to maximum (those arbitrarily close to truly random) for any distribution with min-entropy greater than 0.

In order to accommodate the strided input we must appropriately configure the parallel automata. To do this we will use a source of truly random input to construct the machines, while using the weakly random source to derive the machines' input. A machine defined over a b -bit alphabet that is k -strided effectively transitions over an alphabet of $b \cdot k$ bits. This means that a k -strided configuration of $G_{M,s,r}$ will need $M \cdot s \cdot \log\left(\frac{s^k!}{s^{k-1}!}\right) = \Theta(M \cdot s^{k+1} \log s)$ bits of true randomness (Theorem 4.7.1 and $r \cdot k \log s$ bits of weak randomness, while producing $r \log s$ bits of output.

Theorem 4.7.1. *a k -strided configuration of $G_{M,s,r}$ will need $M \cdot s \cdot (\log s^k! - s \log s^{k-1}!) = \Theta(M \cdot s^{k+1} \log s)$ bits of true randomness*

Proof. To construct the k -strided parallel automata for APPRNG, a random balanced s -partition of the symbol set Σ^k , where $|\Sigma| = s$ uniquely defines each state's outgoing transitions. This procedure thus acts independently on each state in each machine. To count the number of random bits needed to build all machines, we must simply count the number of random bits needed for each state, then multiply by the total number of states across all machines, $M \cdot s$.

The number of balanced s -partitions of a set of size s^k is given by:

$$\frac{s^k!}{(s^{k-1}!)^s}$$

To construct these partitions we begin by permuting our alphabet of size s^k , which has $s^k!$ possibilities. Every block of $\frac{s^k}{s}$ in the permutation then results in one of the balanced s partitions. Each of these partitions is a set, i.e. order doesn't matter, we must "unpermute" each of the blocks of size $\frac{s^k}{s}$ to undo the double counting of blocks containing the same set of elements appearing in different orders in the original permutation.

Each of the blocks is of size $\frac{s^k}{s}$, thus there are $\frac{s^k!}{s}$ ways to permute each block. To count exactly the number of choices for s balanced permutations we must divide the number of permutations of the alphabet, $s^k!$, by $\frac{s^k!}{s} = s^{k-1}!$ for block. This gives the final count of $\frac{s^k!}{(s^{k-1}!)^s}$.

The number of bits required to select one choice of the s partitions at random is given by

$$\begin{aligned}
\log \frac{s^k!}{(s^{k-1}!)^s} &= \log s^k! - \log (s^{k-1}!)^s \\
&= \log s^k! - s \log s^{k-1}! \\
&\in \Theta(s^k \log s^k - s^k \log s^{k-1}) \\
&= \Theta(s^k (\log s^k - \log s^{k-1})) \\
&= \Theta(s^k \log s(k - (k - 1))) \\
&= \Theta(s^k \log s)
\end{aligned}$$

So for all $M \cdot s$ states across all M machines, we need $M \cdot s(\log s^k! - s \log s^{k-1}!) \in \Theta(M \cdot s^{k+1} \log s)$ random bits. \square

The amount of additional randomness needed for striding increases exponentially in terms of the stride length. This may initially seem intractable, but recall from above that the stride needs to be logarithmic in the min-entropy of the distribution. This means that in order to compensate for weak randomness, the amount of additional true randomness needed is linear in the min-entropy of the weakly random distribution.

4.7.4 Experimental Results

Here we demonstrate the capability of the striding technique to allow APPRNG to effectively utilize weak randomness, but still pass a battery of statistical tests. For this example we used the Small Crush test suite. Due to the APPRNG algorithm being emulated on a standard CPU, rather than hardware that can take full advantage of its parallelism, we were constrained from using the more extensive Crush and Big Crush tests for this study as for parameters needed for striding the time and memory constraints on the machine were insurmountable.

There are three principle models for weakly random sources [91] of min-entropy k :

- **Markov Chain Source:** models weak randomness using a random walk of length n over a biased Markov Chain, where each transition p_{ij} is taken with probability $k/n < p_{ij} < 1 - k/n$ [92].
- **Unpredictable Source:** models weak randomness where, $1 \leq i \leq n$, and $b_1, \dots, b_i \in \{0, 1\}$, $k/n \leq \Pr[X_i = 1 \mid X_1 = b_1, \dots, X_{i-1} = b_{i-1}] \leq 1 - k/n$. This means that given i bits the $i + 1^{\text{th}}$ bit is only somewhat unpredictable [93].

- **Bit Fixing Source:** models weak randomness by taking a distribution over n bits where $n - k$ bits are fixed, and the remaining bits are generated randomly [94].

Of these three models we only used the Bit Fixing model for our experimentation. The Unpredictable Source model was developed as an extremely general model of weak randomness. In this model there is an adversary who is allowed to select bits somewhat at random. One could think of this model being realized by an adversarial actor who, whenever a random bit is requested, may maliciously select from a set of unfair coins each having its own bias known to the adversary. We found this model unimplementable for our experimentation. The Markov Chain Source model is a generalization of the Bit Fixing model, and allows for too many degrees of freedom to provide for a compelling experimental narrative.

We found the Bit Fixing model best suited for our experiments due to its ease of implementation, as well as its reliance on only a single parameter to change the degree of weakness of the random source. For our experiments the automata were constructed with truly random input (actually pseudorandom input from Philox), while the input string was sampled from our Bit Fixing source. This source took in a singular parameter t . Every t^{th} b -bit symbol in the input string was fixed to be 0^b , while all remaining bits were truly random (again, pseudorandom input from Philox).

We begin with a configuration of APPRNG which passes all tests. From here we weaken the random input until at least one test fails. Next we increase the stride of the machines and demonstrate that APPRNG passes all the statistical tests even when receiving the same weakly random input. We then repeat this process of weakening until failure, increasing stride, then passing all tests through one more cycle. The APPRNG configurations used are shown in Table 4.6.

Tests were run on the Small Crush battery of statistical tests. APPRNG was configured to run 571 machines in parallel, each with 8 states, transitioning over a 6-bit alphabet, for 550,000 symbols. Stride refers to the number of input symbols consumed per automaton transition. Fixing period refers to the distance between fixed symbols (the rest of the symbols are chosen randomly), where ∞ means that no bits were fixed. Tests Failed refers to the number of tests from the Small Crush battery which identify the output of APPRNG as being non-random.

Table 4.7 shows the minimum fixing period which requires stride of 1, 2, 3 in order to pass all Small-Crush tests. Any fixing period smaller than the value in the left-hand column requires a stride greater than that in the right hand column for APPRNG configured to run 571 machines in parallel, each with 8 states, transitioning over a 6-bit alphabet, for 550,000 symbols. Note that striding automata has dramatic effect in improving APPRNG's tolerance to weakly random input.

| Stride | Fixing Period | Tests Failed |
|--------|---------------|--------------|
| 1 | ∞ | 0 |
| 1 | 10 | 5 |
| 2 | 10 | 0 |
| 2 | 2 | 5 |
| 3 | 2 | 0 |

Table 4.6: Striding APPRNG mitigating weakly random input for 571 parallel 8-state automata over 550,000 6-bit inputs.

| Fixing Period | Stride |
|---------------|--------|
| 10,000 | 1 |
| 6 | 2 |
| 2 | 3 |

Table 4.7: Minimum Stride needed by Fixing Period in for 571 parallel 8-state automata over 550,000 6-bit inputs to pass all Small Crush tests.

We were not able to discover a non-trivial 3-strided configuration of APPRNG which failed the Small Crush tests within the parameters of our testing environment.

4.8 Automata-based Bloom Filtering

APPRNG’s quality pseudorandom output makes it not only useful as a high-throughput pseudorandom number generator, but also implies its usefulness as an algorithm derandomization technique. If one has any application which seeks simulation of many Markov chains, one could use APPRNG to accelerate this application by parallelizing the Markov chain simulation, and reduce the amount of randomness required by sharing the input randomness across the many Markov chains. Without APPRNG’s output having a pseudorandom property, using APPRNG may jeopardize the derandomized algorithm’s consistency. Here we provide some such example application which would not be possible without the pseudorandom property of APPRNG: automata-based Bloom filters.

4.8.1 Bloom Filters

A Bloom filter is a probabilistic set-membership data structure first discovered by Burton Bloom in 1970 [95]. A set-membership data structure is built from an input set S , and for a query item x determines whether $x \in S$. More standard instances of the data structures include binary search trees, and hash tables. Binary search trees have logarithmic query time in terms of $|S|$, and space roughly equal to $|S|$. Hash tables have expected constant time access, with worst case typically either logarithmic or linear depending on the collision resolution strategy, but this expected case performance requires the hash table to be much larger than $|S|$.

Bloom filters allow for worst case constant time set membership queries using space not much larger than S . To achieve this impressive performance, Bloom filters allow for a small probability of giving false positives, so with small probability the Bloom filter will incorrectly conclude $x \in S$. This is similar to hash tables in that both exhibit probabilistic behavior, however hash tables “gamble” with run time (this is called a Las Vegas algorithm), while Bloom filters gamble with accuracy (this is called a Monte Carlo algorithm). In this case Bloom filters have a small probability of false positives, but false negatives are impossible, thus Bloom filters are a one-sided Monte Carlo algorithm.

In order to further illustrate these concepts, consider the following analogy. Let’s say we wish to determine a person’s identity by examining their set of Facebook friends. We might conclude that two individuals are the same person if and only if they have the exact same set of friends on Facebook. We could then probabilistically determine whether a mystery person is among your Facebook friends by (1) taking the union of all people who are friends of your friends, then (2) checking whether or not all friends of the mystery person are among your friends’ friends. If the mystery person has any friend who is not among your friends’ friends, then that person is certainly not your own friend. On the other hand, if all of the mystery person’s friends appear among your friends’ friends, then that mystery person is likely to be your friend. A bloom filter could determine this situation by doing a “roll call”, where it broadcasts each person’s name from the mystery person’s friend list to everyone in your friends’ friends list. If all members in the roll call are present, then that person is likely to be your friend. Conversely, if any person is missing from that set, that person is definitely not your friend. Thus by broadcasting the names, the filter must wait only a constant amount of time for a response.

To construct a Bloom filter, we first define a bit vector v of m 0s, and a list $H = [h_1, \dots, h_k]$ where each $h_i : \{0,1\}^* \rightarrow \{x \in \mathbb{N} \mid 0 \leq x < m\}$ is a hash function. Each hash function takes an arbitrary item as input, and returns an index in v as output. The vector v is configured for queries on set S by taking a hash of every element in S and setting each resulting index of v to be 1. Formally:

$$v[i] = 1 \Leftrightarrow \exists s \in S \exists h_j \in H \ni h_j(s) = i$$

Then for a query on an element x , if x indeed belongs to the set S it must hold that $v[h_j(x)] = 1 \forall h_j \in H$, since otherwise we would have set to 1 any index that was mapped to by one of the hash functions on input x . In the case that $x \notin S$ it may still be the case that all hashed indices happen to be 1, as x could be mapped to a set of indices belonging to the union of indices hashed

to by multiple items from S , i.e. $\exists S' \subseteq S \ni [\{h_j(x) \mid h_j \in H\} \subseteq \{h_j(s) \mid h_j \in H \wedge s \in S'\}]$. For example, assume that $s_1, s_2 \in S$ hash to indices $\{8, 12, 15, 22\}$ and $\{4, 9, 19, 20\}$, respectively, and $x \notin S$ hashes to $\{8, 15, 19, 22\}$. In this case, all the indices that x maps onto would be set to 1 in the bit vector v , so the Bloom filter will erroneously claim that x belongs to S , thus giving a false positive.

The strength of Bloom filtering comes from the seemingly random nature of the hash functions. Each hash function serves as a deterministic approximation of an ideal truly random function, which would take an input string and map it to a randomly selected output from its range in a self-consistent manner (meaning if the function is given the same input multiple times it will always provide the same output). This means that a Bloom filter's hash functions seemingly select a set of indices uniformly at random across the vector v .

If we assume that the hash functions successfully model random functions, we can bound the probability of a false positive error in the Bloom filter. This error depends on the number of bits (m) allocated for the bit vector (v), the number of hash functions (k) in the set of hashes (H), and the number of elements (n) in the query set (S). The probability of a false positive is given by [96]:

$$(1 - e^{-\frac{km}{n}})^k$$

4.8.2 Automata-based Bloom filters

Intuitively, traditional Bloom filters achieve their performance by using hashes to probabilistically “disperse” each element's identity across its bit vector, allowing for a more dense representation of a set of elements through *combinations* of enabled bits. One could say that each hashed-to bit is a feature of that element's identity. We model this behavior by associating each bit in the Bloom filter's vector using a finite state automaton, where that automaton probabilistically “distinguishes” whether it identifies with a given query element. In the same way that Bloom filters accept a string if and only if all aspects of its identity (hashed-to bits) had been seen by some element(s) from S , the automata-based filter will accept if and only if all automata which identify with the query element had also identified with an element of S .

Let's compare these techniques using the Facebook example above. Traditional Bloom filtering requires some central actor to inspect the mystery person's friend list, then verify that each individual belongs to your friends' friends list through a roll call. The automata-based approach instead operates by this central actor broadcasting the name of the mystery person to all people not in your friends' friends list. If any person is friends with that mystery person, then the mystery

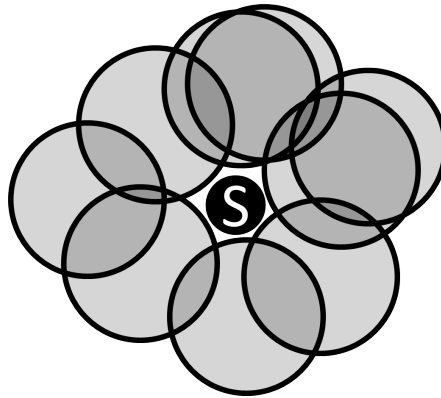


Figure 4.11: An illustration of how automata-based bloom filtering decreases false-positive rates via increased numbers of automata. Each gray circle represents the set of elements accepted by an automaton, with all automata rejecting every element in the set S . A query on any string that falls outside of all the gray circles will result in the response $x \in S$. The more automata running against x , the smaller the probability that x falls in neither the circle for the set S nor any of the gray circles (which would be a false positive).

person is not your friend (as otherwise all of the mystery person's friends would be excluded).

High-level Construction

To build automata-based bloom filters we begin with a list containing many automata, intuitively each automaton representing a bit in the bit vector of the Bloom filter. To emulate the Bloom filter's property of accepting a query element if and only if that element mapped exclusively to indices also mapped to by some element of the set S , we will accept a particular query element if and only if all automata which accept that query element also accepted at least one element of S . False negatives are impossible, as each automaton is deterministic, thus its answers are self-consistent.

Note that the decision of $x \in S$ for automata-based Bloom filtering depends only on the behavior of machines which reject all elements from S . This means that any automaton that accepts an element from S is irrelevant, and can be discarded. The probability of a false positive is determined by p^A , where p is the probability that any particular automaton accepts a random input string and A is the number of automata remaining in our set. We now arrive at a simple core idea of automata-based Bloom filtering: with high probability we determine whether $x \in S$ by testing it on many non-examples of automata which accept strings in S . Since we have many automata which all reject every element in the set S , if we have sufficiently many such automata we can conclude any string rejected by all of them is likely to be in S . A high-level illustration of this phenomenon is shown in Figure 4.11.

We now provide an algorithm for constructing a set of automata which perform set membership queries such that the probability of a false positive is exactly that of a given Bloom filter with a bit

| | To q_s | To q_a |
|------------|----------|----------|
| From q_s | $1 - p$ | p |
| From q_a | $1 - p$ | p |

Table 4.8: Stochastic Transition Matrix of each Markov Chan used for Automata-Based Bloom Filtering

vector v of m bits, a list $H = [h_1, \dots, h_k]$ of k hash functions, and a set S with n elements. We break our discussion into 3 parts: (1) initializing our choice of automata, (2) inserting each element of S into our filter, (3) performing queries over the data structure.

Initialization

To build the filter we first randomly construct a large set of M automata (derivation of this choice of M is done in Section 4.8.2), each of which will accept a random input string with probability p . Each automaton will have 2 states, call them $\{q_s, q_a\}$, where q_s is the start state and q_a is the only accepting state. We construct each automaton by selecting two random sets of $p \cdot |\Sigma|$ symbols to transition from state q_s to q_a and from state q_a back to q_a . The symbols not in each set will define the transitions from q_s to q_s and q_a to q_s respectively.

When each automaton receives a random input string x , the long-term proportion of time the automaton spends in state q_a is exactly p . This holds because it is equivalent to the Markov Chain in with the stochastic transition matrix shown in Table 4.8, whose stationary distribution (which converges after a single transition) shows the proportion of time spent in state q_a is exactly p . This means that at the end of the input the probability the machine halts in state q_a , and therefore accepts, is also p . All that remains to define each automaton is the selection of p . We wish to emulate a Bloom filter with m bits and k hashes, thus the probability that each automaton rejects should be equal to the probability that a fixed index in v is not among the results of the k hashes.

Since there are m bits in v , the probability that the output of a particular hash function is not index i is given by $1 - \frac{1}{m}$, as each hash function should return each index with equal probability. This means that the probability that none of the hash functions returns i is $(1 - \frac{1}{m})^k$, which should also be the probability that our machines reject. We therefore select the probability of acceptance for automata-based Bloom filters as:

$$p = 1 - (1 - \frac{1}{m})^k$$

Note that this construction requires the the choice of alphabet Σ for the transitions must exactly match the alphabet of the inputs. Also, the minimum transition probability for each automaton

is $\frac{1}{|\Sigma|}$. In order to have an automaton with a smaller transition probability than $\frac{1}{|\Sigma|}$, or a larger alphabet than the input string, then input string should be strided, thereby squaring the size of the alphabet.

Insertion

Now that we have set up the automata for the Bloom filter, we must insert all elements from the set S . Traditional Bloom filters execute the k hashes on each element of S , and set the automata of each resulting index to be 1. For automata-based Bloom filtering we run each input of S on the entire set of M automata, and remove any automata which accept.

Since each automaton accepts a input string with probability p , the probability that an automaton is removed as a result of an insertion is p (the probability that automaton accepted), so it has probability $1 - p$ of remaining. The probability that an automaton remains after $n = |S|$ insertions is therefore $(1 - p)^n$. This implies that the number of automata which remain, call this R , after inserting all of S is:

$$\begin{aligned} R &= M(1 - p)^n \\ &= M(1 - (1 - (1 - \frac{1}{m})^k))^n \\ &= M(1 - \frac{1}{m})^{kn} \\ &\approx Me^{\frac{-kn}{m}} \end{aligned}$$

While M is the number of automata we will construct when initializing the Bloom Filter, R will be the number of automata which we must execute for each query. This implies that as k and n grow larger for fixed m , automata-based Bloom filters require progressively fewer automata. For fixed space, as k and n increase, we can therefore emulate a Bloom filter with more bits (larger m).

Querying

To query whether an element x belongs to the set S we run x as the input to all R automata. If any of those automata accept then we can definitely conclude that $x \notin S$, as otherwise that automaton would have been removed during the insertion. If all automata reject then we conclude that x likely belongs to the set S . The probability that this answer was a false positive is given by the probability that all automata accept for a random input string. Since each automaton rejects with probability $(1 - \frac{1}{m})^k$ and we run x over R automata, the probability of a false positive is:

$$\begin{aligned}
& \left(1 - \frac{1}{m}\right)^{kR} \\
&= \left(1 - \frac{1}{m}\right)^{kR} \\
&\approx e^{-\frac{kR}{m}} \\
&\approx e^{-\frac{k}{m}} M e^{-\frac{kn}{m}}
\end{aligned}$$

In the case of traditional Bloom filters a false positive occurs whenever x hashes only to 1s in the vector v . The probability that an arbitrary index in v is a 1 is $(1 - (1 - \frac{1}{m})^{kn})$. This means that the probability that all k hashes map to a 1 for query element x (and therefore the probability of a false positive) is:

$$\begin{aligned}
& \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \\
&\approx \left(1 - e^{-\frac{kn}{m}}\right)^k
\end{aligned}$$

Number of Automata Required

We now finally derive the choice of M so that the probability of false positives for automata-based Bloom filters matches that for traditional Bloom filters. To do this, we must simply solve for M :

$$\begin{aligned}
e^{-\frac{k}{m}} M e^{-\frac{kn}{m}} &= \left(1 - e^{-\frac{kn}{m}}\right)^k \\
M \frac{-k}{m} e^{-\frac{kn}{m}} &= k \ln\left(1 - e^{-\frac{kn}{m}}\right) \\
M &= -m \cdot e^{\frac{kn}{m}} \ln\left(1 - e^{-\frac{kn}{m}}\right)
\end{aligned}$$

This ugly equation gives little intuition on the relative size of M . The equation becomes much simpler, and the story much clearer, when we assume k takes its optimal value to minimize the probability of false positives for m and n , given by $k = \frac{m}{n} \ln 2$

$$\begin{aligned}
M &= -m \cdot e^{\frac{kn}{m}} \ln(1 - e^{\frac{-kn}{m}}) \\
&= -m \cdot e^{(\frac{m}{n} \ln 2) \frac{n}{m}} \ln(1 - e^{(\frac{m}{n} \ln 2) \frac{-n}{m}}) \\
&= -m \cdot e^{\ln 2} \ln(1 - e^{-\ln 2}) \\
&= -m \cdot 2 \ln \frac{1}{2} \\
&= (2 \ln 2)m \\
&\approx 1.386m
\end{aligned}$$

This implies that automata-based Bloom filters must create about 1.386 automata for each bit in the bloom filter for an optimal choice of hash function count. The final expected space complexity of automata-based bloom filters is R , final number of automata remaining after the insertions. We derive the value of R for the optimal choice of $k = \frac{m}{n} \ln 2$ and $M = (2 \ln 2)m$:

$$\begin{aligned}
R &= Me^{\frac{-kn}{m}} \\
&= (2 \ln 2)m \cdot e^{(\ln 2) \frac{m}{n} \frac{-n}{m}} \\
&= (2 \ln 2)m \cdot e^{\ln \frac{1}{2}} \\
&= (\ln 2)m \\
&\approx 0.693m
\end{aligned}$$

This means that per bit of the given Bloom filter, we expect to simulate 0.693 automata and achieve the same false positive rate.

Experimental Verification

We implemented the above described Bloom filtering algorithm in Python. We tested for false positive rates over 1000 queries, varying independently:

- Size of Bloom bit vector (ranged from 100-500)
- Size of inserted set (ranged from 5% to 50% of the size of the Bloom bit vector)

Each test was additionally repeated with 10 repetitions. The set of elements inserted into the Bloom filter (the inserted set above), as well as the set of elements to query were chosen uniformly at random for each trial. Across all tests, the theoretically-approximated number of false positives

was 98 on average per trial, our automata-based bloom filtering implementation observed and average of 84 false positives per trial (87.5% of expected).

Special Considerations

The above analysis shows a configuration for automata-based Bloom filters which match the statistically *expected* performance of traditional Bloom filters. The behavior of Automata-based Bloom filtering will not follow exactly the same distribution, as illustrated by the following examples:

- Each query on a traditional Bloom Filter hashes to *exactly* k bits. On the other hand, automata-based Bloom filters “hash” to k bits *on average*.
- Both Bloom filtering techniques behave probabilistically relative to a random query and a random inserted set. Automata-based Bloom filtering should see greater variance in its performance relative to the inserted set. In an extreme example, it is possible that inserting all elements in the set S into the automata-based Bloom filter results in the elimination of all automata. Should this happen, every query will be a false positive. This situation can only happen in traditional Bloom Filters in the case of a very large inserted set. In order to maintain the same expected performance, the preponderance of cases in which automata-based Bloom filtering outperforms traditional Bloom filtering must compensate for these adverse outliers.
- Automata-based Bloom filters require much more upfront-cost for configuration. Our construction relies on random generation of many automata, requiring a large quantity of random input. Traditional Bloom filters only require allocation of memory and knowledge of two hash functions [97]. It is conceivable that a set of well-behaved automata could be deterministically generated in order to reduce the initialization cost. This consideration is left to future research.
- Automata-based Bloom filters rely on the assumption that APPRNG is a true pseudorandom number generator. For sufficiently large query items, the distribution of the automata acceptances will deviate from the random behavior implicit in our analysis. In the case that APPRNG is a pseudorandom generator, “sufficient largeness” will be considerably larger than in the case when APPRNG is not a pseudorandom generator. We hypothesize that APPRNG, being a true pseudorandom number generator, would allow for exponentially larger inputs before the random behavior suffers, since the definition of pseudorandomness requires that any distinguisher must run in superpolynomial time.

- This automata-based Bloom filter design is not restricted in the same way Bloom filters are. The false positive probability is given by the probability that each state rejects, raised to the power of the number of states. We can therefore reduce the false positive probability beyond the capabilities of traditional Bloom filters by either decreasing the probability of rejection, or by increasing the number of machines.

4.9 Summary

In this work we explored the usage of parallel Markov chain simulation via random finite state automata running on shared input to produce pseudorandom behavior. We briefly mentioned how Markov chains could be simulated by such automata. This implies that automata-based accelerators can efficiently simulate Markov chains through parallel execution on shared input, but only with our demonstration that this produces pseudorandom behavior.

We justified the pseudorandom behavior of our approach by providing a hardness assumption under which the algorithm is a cryptographically secure pseudorandom number generator, as well as showing its capability of passing even the most stringent of tests for pseudorandom behavior. We justified its efficiency by showing that the algorithm belongs to the same complexity class as the fastest known cryptographically secure pseudorandom number generators. Toward its use in practice, we showed its efficiency by modeling its behavior on future automata-based architectures, which shows its capability of outpacing the highest-throughput GPU-based pseudorandom generators at 40.5GB/s of throughput, while consuming $6.8\times$ less energy.

Pseudorandom number generators require truly-random input to operate, and such input is difficult to obtain. To mitigate this issue we provide a method for altering the APPRNG algorithm to be more tolerant to poorly-random input.

Finally, we show that our demonstration of the pseudorandom behavior displayed by parallel automata-based simulation of Markov chains opens up a breadth of new problem solving techniques. We present automata-based Bloom filtering as a first example of a solution which relies on this pseudorandom behavior in order to be feasible. It is able to emulate a traditional Bloom filter with the same false positive rate, while running in constant time by number of hashes, and using 0.7 automata with two states each per bit of the Bloom filter's vector. We hope our results open up Markov chain based probabilistic computing as an exciting new area of research.

Chapter 5

Conclusions and Future Directions

In this dissertation we proposed two new highly-scalable approaches to effectively process massive data sets in the post- Moore’s Law era, namely:

1. algorithms that operate directly on highly compressed data; and
2. leveraging massively parallel finite automata-based architectures for addressing specific problem domains.

We showed that the compression-aware algorithmic approach can extend scalability by exploiting regularity in highly-compressible data, while also avoiding the expensive decompression and re-compression steps. Additionally, we utilized the automata processor’s ability to succinctly implement complex computations via simulation of non-deterministic finite-state automata. We evaluated the efficiency, extensibility, and generality of these non-traditional approaches on big data. We presented promising experimental results and theoretical impossibility arguments, as summarized in this chapter below. We conclude this section with future research directions.

Our contributions serve as the first steps in enabling the design of more sophisticated and nuanced algorithms for grappling with scaling issues in processing and mining large datasets. Since the rate of growth of these massive datasets exceeds the pace of Moore’s Law (even before its recently-revised forecasted slowdown [3]), application-side software resourcefulness is becoming necessary in order to satisfy the growing demands for computing resources. The algorithms, codes, compression schemes, and theories presented in this dissertation will hopefully help usher in future efficient and practical algorithms, and have the potential to fundamentally change the way in which society collects, processes, utilizes, and mines large datasets, and thereby provide a much-needed life-extension to Moore’s Law.

5.1 Compression-Aware Algorithms

5.1.1 Contributions

We began with a study of compression-aware techniques on a particular set of well-understood classical problems: sorting and statistics. Next, we showed some necessary assumptions for accelerating these algorithms. We demonstrated that not all compression techniques are amenable for acceleration of all algorithms, but instead proper design of algorithms for compressed data requires careful pairing of algorithms and compression schemes.

From these lessons, we designed various algorithms which are asymptotically more efficient in both computation time and memory use when running on compressed datasets. We demonstrated that our algorithms can be applied to existing compression schemes, as well as to new compression schemes that are specifically tailored to speed up the algorithms over the compressed data. We applied our methods to graphs and geometric domains, which suggests that these techniques are broadly applicable. While prior work relies on designing or modifying compression schemes to scalably solve specific problems, our results indicate that certain standard compression schemes are inherently compatible with many common algorithms.

5.1.2 Future Directions

We sought to define a framework for the development of new algorithms to operate directly on compressed data. To broaden the applicability of compression-aware techniques, we need to study more compression schemes and their amenability to various classes of algorithms. Our non-existence proof regarding sorting compressed lists provides a good rule-of-thumb for when a compression scheme could enable a more efficient compression-aware approach for a particular algorithm, namely when the data is more compressible over a distribution in which an algorithm sees better average-case performance. This rule provides a quick way to rule out particular applications, which should help accelerate future explorations of the compression-aware algorithms design space.

Some of the above results may also be suitable for the development of compression-aware data structures. For instance, can we build KD-Trees or Voronoi diagrams on set-of-lines compressed data? Can we use graph-based compression schemes to compress graph-like data structures over other data? And in general, which data structures are most effective in compression-aware algorithmic design?

Alternate Definitions of Compression

The goal of the compression-aware algorithms approach is to discover compression techniques which allow for fast-running algorithms without seriously compromising the compression ratio. One potential concern, however, is that solutions to certain problems may be “hidden” within the compressed data. In other words, the compression scheme may pre-compute some functions over the data during the compression phase (akin to a data *preprocessing* phase in traditional algorithms), and then store those pre-computed values alongside the compressed data, for later easy retrieval. This would be outside the spirit of our study, but in order to avoid this type of algorithmic “cheating”, we must devise a more precise definition of compression. For example, Feder and Motwani [33] present a working definition of a graph compression, as follows:

Let $\mathcal{G}(V, E)$ be a labeled graph with $|V| = n$ and $|E| = m$. A compression of \mathcal{G} is defined as a labeled graph $\mathcal{G}^*(V^*, E^*)$ such that:

1. $n^* = |V^*|$ is polynomial in n .
2. $m^* = |E^*|$ is significantly smaller than m , i.e. $m^* = o(m)$.
3. The mapping $*$: $\mathcal{G} \rightarrow \mathcal{G}^*$ is 1 – 1.

This definition is insufficient, however, for the purposes of designing compression-aware algorithms. The restriction that n^* be polynomial in n allows solutions to algorithms of size polynomial in n to be “hidden” within the compression scheme. For this reason we might seek to restrict the schemes addressed to those which asymptotically approach entropy. This property has been proven to hold for compression schemes in different domains (such as strings [12, 13]). With this restriction, only constant space solutions (decision problems) may be hidden in the compression scheme without violating our definition, so we avoid these. To the best of our knowledge, industry has not seen widespread adoption of any graph or point set compression scheme that satisfies these requirements.

Not yet having an entropy-rate compression scheme or precise “non-cheating” restrictions on the contents of compressed data, an honest-effort approach will have to suffice for now. Indeed, this is the standard that we currently uphold with our presented schemes. None of the compression schemes used in this dissertation compute or store any hidden solutions to the problems / algorithms that we present, nor do they utilize metadata which assists computation.

Thus, hiding precomputed solutions within compressed data will sacrifice compression ratio, and may also obscure the potential benefits of compression-aware computations, which ideally

would not require such metadata to achieve dramatic accelerations. We hope that these observations will initiate discussions regarding how to formalize such restrictions on algorithm-aware compressions in ways that will leverage their full potential and super-scalability.

Using Imprecise Hardware on Lossy-compressed data

In Section 2.5.1 we discussed a lossy algorithmically-aware compression scheme for geometric data. Lossy compression schemes may compress two different raw-data inputs into the same compressed value, thus creating potential uncertainty when decompressing. We could leverage the tolerance to lossyness of certain applications, in order to achieve further gains in performance and power efficiency, by running the algorithm on imprecise hardware [98].

Traditional hardware design requires all computations to have strict accuracy guarantees. However, many applications (e.g. video and audio) do not require extreme precision and are inherently fairly fault tolerant. Thus relaxing the hardware's perfect precision constraints can enable potential time and power efficiency gains. In other words, rather than insist that all of the potential imprecision should reside in the high-level implementation (i.e. compression lossyness), we could instead choose to spread the imprecision across both the hardware and the software in fault-tolerant applications.

In spite of the uncertainty resulting from lossy-compression and execution over imprecise hardware, we can still be highly effective in performing complex computations. Indeed, a natural example of this "strategy" is the human brain itself. Our brains are computers based on imprecise, unreliable neurological components and connections, that can nevertheless perform highly sophisticated (to the point of being inimitable) computations. Evolution clearly managed to obtain impressive computational power from imprecise components (i.e., neurons), while allowing significant and obvious deviations from absolute correctness. Perhaps these billions of years of trial-and-error provide deep lessons for us scientists and engineers, as we contemplate future computer designs.

Natural applications which can benefit from lossy-compressed computation using imprecise hardware include those that are limited by human perception (e.g., multimedia), or contain a significant amount of noise in the system (e.g., classification, detection, and tracking in sensor networks, or image/video processing and compression [99]). Additional potential beneficiaries include inherently randomized or stochastic algorithmic approaches, such as simulated annealing, genetic algorithms, neural networks, fuzzy systems, Monte-Carlo approaches, artificial intelligence problems, numerous approximation schemes for computationally intractable problems, and those modelled by parallel markov chains (e.g. automata-based bloom filtering of Section 4.8).

5.2 Automata Computing

In this work we have shown that the automata computing paradigm has surprising potential, as well as surprising limitations. We characterized the class of problems which can be implemented on automata-based coprocessors in the model of the Micron AP. We also provided applications with surprising properties not previously displayed by automata-based designs (e.g. APPRNG and Bloom filtering), broadening our intuitive understanding of possible applications.

5.2.1 Contributions

Complexity-Theoretic Analysis of Automata Processing

We began with a proof that, under proper assumptions regarding computing resources availability, the automata processor can recognize no more than the regular languages. This argument helped dispel a persistent misconception regarding the Turing-completeness of the AP. We do not claim that the proof of Turing completeness of the AP via simulation of cellular automata is incorrect. Rather, we argued that this only reinforces an incomplete perspective of automata hardware.

There is an ancient Indian parable about three blind men encountering an elephant for the first time. Being blind, they are each unable to appreciate the full size and form of the elephant, as they have no way to study it other than by touch. The first blind man concludes that elephants are long and slender, much like a snake. The second man concludes that elephants are sturdy and tall, much like a tree. The third man stated that elephants are smooth and hard, much like a spear. These three observations at first seem completely contradictory, until you realize that the first man is touching the elephant's trunk, the second its leg, and the third its tusk. All of them experienced the elephant differently, and each of their experiences was incomplete, albeit not necessarily incorrect from each blind person's narrow perspective.

As researchers trying to study the fundamental characteristics of our computing artifacts, we must realize we are in some ways not unlike the blind men describing an elephant. Thus stating that the automata processor is Turing complete is correct as a statement of the behavior of the hardware when its resources are unbounded. Perhaps in the future, should Moore's Law continue long enough, this may indeed be the most useful characteristic of the AP hardware. This would likely include several orders of magnitude more states and Boolean gates in the hardware (approximately billions).

Meanwhile, our result that the AP computes only regular languages is not only also correct, but is better-suited for designers in some scenarios, as it better matches the design constraints of

potential applications with the limitations of the hardware. Such observations may thus avoid some unintended consequences in designing automata-based applications. For example, a developer working under the impression that the AP is Turing complete might misguidedly feel more empowered to pursue an application which requires far too many resources than what is practically available on the AP.

AP-Based Pseudorandom Number Generation

We showed a new approach to pseudorandom number generation, as well as a novel application for automata processing. Typical pseudorandom number generators operate by updating some amount of centralized state (called their seed) in a hard-to-predict way. By using the MISD (multiple instruction, single data) parallelism enabled by automata computing we instead distribute that state across many automata, thereby also distributing the work required to update that state. As a result, we are able to produce pseudorandom numbers with efficiency comparable to the most efficient pseudorandom number generators available, both in a theoretical sense as well as in a practical sense. To show theoretical efficiency, we demonstrated that APPRNG belongs to complexity class NC^0 . Toward practical efficiency, our experiments show comparable efficiency to the Philox algorithm with minor modifications to the AP hardware.

APPRNG is also very adaptable to many use cases. It allows for an easy trade off between performance and pseudorandomness by adjusting the length of the reconfiguration threshold. We also presented a way of adapting APPRNG to be better suited for weakly random input.

Finally in this chapter we gave a second novel application for automata processing—Bloom filters. We find this result interesting because it takes a model of computation conceived to study what can be done without the use of memory (finite state automata), and uses it as a form of memory. This technique toward Bloom filtering is able to at least match the probabilistic performance of traditional Bloom filtering, while accelerating queries and insertions to run in constant time by the number of hash functions.

5.2.2 Future Directions

Complexity-Theoretic Analysis of Automata Processing

While our result showing that the AP only computes regular languages may seem pessimistic, we also provide gems of optimism for how automata-based architectures may be designed in order to broaden their class of accepted languages. For example, the construction for converting counters to states and gates demonstrates that the finiteness of the thresholding behavior fundamentally restricts the power of those components. Counter designs which allow for the capacity to vary

with input string size would absolutely increase the power of the machines underlying the AP architecture. It also shows that even though counters add nothing to computability of AP machines, their design is so efficient that applications which make proper use of them will find them indispensable.

The final result from this section compares automata models of computation to circuit models. With this emerging age of heterogeneous computing, improving upon this result requires more research in developing theory relating a vast range of computing models. With this, we will be more capable of categorizing problems by their computational complexity relative to these models. Eventually, this would provide a more clear guide to matching algorithms (or even subroutines) to their best-suited hardware, and thereby making optimal use of available computing resources.

AP-Based Pseudorandom Number Generation

Automata-based Bloom filtering serves as a novel application of APPRNG. This demonstrates that APPRNG, in addition to serving as a pseudorandom number generator, is the necessary first step allowing for many new applications based on the simulation of parallel Markov chains. Our Bloom filter implementation is only the second step. These initial results serve to motivate and guide future algorithm design based on these parallel Markov chains.

We hope to have commenced a new area of exploration for automata computing. Previous to our work, applications for automata-based computing were restricted to pattern matching or pattern discovery domains. By showing an intuitively original application of automata computing (a pattern obscuring one), we undo any predispositions that the automata computing paradigm is inherently restricted to pattern matching.

5.3 Automata-based Compression-Aware Algorithms

In this dissertation we demonstrated the practicality of using compression-aware algorithms for computing on compressed input, and separately we have shown the usefulness of designing problems in a way that takes advantage of automata-based accelerators. A next natural goal is to synergistically compound the benefits of these two strategies by designing compression-aware algorithms suitable for automata-based acceleration.

Many decompression algorithms are modelled by finite state automata, e.g. Huffman codes [100], Asymmetric Numerical Systems [101], and Antidictionaries [102]. An example of a Huffman tree (or equivalently, Huffman automaton) built on text with character frequencies shown in Table 5.1 is shown in Figure 5.1. In these cases, intuition suggests that compression-aware automata-based designs should be feasible, since they would be a composition of two algorithms which are

| Letter | a | b | c | d | e | f |
|-----------|----|-----|-----|-----|------|------|
| Frequency | 45 | 13 | 12 | 16 | 9 | 5 |
| Code | 0 | 101 | 100 | 111 | 1101 | 1100 |

Table 5.1: Letter frequency and corresponding Huffman code for the Huffman Tree shown in Figure 5.1.

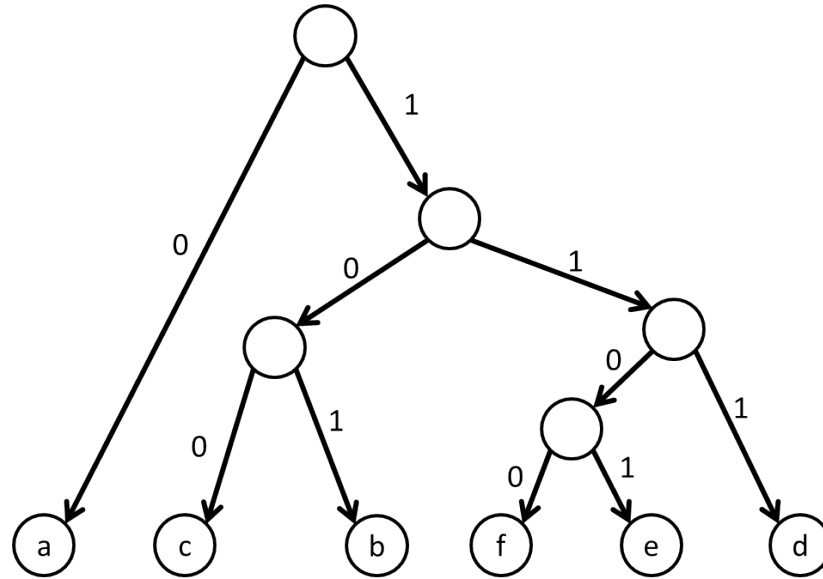


Figure 5.1: Huffman Tree for the letter frequencies shown in Table 5.1. This tree could also be viewed as a Moore machine, where once the computation reaches one of the leaves of the tree, it will output its label as the corresponding character.

both implementable as automata (an automata-based decompression algorithm and an automata-based classical algorithm). For this reason, a first step in compression-aware automata-based design should be to identify compression schemes whose decompression algorithms themselves can be implemented as automata.

Pattern matching over compressed data is a particularly well-studied problem (see [103] for a survey), and therefore constitutes another promising future direction. One early result from Moura et al. [104] gives a method for approximate pattern matching over Huffman-coded input. In this method the authors perform a generalized Huffman code in which codes are bytes rather than bits (i.e. the Huffman tree will have a branching factor of 128). With this modification, when patterns are compressed using the same Huffman tree as the input string, a standard exact pattern matching algorithm may be naively applied to match the compressed pattern on the compressed string. In the case of approximate pattern matching, doing this in a compression-aware way requires enumerating all possible patterns that are within a certain distance of the desired pattern. This could produce a very large number of patterns to match against. While this would result in a

substantial slowdown when run on a sequential architecture, automata-based accelerators may be the right tool for reigning in the large pattern set.

Ideally, the advantages offered by compression-aware algorithm design and automata-based accelerations should behave synergistically. One drawback of automata-based computing is that it is impossible to compute faster than linear time by the input size. However, if the input has been compressed before computation, the automaton will run in linear time relative to the compressed data, resulting in substantial acceleration.

Compression-aware approaches additionally gain benefit from each input bit having higher entropy (thus the algorithm gains efficiency as each bit contains more information, like each unpopped kernel of corn having higher density). When implemented as an automata-based algorithm, compression-awareness could result in a simplification of the automata, since some complexity has been offloaded onto the compression. In this case, the compression-aware approach would enhance the effectiveness of the automata-based algorithm by diminishing the computing resources necessary for computation, and at the same time reducing the amount of processing time required.

Should such super-scalable compression-aware automata-based algorithms be discovered, to paraphrase Mark Twain, rumors of the dire consequences of the death of Moore's Law would have been greatly exaggerated.

Bibliography

- [1] B. L. Robinson G. Robins and B. S. Sethi. On detecting spatial regularity in noisy images. *Information Processing Letters*, 69:189–195, 1999.
- [2] J. Hurwitz, A. Nugent, F. Halper, and M. Kaufman. *Big Data for Dummies*. John Wiley & Sons, Hoboken, NJ, USA, 2013.
- [3] T. Simonite. Intel puts the brakes on moore’s law. MIT Technology Review, 2016.
- [4] N. Brunelle, G. Robins, and a. shelat. Algorithms for compressed inputs. In *Data Compression Conference*, page 441, 2013.
- [5] N. Brunelle, G. Robins, and a. shelat. Compression-aware algorithms for massive datasets. In *Data Compression Conference*, page 478, 2015.
- [6] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.
- [7] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 99(Preliminary):1, 2014.
- [8] J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, and K. Skadron. Generating efficient and high-quality pseudo-random behavior on automata processors. In *International Conference on Computer Design*, pages 622–629, October 2016.
- [9] J. Wadden and N. Brunelle. System, method, and computer-readable medium for high throughput pseudo-random number generation. Patent Application no. US 2017/0083288 A1, March 2017.
- [10] S. M. Ross. *Introduction to Probability Models*, chapter 5, pages 312–339. Academic Press, 10th edition, 2010.
- [11] M. Charikar, E. Lehman, Ding Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, July 2005.
- [12] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
- [13] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions Information Theory*, 24(5):530–536, September 1978.
- [14] A. Amir, G. M. Landau, and D. Sokol. Inplace 2d matching in compressed images. In *SIAM-ACM Symposium on Discrete Algorithms*, pages 853–862, 2003.
- [15] J. Kärkkäinen and E. Ukkonen. Lempel-ziv parsing and sublinear-size index structures for string matching. In *South American Workshop on String Processing*, pages 141–155, 1996.

- [16] Y. Lifshits. Processing compressed texts: a tractability border. In *Combinatorial Pattern Matching*, pages 228–240, 2007.
- [17] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Combinatorial Pattern Matching*, pages 113–124, 1994.
- [18] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Algorithms and Complexity*, pages 306–315, 2000.
- [19] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in z-compressed files. *Journal of Computer and System Sciences*, 52:299–307, 1993.
- [20] P. Bille, R. Fagerberg, and I. L. Gørtz. Improved approximate string matching and regular expression matching on ziv-lempel compressed texts. *ACM Transactions on Algorithms*, 6(1):3:1–3:14, December 2009.
- [21] P. Cégielski, I. Guessarian, Y. Lifshits, and Y. Matiyasevich. Window subsequence problems for compressed texts. In *International Computer Science Conference on Theory and Applications*, pages 127–136, 2006.
- [22] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over ziv-lempel compressed text. In *Combinatorial Pattern Matching*, pages 195–209, 2000.
- [23] V. Mäkinen, G. Navarro, and E. Ukkonen. Approximate matching of run-length compressed strings. In *Combinatorial Pattern Matching*, pages 31–49, 2001.
- [24] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Data Compression Conference*, pages 459–468, 2001.
- [25] O. Arbell, G. M. Landau, and J. S. B. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307–314, September 2002.
- [26] H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Information Processing Letters*, 54(2):93–96, April 1995.
- [27] M. Crochemore, G. M. Landau, and M. Ziv-ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 679–688, 2002.
- [28] D. Hermelin, G. M. Landau, S. Landau, and O. Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In *Symposium on Theoretical Aspects of Computer Science*, pages 529–540, 2009.
- [29] J. J. Liu, G. S. Huang, Y. L. Wang, and R. C. T. Lee. Edit distance for a run-length-encoded string and an uncompressed string. *Information Processing Letters*, 105(1):12–16, January 2008.
- [30] P. Loh, M. Baym, and B. Berger. Compressive genomics. *Nature Biotech*, 30(7):627–630, July 2012.
- [31] R. Dugad and N. Ahuja. A fast scheme for image size change in the compressed domain. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(4):461–474, April 2001.
- [32] R. V. Babu and K.R. Ramakrishnan. Compressed domain human motion recognition using motion history information. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages III – 41–4 vol.3, April 2003.
- [33] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. In *Journal of Computer and System Sciences*, pages 123–133, 1991.

- [34] C. Karande, K. Chellapilla, and R. Andersen. Speeding up algorithms on compressed web graphs. In *Conference on Web Search and Data Mining*, pages 272–281, 2009.
- [35] A.B. Kahng and G. Robins. Optimal algorithms for determining regularity in pointsets. In *Canadian Conference on Computational Geometry*, pages 167–170, 1991.
- [36] A.B. Kahng and G. Robins. Optimal algorithms for extracting spatial regularity in images. In *Pattern Recognition Letters*, pages 757–764, 1991.
- [37] G. Robins and B. L. Robinson. Landmine detection from inexact data. In *International Symposium on Aerospace/Defence Sensing and Dula-Use Photonics*, pages 189–195, 1994.
- [38] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Data Compression Conference*, pages 296–305. IEEE Computer Society, 1999.
- [39] S. H. Gerez. *Algorithms for VLSI Design Automation*. John Wiley & Sons, Inc., 1st edition, 1999.
- [40] N. A. Sherwani. *Algorithms for VLSI Physical Design Automation*. Kluwer Academic Publishers, 1993.
- [41] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *International World Wide Web Conference*, pages 595–602, 2004.
- [42] V. Kumar, S. Arya, and H. Ramesh. Hardness of set cover with intersection 1. In *International Colloquium on Automata, Languages, and Programming*, pages 624–635, 2000.
- [43] N. Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM*, 31(1):114–127, January 1984.
- [44] H. Cohn, A. Kumar, S. Miller, D. Radchenko, and M. Viazovska. The sphere packing problem in dimension 24. *Annals of Mathematics*, 185(3):1017–1033, 2016.
- [45] F. Claude and G. Navarro. A fast and compact web graph representation. In *String Processing and Information Retrieval*, pages 118–129, 2007.
- [46] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
- [47] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings. In *SIAM-ACM Symposium on Discrete Algorithms*, pages 373–389, 2011.
- [48] S. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1):2–22, 1985. International Conference on Foundations of Computation Theory.
- [49] E. Allender. Circuit complexity before the dawn of the new millennium 1. In *Foundations of Software Technology and Theoretical Computer Science*, pages 1–18, 1997.
- [50] M. Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.
- [51] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1979.
- [52] R.A. Baeza-Yates. *Efficient Text Searching*. PhD thesis, University of Waterloo, 1989.
- [53] K. Wang and K. Skadron. Cellular automata on the micron automata processor. University of Virginia Technical Report # CS-2015-03, 2015.
- [54] S. Wolfram. *A New Kind of Science*. Wolfram Media Inc., Champaign, Illinois, USA, 2002.

- [55] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.
- [56] A. Fellah, H. Jürgensen, and S. Yu. Constructions for alternating finite automata. *International Journal of Computer Mathematics*, 35(1-4):117–132, 1990.
- [57] J. Brzozowski and E. Leiss. On equations for regular languages, finite automata, and sequential networks. *Theoretical Computer Science*, 10(1):19–35, January 1980.
- [58] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar*. Springer-Verlag New York, Inc., New York, NY, USA, 1997.
- [59] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, October 1980.
- [60] M. Holzer and B. König. On deterministic finite automata and syntactic monoid size. *Theoretical Computer Science*, 327(3):319 – 347, 2004. Developments in Language Theory.
- [61] K. Roy, A. Srivastava, M. Nourian, M. Becchi, and S. Aluru. High performance pattern matching using the automata processor. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 1123–1132, May 2016.
- [62] K. Wang, Y. Qi, J.J. Fox, M.R. Stan, and K. Skadron. Association rule mining with the micron automata processor. In *Parallel and Distributed Processing Symposium*, pages 689–699, May 2015.
- [63] K. Zhou, J.J. Fox, Wang K, D.E. Brown, and K. Skadron. Brill tagging on the micron automata processor. In *Semantic Computing*, pages 236–239, February 2015.
- [64] C. Bo, K. Wang, J. Fox, and K. Skadron. Entity resolution acceleration using micron’s automata processor. In *SRC TechCon*, 2016.
- [65] T. Tracy II, Y. Fu, I. Roy, E. Jonas, and P. Glendenning. Toward machine learning on the automata processor. In *International Supercomputing Conference - High Performance Computing*, 2016.
- [66] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (2nd edition)*. Chapman and Hall, 2014.
- [67] O. Goldreich, S. Vadhan, and A. Wigderson. Simplified derandomization of bpp using a hitting set generator. In *Studies in Complexity and Cryptography*, pages 59–67, 2011.
- [68] N. Metropolis. The beginning of the monte carlo method. *Los Alamos Science*, No. 15, 1987.
- [69] L Blum, M Blum, and M Shub. A simple unpredictable pseudo random number generator. *SIAM Journal on Computing*, 15(2):364–383, May 1986.
- [70] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.
- [71] B. Applebaum. Pseudorandom generators with long stretch and low locality from random local one-way functions. *SIAM Journal on Computing*, 42(5):2008–2037, 2013.
- [72] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.
- [73] P. L’Ecuyer and R. Simard. Testu01: A c library for empirical testing of random number generators. *ACM transactions on mathematical software*, 33(4), August 2007.

- [74] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
- [75] G. Marsaglia. Diehard: a battery of tests of randomness. See <http://stat.fsu.edu/~geo/diehard.html>, 1996.
- [76] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, S. Vo, and L. E. Bassham, III. A statistical test suite for random and pseudorandom number generators for cryptographic applications, 2010.
- [77] J. Wadden, K. Wang, M. Stan, and K. Skadron. Uses for random and stochastic input on micron's automata processor. University of Virginia Technical Report # CS-2015-06, 2015.
- [78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.
- [79] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, November 1978.
- [80] L. Pitt and M. K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *Journal of the ACM*, 40(1):95–142, January 1993.
- [81] M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM*, 41(1):67–95, January 1994.
- [82] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987.
- [83] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *ACM Symposium on Theory of Computing*, pages 411–420, 1989.
- [84] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie. Efficient learning of typical finite automata from random walks. In *ACM Symposium on Theory of Computing*, pages 315–324, 1993.
- [85] D. Angluin and D. Chen. Learning a random dfa from uniform strings and state information. In *International Conference on Algorithmic Learning Theory*, pages 119–133, 2015.
- [86] T. Reuters. Samsung moving to 40-nm dram. See <http://www.pcmag.com/article2/0,2817,2340419,00.asp> [retrieved July 2017], 2009.
- [87] M. Manssen, M. Weigel, and A. K. Hartmann. Random number generators for massively parallel simulations on gpu. *The European Physical Journal-Special Topics*, 210(1):53–71, 2012.
- [88] Hybrid memory cube specification 2.0. http://www.hybridmemorycube.org/files/SiteDownloads/HMC-30G-VSR_HMCC_Specification_Rev2.0_Public.pdf.
- [89] R. Shaltiel. *An Introduction to Randomness Extractors*, pages 21–41. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [90] L. Trevisan. Extractors and pseudorandom generators. *Journal of the ACM*, 48(4):860–879, July 2001.
- [91] R. Shaltiel. Recent developments in explicit constructions of extractors. In *Bulletin of the EATCS*, volume 77, pages 67–95, 2002. Special Issue on Cryptography.
- [92] M. Blum. Independent unbiased coin flips from a correlated biased source: a finite state markov chain. In *Proceedings of the 26th International Conference on Algorithmic Learning Theory - Volume 9355*, pages 425–433, 1984.

- [93] M. Santha and U. V. Vazirani. Generating quasi-random sequences from semi-random sources. *Journal of Computer and System Sciences*, 33:75–87, 1986.
- [94] A. Cohen and A. Wigderson. Dispersers, deterministic amplification, and weak random sources. In *Symposium on Foundations of Computer Science*, 1989.
- [95] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.
- [96] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.
- [97] A. Kirsch and M. Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Structures and Algorithms*, 33(2):187–218, September 2008.
- [98] J. Huang, J. Lach, and G. Robins. A methodology for energy-quality tradeoff using imprecise hardware. In *Design Automation Conference*, pages 504–509, 2012.
- [99] R. L. De Queiroz. Processing jpeg-compressed images and documents. *IEEE Transactions on Image Processing*, 1999.
- [100] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.
- [101] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. The use of asymmetric numeral systems as an accurate replacement for huffman coding. In *Picture Coding Symposium*, pages 65–69, May 2015.
- [102] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In *International Colloquium on Automata, Languages and Programming*, pages 261–270. Springer, 1998.
- [103] D. Adjeroh, T. Bell, and A. Mukherjee. *Pattern Matching in Compressed Texts and Images*. Now Publishers Inc., Hanover, MA, USA, 2013.
- [104] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Symposium on String Processing and Information Retrieval*, pages 90–95. IEEE CS Press, 1998.

Appendix A

Compression-Aware Algorithms implementations

1.1 Lossless Set of Lines Python Code

```
# AMESCS Algorithm Implementation for Arbitrary Dimensions
# Input: An arbitrary finite pointset  $P$  in  $E^d$ 
# Output: All collinear equally-spaced subsets of  $P$ 
```

```
import ipdb
import numpy
from graph import Graph
import sys
```

```
class CP: # Compressed Points
    def __init__(self):
        self.lines = []
        self.points = []
```

```

# utility functions
def collinear(p1, p2, p3):
    if dist(p1, p2) + dist(p2, p3) == dist(p1, p3):
        return True
    else:
        return False

def dist(p1, p2):
    return numpy.linalg.norm(p2 - p1)

def ces(p1, p2, p3):
    return collinear(p1, p2, p3) and dist(p1, p2) == dist(p2, p3)

# /utility functions

class PointCompressor:
    def __init__(self, dimensions=2, pmin=0, pmax=10, pcount=500):

        self.dimensions = dimensions
        self.pcount = pcount
        self.pmax = pmax
        self.pmin = pmin

    def compress_points(self, P):
        # P = (p1, ..., pn) = Sort P by x1 coordinate
        # G = (V,E) = (EMPTY, EMPTY)

        self.pcount = len(P)

```



```

cp = CP()
P = P[P[:, o].argsort()] # weird sorting magic
g = Graph(self.dimensions)

for i in range(o, len(P) - 1):
    if P[i][o] == P[i + 1][o]:
        sys.exit("X coordinate for each point must be unique. "+
            "A standard method of handling this would be "+
            "rotating every point by a small amount "+
            "around the z axis, because of imprecision "+
            "of floats in Python, this technique causes "+
            "lossiness that can break the algorithm "+
            "(i.e. shift points minutely so "+
            "colinear points are not stored as such)")

added = []
ipdb.set_trace()
for A in range(o, self.pcount - 1):

    B = A + 1
    C = A + 2
    while A < self.pcount \
        and C < self.pcount \
        and B < self.pcount:
        if P[A][o] != P[B][o] and \
            P[B][o] != P[C][o] and \
            P[A][o] != P[C][o] and \
            (ces(P[A], P[B], P[C])):

            # add the new vertices
            newv1 = [P[A], P[B]]
            newv2 = [P[B], P[C]]

```

```

newv1t = [tuple(row) for row in newv1]
newv2t = [tuple(row) for row in newv1]

if newv1t not in added:
    added.append(newv1t)
    t = numpy.concatenate((g.vertices, [newv1]))
    g.vertices = t
if newv2t not in added:
    added.append(newv2t)
    t = numpy.concatenate((g.vertices, [newv2]))
    g.vertices = t
# remove duplicates
newedge = [[P[A], P[B]], [P[B], P[C]]]
g.edges = numpy.concatenate((g.edges, [newedge]))
if P[B][o] - P[A][o] > P[C][o] - P[B][o]:
    # review this line eventually -
    # it was just ">" in the pseudocode, not ">="
    C += 1
else:
    B += 1
    # this one goes too high and breaks the whole thing

vertex_tuples = []
edge_tuples = []
for v in g.vertices:
    p1 = tuple(v[0])
    p2 = tuple(v[1])
    vertex_tuples.append((p1, p2))
for e in g.edges:
    v1 = e[0]
    v2 = e[1]

```

```

    p1 = tuple(v1[0])
    p2 = tuple(v1[1])
    p3 = tuple(v2[0])
    p4 = tuple(v2[1])
    edge_tuples.append(((p1, p2), (p3, p4)))

# for each unexplored point:
# conduct BFS on it and add each
# explored point to one list in a list

explored_vertices = []
for v in vertex_tuples:
    if v not in explored_vertices:
        # create empty set S
        S = set()
        # create empty queue Q
        Q = []
        # add root to S
        S.add(v)
        # enqueue root
        Q.append(v)
        # while Q is not empty:
        while Q:
            # cur = Q.dequeue()
            cur = Q.pop(0)
            explored_vertices.append(cur)
            # for each node n that is adjacent to current:
            n1 = [n[0] for n in edge_tuples if cur == n[1]]
            n2 = [n[1] for n in edge_tuples if cur == n[0]]
            neighbors1 = n1
            neighbors2 = n2
            neighbors = neighbors1 + neighbors2

```

```

        for n in neighbors:
            # if n is not in S:
            if n not in S:
                # add n to S
                S.add(n)
                # Q.enqueue(n)
                Q.append(n)

        s_list = sorted(S, key=lambda item: (item[0], item[1]))
        compressed_line = (s_list[0], len(s_list) - 1)
        cp.lines.append(compressed_line)

    for p in P:
        p = tuple(p)
        if (p not in [n[0] for n in vertex_tuples]) \
            and (p not in [n[1] for n in vertex_tuples]):
            cp.points.append(p)

    return cp

```

1.2 Lossy Set of Lines Python Code

```

from __future__ import generators
import Points
import math
from scipy.optimize import linprog

# Given a point, the left-most point, the bottom-most point,
#and epsilon give a point back corresponding to grid slot
# e.g. given (3, 5) would mean grid[3][5]
def find_bucket(pt, left, bottom, eps):
    """Returns the array indices that a point belongs to
    after generating the grid

    Arguments:

```

```

    pt {list of integers} — the point from Points.py
    that you want to find the grid slot for
    left {list of integers} — The leftmost point in the dataset ,
    retrived from Points.get_left_point()
    bottom {list of integers} — The bottommost point in the
    dataset , retrived from Points.get_bottom_point
    eps {float} — Degree of error
    """

    return [int((pt[0]-(left[0]-4*eps))),int((pt[1]-(bottom[1]-4*eps)))]

def extend_right(point1 , point2 , epsilon , points , grid):
    """[summary]

    [description]

    Arguments:
        point1 {[type]} — [description]
        point2 {[type]} — [description]
        epsilon {[type]} — [description]
        points {[type]} — [description]
        grid {[type]} — [description]

    Returns:
        [type] — [description]
    """
    canExtend = True
    strtpts = [point1 , point2]
    while(canExtend):
        idlpt=[ strtpts[-1][0]+math.fabs( strtpts[-1][0]- strtpts[-2][0]),
        strtpts[-1][1]+math.fabs( strtpts[-1][1]- strtpts[-2][1])]
        lpt = Points.get_left_point(points)

```

```

bpt = Points.get_bottom_point(points)
bucket = find_bucket(idlpt, lpt, bpt, epsilon)
numPoints = len(strtpts)
for x in range(-1, 2):
    for y in range(-1, 2):
        if numPoints < len(strtpts):
            continue
        try:
            slot = grid[bucket[0] + x][bucket[1] + y]
            for point in slot:
                if point in strtpts:
                    continue
                if math.fabs(point[0] - idlpt[0]) <= 4*epsilon
                and math.fabs(point[1] - idlpt[1]) <= 4*epsilon:
                    c = [0, 0, epsilon]
                    A = []
                    b = []
                    for i in range(len(strtpts)):
                        #two bounds for each point
                        A.append([strtpts[i][0], 1, -1])
                        b.append(strtpts[i][1])
                        A.append([-strtpts[i][0], -1, -1])
                        b.append(-strtpts[i][1])
                    A.append([point[0], 1, -1])
                    b.append(point[1])
                    A.append([-point[0], -1, -1])
                    b.append(-point[1])
                    lp=linprog
                    r=lp(c, A_ub=A, b_ub=b, options={"disp": True})
                    if r.success and float(r.x[-1]) <= epsilon:
                        strtpts.append(point)

```

```

        except IndexError as err:
            #avoid out of bounds on the boundry of the grid
            # print(str(bucket[0]))
            # print(str(bucket[1]))
            # print(len(grid))
            # print(len(grid[0]))
            # print("ERRRRR")
            continue
    if numPoints < len(strtpts):
        canExtend = True
    else:
        canExtend = False
    return strtpts

```

```
def extend_left(point1, point2, epsilon, points, grid):
```

```
    """[summary]
```

```
    [description]
```

```
    Arguments:
```

```
        point1 {[type]} — [description]
```

```
        point2 {[type]} — [description]
```

```
        epsilon {[type]} — [description]
```

```
        points {[type]} — [description]
```

```
        grid {[type]} — [description]
```

```
    Returns:
```

```
        [type] — [description]
```

```
    """
```

```
    canExtend = True
```

```
    strtpts = [point1, point2]
```

```

while(canExtend):
    idlpt=[ strtpts[-1][0]-math.fabs( strtpts[-1][0]- strtpts[-2][0]),
    strtpts[-1][1]-math.fabs( strtpts[-1][1]- strtpts[-2][1])]
    lpt = Points.get_left_point(points)
    bpt = Points.get_bottom_point(points)
    bucket = find_bucket(idlpt, lpt, bpt, epsilon)
    numPoints = len(strtpts)
    for x in range(-1, 2):
        for y in range(-1, 2):
            if numPoints < len(strtpts):
                continue
            try:
                slot = grid[bucket[0] + x][bucket[1] + y]
                for point in slot:
                    if point in strtpts:
                        continue
                    if math.fabs(point[0]-idlpt[0])<=4*epsilon
                    and math.fabs(point[1] - idlpt[1])<=4*epsilon:
                        c = [0, 0, epsilon]
                        A = []
                        b = []
                        for i in range(len(strtpts)):
                            #two bounds for each point
                            A.append([ strtpts[i][0], 1, -1])
                            b.append( strtpts[i][1])
                            A.append([- strtpts[i][0], -1, -1])
                            b.append(- strtpts[i][1])
                        A.append([ point[0], 1, -1])
                        b.append(point[1])
                        A.append([- point[0], -1, -1])
                        b.append(-point[1])
                        lp = linprog

```



```

        r=lp(c,A_ub=A,b_ub=b,options={"disp":True})
        if r.success and float(r.x[-1]) <= epsilon:
            strtpts.append(point)

    except IndexError as err:
        #avoid out of bounds on the boundry of the grid
        # print(str(bucket[0]))
        # print(str(bucket[1]))
        # print(len(grid))
        # print(len(grid[0]))
        # print("ERRRRR")
        continue
    if numPoints < len(strtpts):
        canExtend = True
    else:
        canExtend = False
return strtpts

```

```
def check_distance(points, epsilon):
```

```
    """Returns true if any two points are within 8*epsilon of each other
```

Arguments:

points{list of a list of integers}--

List of points from Points.py

epsilon {float} — Degree of error

Returns:

bool — If all of the points are $\geq 8 \times \text{epsilon}$ apart,

return false; Otherwise return true

```
    """
```

```

    for point1 in points:
        for point2 in points:
            if point1 == point2:
                continue
            if distance(point1, point2) < 8 * epsilon:
                return True
    return False

def distance(point1, point2):
    """Distance between two points

    Arguments:
        point1 {list of integers} —
        point2 {list of integers} —

    Returns:
        float — distance between the two points
    """
    return math.sqrt((point1[0]-point2[0])**2+(point1[1]-point2[1])**2)

epsilon = float(1/8)
points = Points.generatePoints(75, 0, 30, 0, 30)
while(check_distance(points, epsilon)):
    points = Points.generatePoints(75, 0, 30, 0, 30)
#points = [[4,2], [2,2], [0,2], [4, 4], [2, 4], [0, 4]]
grid = []

#calculate how many grid blocks there are in each dimension
rpt = Points.get_right_point(points)
lpt = Points.get_right_point(points)
tpt = Points.get_top_point(points)

```

```

bpt = Points.get_bottom_point(points)
width = (rpt[0] + 4 * epsilon - (lpt[0] - 4 * epsilon)) / (8 * epsilon)
height = (tpt[1] + 4 * epsilon - (bpt[1] - 4 * epsilon)) / (8 * epsilon)

#create the grid
for x in range(math.ceil(width)):
    grid.append([])
for slot in grid:
    for y in range(math.ceil(height)):
        slot.append([])

#place each point in its corresponding grid slot
left = Points.get_left_point(points)[0] - 4 * epsilon
bottom = Points.get_bottom_point(points)[1] - 4 * epsilon
for point in points:
    first = math.floor(point[0] - left)
    second = math.floor(point[1] - bottom)
    try:
        lpt = Points.get_left_point(points)
        bpt = Points.get_bottom_point(points)
        bucket = find_bucket(point, lpt, bpt, epsilon)
        grid[bucket[0]][bucket[1]].append(point)
    except IndexError as err:
        print(len(grid))
        print(first)
        print(len(grid[0]))
        print(second)

extend = []
for i in points:
    for j in points:
        if i is not j:

```

```

        right = extend_right(i, j, epsilon, points, grid)
        left = extend_left(i, j, epsilon, points, grid)
        combined = []
        for k in range(len(left) - 1, 1, -1):
            combined.append(left[k])
        for k in range(len(right)):
            combined.append(right[k])
        extend.append(combined)
for e in extend:
    if len(e) > 3:
        print(e)

import random

def generatePoints(num, x1, x2, y1, y2):
    """Returns a generated list of points in a given area

    Returns a list of <num> points randomly distributed from
        (x1, y1) to (but not including) (x2, y2)

    Arguments:
        num {int} — Number of points to be generated
        x1 {int} — X-coordinate of first (inclusive) corner
        x2 {int} — X-coordinate of second (exclusive) corner
        y1 {int} — Y-coordinate of first (inclusive) corner
        y2 {int} — Y-coordinate of second (exclusive) corner
    """
    points = []
    while(len(points) < num):
        rx = random.randint(x1, x2)
        rneg1to1_a = random.randint(-1, 1)
        roto2_a = random.randint(0, 2)
        ry = random.randint(y1, y2)

```

```

    rneg1to1_b = random.randint(-1, 1)
    roto2_b = random.randint(0,2)
    point = [rx+rneg1to1_a*roto2_a/2.0,ry)+rneg1to1_b*roto2_b/2.0]
    if point not in points:
        points.append(point)
return points

```

```
def get_left_point(points):
```

```
    """Returns the the left-most point given a list of points.
```

Given more than one left-most (lowest x-value) point,
the point with the lowest y-value is returned.

Arguments:

points {list of points} —

The list of points to be included in the
search for the left-most point.

```
    """
```

```
    points.sort(key=lambda point: point[0])
```

```
    left = []
```

```
    for point in points:
```

```
        if point[0] == points[0][0]:
```

```
            left.append(point)
```

```
    left.sort(key=lambda data: data[1])
```

```
    return left[0]
```

```
def get_right_point(points):
```

```
    """Returns the right-most point, given a list of points.
```

Given more than one right-most (highest x-value) point,
the point with the lowest y-value is returned.

Arguments:

points {list of points} — The list of points to be included in the search for the left–most point.

"""

```
points.sort(key=lambda point: point[0])
```

```
bottom = []
```

```
for point in points:
```

```
    if point[0] == points[len(points) - 1][0]:
```

```
        bottom.append(point)
```

```
bottom.sort(key=lambda data: data[1])
```

```
return bottom[0]
```

```
def get_bottom_point(points):
```

```
    """Returns the lowest y–value point given a list of points.
```

Returns lowest x–value point as a secondary key

if there is more than one lowest y–value point.

Arguments:

points {list of points} — The list of points from which to choose a bottom–most point.

"""

```
points.sort(key=lambda point: point[1])
```

```
bottom = []
```

```
for point in points:
```

```
    if point[1] == points[0][1]:
```

```
        bottom.append(point)
```

```
bottom.sort(key=lambda data: data[0])
```

```
return bottom[0]
```

```
def get_top_point(points):
```

```
    """Returns the highest y–value point given a list of points.
```

Returns lowest x-value point as a secondary key if there is more than one highest y-value point.

Arguments:

points {list of points} — The list of points from which to choose a top-most point.

"""

```
points.sort(key=lambda point: point[1])
bottom = []
for point in points:
    if point[1] == points[len(points) - 1][1]:
        bottom.append(point)
bottom.sort(key=lambda data: data[0])
return bottom[0]
```

1.3 Set-of-Lines Nearest Neighbor and Range Searches

```
def custom_round(x1, base=10):
    return float(base * round(float(x1) / base))
```

```
def distance(x1, y):
    return np.linalg.norm(y - x1)
```

```
def all_line_points_in_range(line, point, rng):
    # line defined as two points (A and B)
    l = np.asarray(line[0], dtype='float64')
    reps = line[1]
    c = np.array(point) # c is the query point
    a = l[0] + o # a is the first point on line
    b = l[1] + o # b is the second point on line
```

```

dist = distance(a, b)
slope = b - a
max_dist = dist * (reps + 1)
b -= a
c -= a # translate all points
farthest_point = l[0] + (slope * (reps + 1))
if np.linalg.norm(b) == 0:
    n = 0
else:
    n = b / np.linalg.norm(b)
d = np.dot(n, c) * n + a

# undo translation to origin
c += a
b += a
query_point = c

# next we find the distance to the hypothetical closest point
# if the collinear points repeated forever
d_dist = np.linalg.norm((d - a))
dist_query_to_line = np.linalg.norm((query_point - d))

# if the hypothetical closest point is too far away,
# we return the empty set
if dist_query_to_line >= rng:
    return None

d2 = dist_query_to_line * dist_query_to_line
chord_length = 2 * math.sqrt((rng * rng) - d2)
# chord length = sqrt(r^2 - d^2) * 2

# distance from our hypothetical point to the beginning of the line

```



```

first_included_dist = d_dist - (chord_length / 2)
# the number of reps it would take to get to that point
first_included_reps = math.ceil(first_included_dist / dist)
# the point itself
first_included_point = l[0] + (slope * first_included_reps)

# check if first included point is before the beginning of the line.
# If so, set beginning of line to A
if distance(farthest_point, first_included_point) > max_dist:
    first_included_point = a
    first_included_reps = 0

# same as above
last_included_dist = d_dist + (chord_length / 2)
# check if last_included_dist is greater than max dist.
# If so, use max_dist instead of last_included_dist
if last_included_dist > max_dist:
    last_included_dist = max_dist
last_included_reps = math.floor(last_included_dist / dist)
num_incl_pts = last_included_reps - first_included_reps + 1
frst_incl_pt=first_included_point

if number_of_included_points == 1:
    return tuple(first_included_point)
if number_of_included_points == 2:
    t=tuple(((frst_incl_pt, frst_incl_pt + slope), 0))
    return t
if number_of_included_points > 2:
    t=tuple(((frst_incl_pt, frst_incl_pt+slope), num_incl_pts-2))
    return t

```

```

def range_search(qp, rng, cp):
    points = cp.points
    lines = cp.lines
    point_list = []
    for l in lines:
        pts = all_line_points_in_range(l, qp, rng)
        if pts:
            for p in pts:
                point_list.append(p)
    for p in points:
        dist = np.linalg.norm(qp - p)
        if dist < rng:
            point_list.append(tuple(p))
    return point_list

```

```

def closest_point_on_line(line, point):
    # line defined as two points (A and B)
    l = np.asarray(line[0], dtype='float64')
    reps = line[1]
    c = np.array(point) # c is the query point
    a = l[0] + 0 # a is the first point on line
    b = l[1] + 0 # b is the second point on line
    dist = distance(a, b)
    slope = b - a
    max_dist = dist * (reps + 1)
    b -= a
    c -= a # translate all points towards origin
    farthest_point = l[0] + (slope * (reps + 1))
    if np.linalg.norm(b) == 0:
        n = 0
    else:

```

```

        n = b / np.linalg.norm(b)
    d = np.dot(n, c) * n + a # d is the point on the *continuous* line
    # which is closest to the query

    # undo translation to origin
    c += a
    b += a

    d_dist_beginning = distance(d, a)
    d_dist_end = distance(d, farthest_point)

    past_end = d_dist_beginning >= (max_dist - (dist / 2))
    past_end = past_end and d_dist_beginning > d_dist_end
    past_beginning = d_dist_end >= (max_dist - (dist / 2))
    past_beginning = past_beginning and d_dist_end > d_dist_beginning

    rounded_dist = custom_round(d_dist_beginning, base=dist)
    closest_point_reps = (rounded_dist / dist)
    closest_point = a + (slope * closest_point_reps)

    if past_end:
        return farthest_point
    elif past_beginning:
        return a
    else:
        return closest_point

def nearest_neighbor(qp, cp):
    # for each line, find the closest point on that line
    # this means making a perpendicular line
    # and finding where the two intersect

```

```

# then rounding that point to the nearest existing neighbor
points = cp.points
lines = cp.lines
query = qp
for l in lines:
    points.append(closest_point_on_line(l, query))

nearest_point = points.pop()
nearest_dist = np.linalg.norm(qp - nearest_point)
# should be infinite
for p in points:
    # figure out norm of qp-p
    # if less than before, set as new nearest
    dist = np.linalg.norm(qp - p)
    if dist < nearest_dist:
        nearest_point = p
        nearest_dist = dist
return nearest_point

def decompress(cp):
    ret_pt_list = []
    for p in cp.points:
        ret_pt_list.append(np.asarray(p))

    for line in cp.lines:
        reps = line[1]
        l = np.asarray(line[0])
        slope = l[1] - l[0]
        ret_pt_list.append(np.asarray(copy(l[0])))
        ret_pt_list.append(np.asarray(copy(l[1])))
        for i in range(1, reps + 1):

```

```

        l1 = np.asarray(copy(l[1]))
        mod = slope * i
        app = l1 + mod
        ret_pt_list.append(np.asarray(copy(app)))

    ret_pt_list = np.stack(ret_pt_list)
    return ret_pt_list

def naive_nn(qp, points):
    closest_point = points[0]
    closest_dist = distance(qp, closest_point)
    for p in points[1:]:
        cur_point = p
        cur_dist = distance(qp, cur_point)
        if cur_dist <= closest_dist:
            closest_dist = cur_dist
            closest_point = copy(cur_point)

    return closest_point

def naive_range_search(qp, rng, points):
    ret_list = []
    for p in points:
        if distance(qp, p) < rng:
            ret_list.append(copy(p))
    return ret_list

```

Appendix B

Poster DCC 2013

We presented the following poster and abstract, entitled "Algorithms for Compressed Inputs", at the IEEE 2013 Data Compression Conference, held at Snowbird, Utah, during March 2013.

Algorithms for Compressed Inputs

Nathan Brunelle, Gabriel Robins, abhi shelat

Problem:

- The size of data sets is dramatically increasing, effectively slowing down the performance of algorithms operating on that data.
- Although data is compressed for storage, most algorithms require explicit and uncompressed inputs

Our Solution:

- Adopt a framework by which algorithms are designed to exploit the regularity of input data, and are measured by their ability to do so.
- We suggest the development of algorithms designed to operate directly on compressed data.
- Here we present common algorithms for common compression schemes

Sequence Compression

Uncompressed Data:

"It was the best of times it was the worst of times it was the epoch of incredulity it was the season of light it was the season of darkness..."

LZ 77

Sort

$$O(C + \sum |\log(\Sigma| + n) \cdot \text{Time})$$



Compressed Input:

"It was the best of times (1 3) worst (5 2) (7 3) age (11 1) wisdom (13 5) foolishness (19 3) epoch (23 1) belief (25 5) incredulity (31 3) season (35 1) light (31 5) darkness"

Statistic $O(C) \cdot \text{Time}$

it was the best of times (1, 3) worst (5, 2) (7, 3) age (11, 1) wisdom (13, 5) foolishness (19, 3) epoch (23, 1) belief (25, 5) incredulity (31, 3) season (35, 1) light (31, 5) darkness"

Index: 17 = of

Context Free Grammar

Sort

$$O(C \cdot |\Sigma|) \cdot \text{Time}$$

It = <1,0,...,0>

Was = <0,1,0,...,0>

Darkness = <0,...,0,1>

A1 = it + was + the = <1,1,1,0,...,0>

A2 = of = <0,0,0,1,...,0>

A3 = A2 + times = <0,0,0,1,1,0,...,0>

A0 = <0,0,0,1,...,2,1,1>

LZ 78

Convert to CFG

$O(C) \cdot \text{Time}$

(0,1) (0,was) (0,the) (0,best) (0,of) (0,times) (1,was) (3,worst) (5,times).

A0 → A1 A2 A3 A4 A5 A6 A7 A8 A9
A1 → it
A2 → was
A3 → the
A4 → best
A5 → of
A6 → times
A7 → A1 was
A8 → A3 worst
A9 → A5 times

Graph Compression

Definition: Graph Compression

Previous: A compression of $G(V,E)$, called $G^*(V^*,E^*)$, must satisfy:

- $|V^*|$ is polynomial in $|V|$
- $E^* = o(E)$
- The mapping is 1-1

Re-Pair

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

Compressed Input

Uncompressed Data

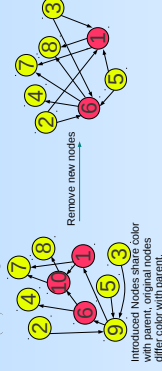
Compressed Input

Uncompressed Data

Issue: Satisfies the definition but has a trivial algorithm for bipartite assignment [compressed data] (red nodes)
Solution: Restrict to schemes which asymptotically approach entropy.
Forbidden decision problems.

Bipartite Assignment

$O(C) \cdot \text{Time}$



Boldi-Vigna

Compressed Input

| Node | Reference | Copy | Sequence | Residual | Original |
|------|-----------|------|----------|----------|--------------------------------------|
| 1 | 1 | 0101 | (8, 2) | 4, 13 | 4, 8, 9, 13 |
| 2 | | | (15, 4) | 3 | 3, 4, 13, 15, 16, 17, 18, 27, 28, 29 |
| ... | ... | ... | ... | ... | ... |

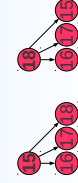
Bipartite Assignment

$O((|V| + s) \cdot \alpha(|V|)) \cdot \text{Time}$

| Node | Reference | Copy | Sequence | Residual | Original |
|------|-----------|------|----------|----------|--------------------------------------|
| 1 | 1 | 0101 | (8, 2) | 4, 13 | 4, 8, 9, 13 |
| 2 | | | (15, 4) | 3 | 3, 4, 13, 15, 16, 17, 18, 27, 28, 29 |
| ... | ... | ... | ... | ... | ... |

A node must share its color with the nodes in its reference field. All nodes in the reference field must be opposite color. All nodes in the reference field must be opposite color.

Sequences are checked on nodes using disjoint sets of bits. The sequence for each color (one with min element as representative, one with max element).



Algorithms for Compressed Inputs

Nathan Brunelle, Gabriel Robins, abhi shelat

Department of Computer Science, University of Virginia, Charlottesville, Virginia 22904, USA
 njb2b@virginia.edu, robins@cs.virginia.edu, shelat@cs.virginia.edu

Abstract We study compression-aware algorithms, i.e. algorithms that can exploit regularity in their input data by directly operating on compressed data. While popular with string algorithms, we consider this idea for algorithms operating on numeric sequences and graphs that have been compressed using a variety of schemes including LZ77, grammar-based compression, a graph interpretation of Re-Pair, and a method presented by Boldi and Vigna in The WebGraph Framework. In all cases, we discover algorithms outperforming a trivial approach: to decompress the input and run a standard algorithm. We aim to develop an algorithmic toolkit for basic tasks to operate on a variety of compression inputs.

Algorithms for Compressed Sequences We consider sorting algorithms that operate on data produced by the following three compression schemes: LZ77, context free grammar representation (called CFG), and LZ78. Note that in CFG an array is represented as the singleton language parsed from a grammar.

For sorting an LZ77-compressed sequence of numbers, we present a sorting algorithm which operates in time $O(C + |\Sigma| \log |\Sigma| + n)$ where C is the compression size, n is the length of the sequence, and Σ is the set of unique numbers in the input list. In most instances $C \ll n$, thus our algorithm in practice achieves linear sorting as compared to the classical algorithm's $O(n \log n)$ worst-case performance. We also present a way of indexing into the sequence in $O(C)$ time.

For sorting a list compressed by a context-free grammar (with LZ78 as a special case) we present an algorithm which finds the sorted sequence in $O(C \cdot |\Sigma|)$ time. Here, C represents the total number of symbols in all of the grammar's substitution rules. This result has the advantage of being independent of the size of the uncompressed list. From here, we can produce a grammar for the sorted list which has size $O(|\Sigma| \log n)$, where n is the length of the decompressed list. The classical approach would require $O(n \log n)$ time to decompress and then sort.

Algorithms for Compressed Graphs Next we consider topological sort and bipartite assignment on graphs under these two compression schemes: a graph interpretation of the Re-Pair compression scheme, and the scheme presented by Boldi and Vigna in the WebGraph Framework (called BV).

For graphs compressed using the Re-Pair algorithm, we perform both algorithms in $O(C)$ time. Re-Pair is a form of grammar compression, so C is the number of terms on the right side of the grammar's parse rules. These improvements compare favorably with the $O(|V| + |E|)$ trivial approach.

We present an algorithm which performs bipartite checking on a BV compressed graph. This algorithm runs in $O(|V| + s)$ time where $|V|$ is the number of vertices in the graph. In a graph's adjacency list, after vertex labels are sorted, there are often blocks of sequential labels. The BV compression scheme represents these sequences en masse, and s is the total number of such sequences. This improves the running time of the classical $O(|V| + |E|)$ approach.

Appendix C

Poster DCC 2015

We presented the following poster and abstract, entitled "Compression-Aware Algorithms for Massive Data Sets", at the IEEE 2015 Data Compression Conference, held at Snowbird, Utah, during April 2015.

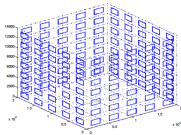


Compression-Aware Algorithms for Massive Data Sets

Nathan Brunelle, Gabriel Robins, abhi shelat

Department of Computer Science, University of Virginia

www.cs.virginia.edu/robins/compression



Problem:

- Massive **datasets** are stored **compressed**
- Algorithms** operate on **uncompressed** data

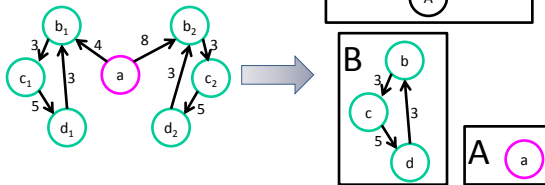
Solution: Operate *directly* on compressed data

Compression-aware Algorithms

Algorithmically-aware Compressions



Hierarchical Graph Compression



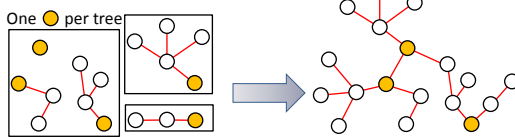
Minimum Spanning Tree

Modified Kruskal's

For each component in Compression:

Build forest of MSTs (don't merge "edge" nodes=●)

Merge forest into a single MST



Single Source Shortest Path

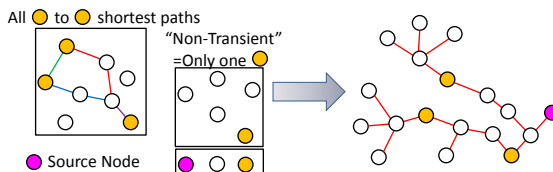
For each component:

Find all edge-edge shortest paths

Build "supergraph" of components

Find shortest path through "supergraph"

Find shortest path for all "non-transient" components



Set of Lines Compression

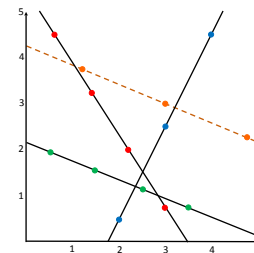
Assumption:

- Points are often collinear
- Occur at regular intervals

Equation of dashed line:

$$y = -\frac{5}{12}x + 4.25$$

Interval: 6.34



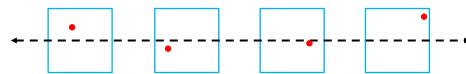
Compression Algorithm

Find all maximal equally-spaced collinear subsets $O(n^2)$

Find Minimal set of lines to cover points (approximate)

Lossy Case

- Points "nearly" regular and collinear
 - Runs $O(n^{5/2})$
- Must fit points to their line
 - Linear programming solution



Nearest Neighbor Queries

Input: Query Point p and a set of L lines in d dimensions

Output: the nearest point to p in the compressed point set

For each axis x_i do:

Project points onto the x_i $O(1)$

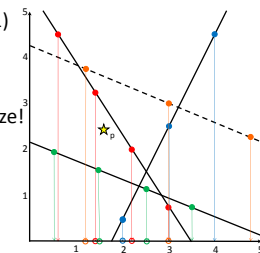
Save 2 nearest points $O(1)$

Find closest remaining pt $O(d \cdot L)$

$d \cdot L$ is **linear** in the compressed size!

This algorithm generalizes to:

- Manhattan range queries
- Euclidean range queries
- Convex Polytope Membership



Compression-Aware Algorithms for Massive Datasets

Nathan Brunelle, Gabriel Robins, abhi shelat

Department of Computer Science, University of Virginia, Charlottesville, VA, 22904
 njb2b@virginia.edu, robins@cs.virginia.edu, shelat@cs.virginia.edu

Abstract While massive datasets are often stored in compressed format, most algorithms are designed to operate on uncompressed data. We address this growing disconnect by developing a framework for compression-aware algorithms that operate directly on compressed datasets. Synergistically, we also propose new algorithmically-aware compression schemes that enable algorithms to efficiently process compressed data. In particular, we apply this general methodology to geometric / CAD datasets that are ubiquitous in areas such as graphics, VLSI, and geographic information systems. We develop algorithms and corresponding compression schemes that address different types of datasets, including pointsets and graphs. Our methods are more efficient than their classical counterparts, and they extend to both lossless and lossy compression scenarios. This motivates further investigation of how this approach can enable algorithms to process ever-increasing big data volumes.

Motivation Big datasets are emerging in all sectors of society, including industry, government, academia, and science [2]. Achieving the full potential of this data deluge requires addressing new and open questions, especially with respect to the scalability of data creation, storage, and processing. While much data is stored in compressed format, very few classic algorithms are able to process compressed data. We see this disconnect as an opportunity to mitigate the growing gap between dataset sizes and processing capability [1].

Compression-Aware Algorithms To overcome these problems we design algorithms to operate directly on compressed data. Each algorithm's speed and memory usage dramatically improve with the input's compressibility (i.e., descriptive complexity). This improvement derives from leveraging highly repetitive or parametrically specified input structures, leading to much smaller inputs (and outputs), and enabling algorithms to manipulate very large composite objects while interacting only with their succinct descriptions. To further bolster the efficacy of compression-aware algorithms, we also explore the design of algorithmically-aware compression schemes. These schemes are specifically designed to support a broad range of operations on compressed data.

We address classical problems in graph-based and geometrical domains, including topological sort, minimum spanning trees, shortest paths, nearest neighbors, range queries, and convex hulls. In the graph domain we present more efficient algorithms that operate on a pre-existing compression scheme. In the geometrical domain we present new compression schemes which enable the proposed algorithms to run more efficiently. See <http://www.cs.virginia.edu/robins/compression> for more details.

References

- [1] N. Brunelle, G. Robins, and a. shelat. Algorithms for compressed inputs. In *Data Compression Conference*, page 478, 2013.
- [2] J. Hurwitz, A. Nugent, F. Halper, and M. Kaufman. *Big Data for Dummies*. John Wiley Sons, Hoboken, NJ, USA, 2013.

Appendix D

APPRNG Patent Application

The following is U.S. Patent Application no. US2017/0083288A1 (filed April 6, 2016 and published on March 23, 2017), which is based on some of the research presented in this dissertation.



US 20170083288A1

(19) **United States**
(12) **Patent Application Publication** (10) **Pub. No.: US 2017/0083288 A1**
WADDEN et al. (43) **Pub. Date: Mar. 23, 2017**

(54) **SYSTEM, METHOD, AND COMPUTER-READABLE MEDIUM FOR HIGH THROUGHPUT PSEUDO-RANDOM NUMBER GENERATION**

(71) Applicant: **University of Virginia Patent Foundation, d/b/a University of Virginia Licensing & Ventures Group, Charlottesville, VA (US)**

(72) Inventors: **John Pierson WADDEN, Charlottesville, VA (US); Nathan James BRUNELLE, Charlottesville, VA (US)**

(73) Assignee: **University of Virginia Patent Foundation, d/b/a University of Virginia Licensing & Ventures Group**

(21) Appl. No.: **15/091,925**

(22) Filed: **Apr. 6, 2016**

Publication Classification

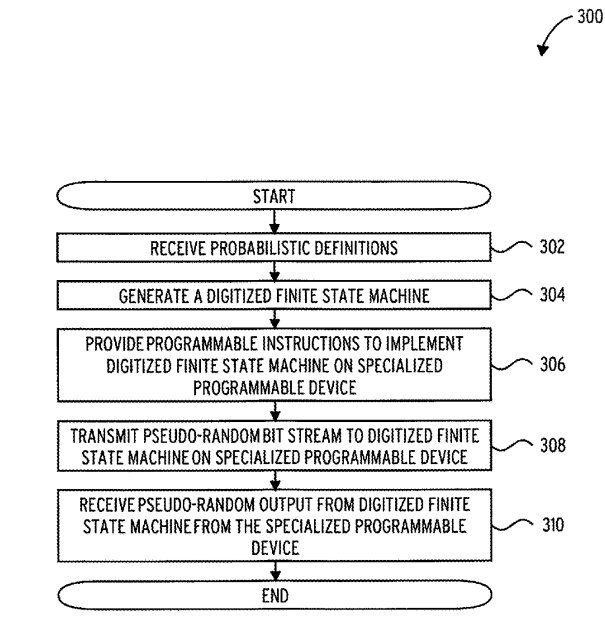
(51) **Int. Cl.**
G06F 7/58 (2006.01)

(52) **U.S. Cl.**
CPC G06F 7/584 (2013.01); **G06F 2207/583** (2013.01)

(57) **ABSTRACT**
Disclosed embodiments include systems, methods, and computer-readable media for generating pseudo-random numbers. Disclosed embodiments may receive, by the at least one processor, range data indicating a range of numbers. Disclosed embodiments may generate, based on the range data and by the at least one processor, a digitized finite state machine configured to produce pseudo-random output within the range of numbers. Further, disclosed embodiments may provide, by the at least one processor to a specialized pattern-matching device, programmable instructions to implement the digitized finite state machine on the specialized pattern-matching device. Disclosed embodiments may transmit, by the at least one processor to the specialized pattern-matching device, a pseudo-random bit stream for processing by the digitized finite state machine. Disclosed embodiments may receive, by the at least one processor from the specialized pattern-matching device, pseudo-random output from the digitized finite state machine.

Related U.S. Application Data

(60) Provisional application No. 62/147,045, filed on Apr. 14, 2015.



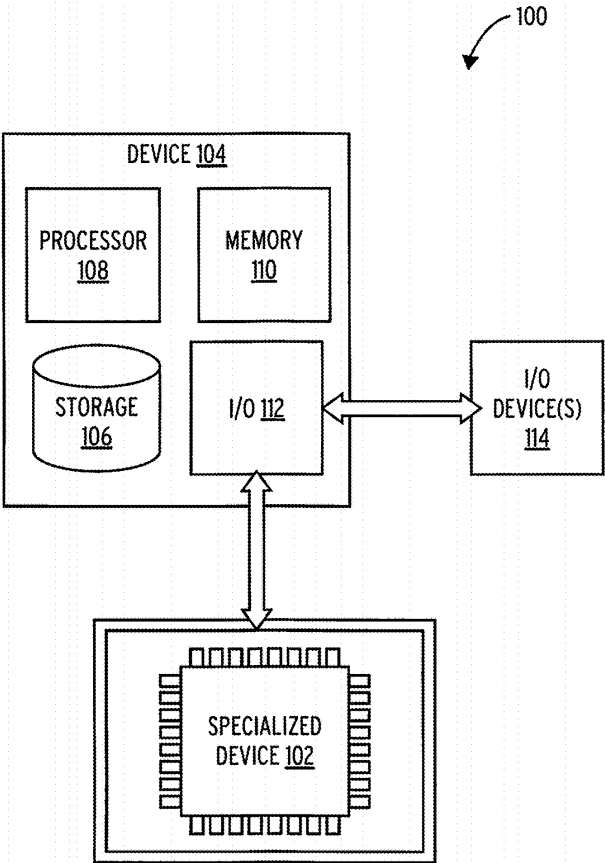


FIG. 1

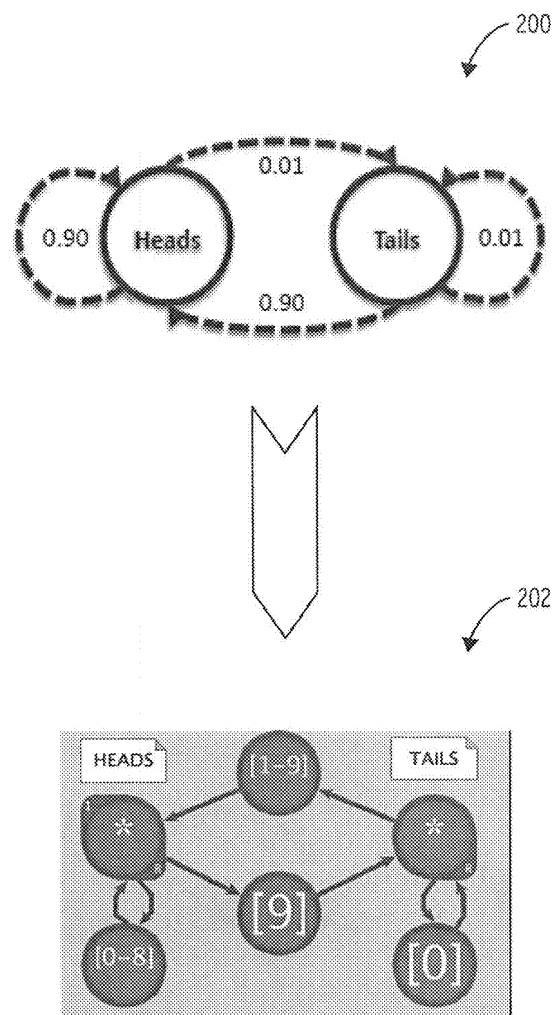


FIG. 2

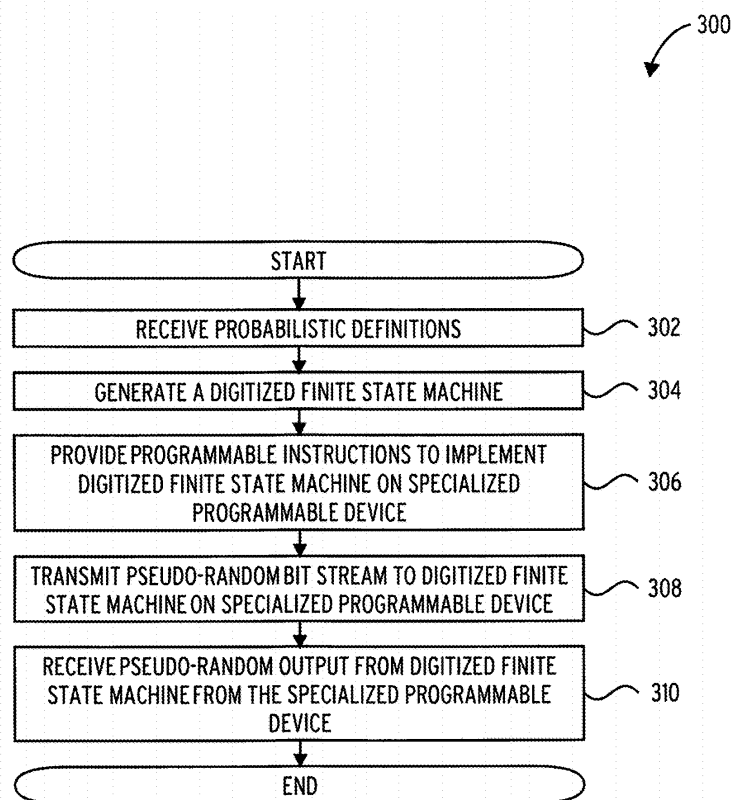


FIG. 3

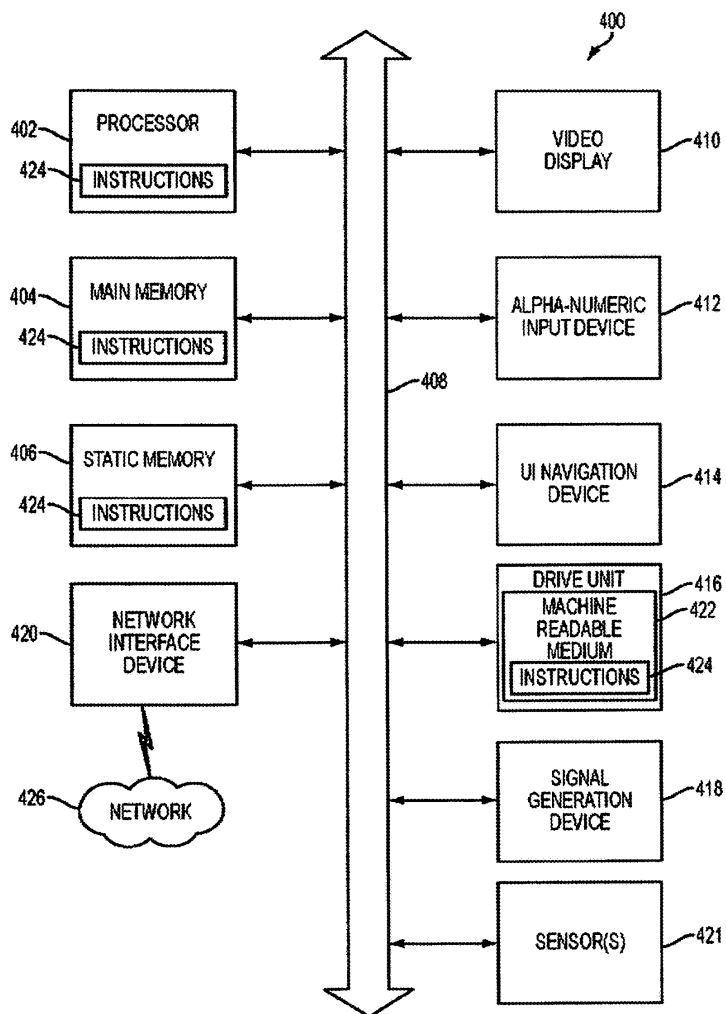


FIG. 4

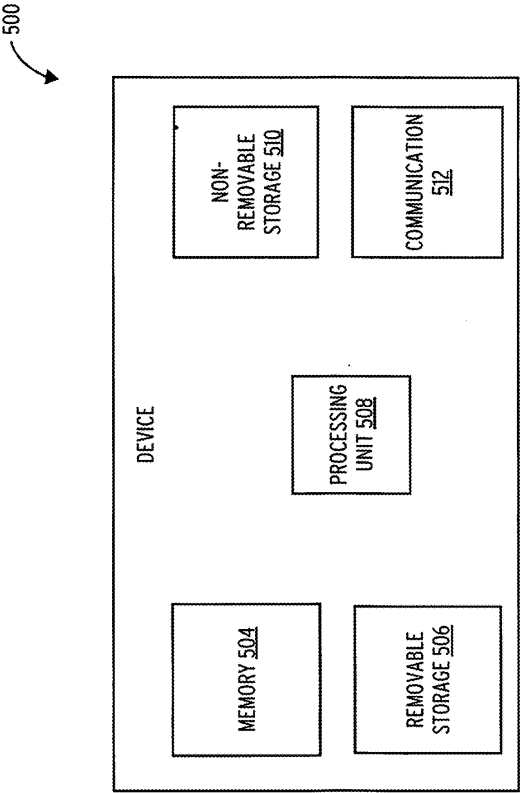


FIG. 5

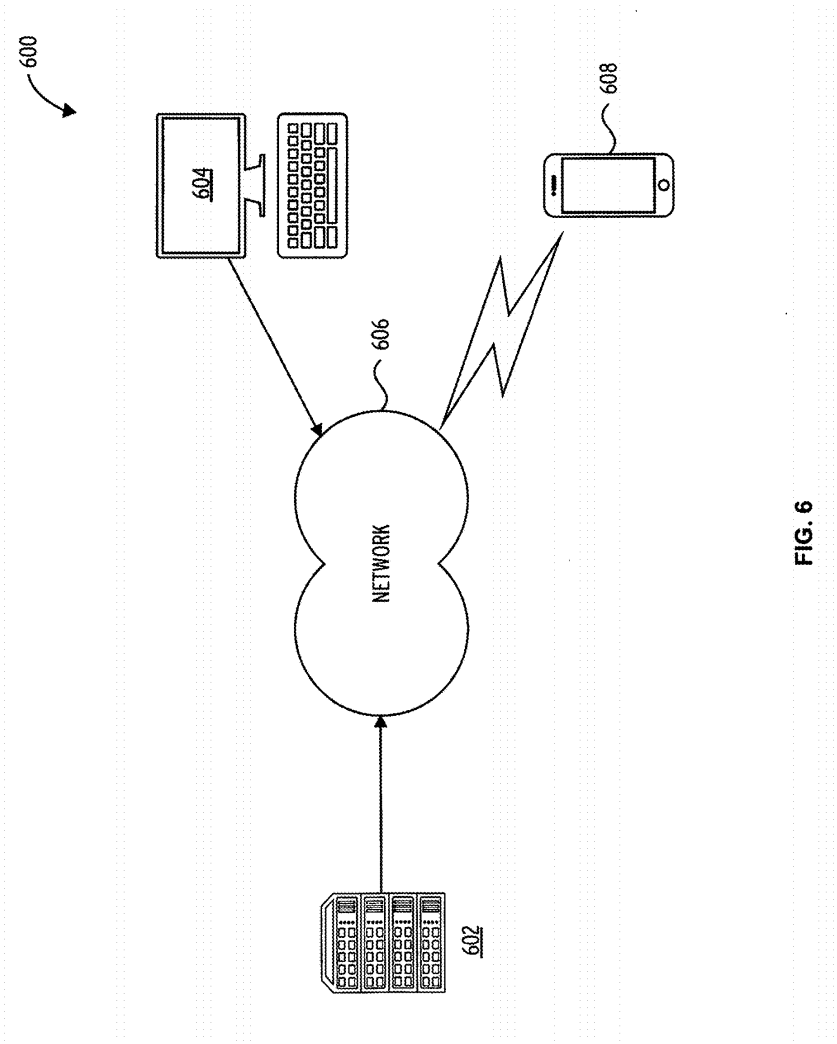


FIG. 6

SYSTEM, METHOD, AND COMPUTER-READABLE MEDIUM FOR HIGH THROUGHPUT PSEUDO-RANDOM NUMBER GENERATION

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of U.S. Provisional Patent Application No. 62/147,045, filed on Apr. 14, 2015. The contents of the above-referenced application are expressly incorporated herein by reference for all purposes.

GOVERNMENT LICENSE RIGHTS

[0002] This invention was made with government funds under Agreement No. HR0011-13-3-0002 awarded by DARPA. The U.S. Government has rights in this invention.

BACKGROUND

[0003] Pseudo-random number generation (PRNG) may be used in simulation and cryptographic applications. For example, Monte Carlo methods are pervasive simulation tools in physical and social sciences and rely on continuous random sampling to drive simulations of unpredictable processes. Monte Carlo simulations were among the first use cases for computers and are arguably some of the most important algorithms ever invented. Because fast and high-quality random number generation may be on the critical path of these applications, developing fast and high-quality PRNGs may improve the quality and speed of computational sciences.

[0004] Research into creating high-quality pseudo-random sequences has existed since the first Monte Carlo simulation on a digital computer. Today, while there are many PRNG algorithms, not all are created equal. No matter the method, pseudo-random output that is harder to distinguish from a truly random output better represents a truly random number stream. Statistical tests, such as the Knuth test, exist to identify patterns in pseudo-random sequences. These tests form the basis of many modern statistical test suites. The most comprehensive and stringent tests are the BigCrush test battery from TestU01 suite, which includes the functionality of the Knuth tests, DIEHARD, and the NIST statistical test suite. A test in the suite fails if it identifies a property of the pseudo-random sequence that should not exist in true randomness.

BRIEF SUMMARY

[0005] In accordance with embodiments of the present disclosure, computer-implemented systems, methods, and computer-readable media are provided for generating pseudo-random numbers. Embodiments of the present disclosure also include computerized systems, methods, and computer-readable media for programming a specialized pattern-matching device with a digitized finite state machine based on probabilistic characteristics. Embodiments of the present disclosure may be implemented for generating pseudo-random output.

[0006] In accordance with an embodiment, a computer-implemented system is provided for generating pseudo-random numbers. The system may include a storage device that stores instructions and at least one processor that executes the instructions. The instructions may cause the at least one processor to receive, by the at least one processor,

range data indicating a range of numbers. Also, the instructions may cause the at least one processor to generate, based on the range data and by the at least one processor, a digitized finite state machine configured to produce pseudo-random output within the range of numbers. Further, the instructions may cause the at least one processor to provide, by the at least one processor to a specialized pattern-matching device, programmable instructions to implement the digitized finite state machine on the specialized pattern-matching device. The instructions may cause the at least one processor to transmit, by the at least one processor to the specialized pattern-matching device, a pseudo-random bit stream for processing by the digitized finite state machine. The instructions may additionally cause the at least one processor to receive, by the at least one processor from the specialized pattern-matching device, pseudo-random output from the digitized finite state machine.

[0007] Computer-readable media are also provided for implementing methods of the present disclosure. Additional embodiments and related features of the present disclosure are presented herein.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

[0008] To easily identify the discussion of any particular element or act, the most significant digit or digits in a reference number refer to the figure number in which that element is first introduced.

[0009] FIG. 1 illustrates an exemplary system for generating pseudo-random numbers in accordance with disclosed embodiments.

[0010] FIG. 2 illustrates an exemplary finite state machine for generating pseudo-random numbers in accordance with disclosed embodiments.

[0011] FIG. 3 illustrates an exemplary process for generating pseudo-random numbers in accordance with disclosed embodiments.

[0012] FIG. 4 illustrates a block diagram illustrating an example of a machine upon which one or more aspects of embodiments of the present invention can be implemented.

[0013] FIG. 5 illustrates an exemplary computing device in accordance with disclosed embodiments.

[0014] FIG. 6 illustrates an exemplary computing environment in accordance with disclosed embodiments.

DETAILED DESCRIPTION

[0015] Disclosed embodiments may be directed to systems and methods for a fast, scalable, and high-quality pseudo-random number generator (PRNG). In designing a random number generator, one may be faced with deciding between the competing tradeoffs of efficient computer processing and the quality of randomness of the results. For example, a random number generator may produce results with a very high amount of randomness (e.g., quality). However, the high quality generator may require impractical computing resources and/or utilize an excessively long run time. Disclosed embodiments may be related to an improved PRNG that improves computing efficiency while maintaining a predefined level of quality for the randomness of the results. Additionally, disclosed embodiments may permit selection of higher quality results (e.g., increased randomness).

[0016] As the breakdown in Dennard scaling makes it increasingly expensive to improve performance of traditional serial von Neumann architectures, heterogeneous computing, involving graphics processing units (GPUs), digital signal processors (DSPs), field programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs) and other processors may provide improved solutions. By matching computation kernels to the most effective or efficient available processor, disclosed embodiments may provide power efficiency and performance gains at current transistor technology nodes. Micron, leveraging their experience and IP in memory technology, has developed the Automata Processor (AP), a large-scale, native-hardware implementation of non-deterministic finite automata (NFA). While the AP is not suitable for traditional integer or floating point computation, NFAs are extremely powerful and efficient pattern matchers, and have been shown to provide large speedups over von Neumann architectures such as CPUs and GPUs for rule-based datamining kernels.

[0017] An AP implements an NFA using a reconfigurable network of state transition elements (STEs) that consume an input stream of 8-bit symbols. Each STE can be activated and cause transitions to other STEs. STEs are capable of single-bit reports, analogous to “accepting states” in traditional NFAs. Disclosed embodiments may utilize an AP to form a fast, scalable, and high-quality PRNG.

[0018] Instead of driving automata transitions using conventional input (e.g. a DNA sequence), disclosed embodiments may dictate automata transitions using input designed to be random or pseudorandom. Because activations of STEs in the AP are conditional on the input stream, a probabilistic or random input stream may provide probabilistic or random automata transitions, even though the transition rules are deterministic. Thus, probabilistic automata, including finite state Markov chains, may be emulated using the AP.

[0019] Accordingly, disclosed embodiments may create a scalable, high-throughput, and high-quality PRNG using Markov chains modeled by STEs on an AP. Some embodiments may use parallel Markov chains to model rolls of fair dice, and then combine the results of each roll into a new random output string. By combining the output of parallel rolls, driven by a single stream of random input symbols, disclosed embodiments may construct a new pseudo-random output many times larger than the random input used to drive transitions on a chip. Though, emulating Markov chains using NFAs with fixed transition functions may cause any number of parallel Markov chains that consume the same input to produce output that may be correlated. For example, some output configurations of the states of Markov chains may be more probable than others, and thus the random output may eventually appear non-uniform, which may be important to avoid when attempting to create pseudo-random numbers. Accordingly, disclosed embodiments may address the effect of the number and size of parallel Markov chains on the quality of pseudo-random output, as well as the maximum duration for running parallel Markov chains before detecting non-uniform output. Disclosed embodiments may implement the AP on a modern memory specification and technology node to provide 40 GB/s of high-quality random throughput per chip.

[0020] FIG. 1 illustrates exemplary system **100** for generating pseudo-random numbers. In some embodiments, system **100** may include computing device **104** to perform disclosed processes.

[0021] In some embodiments, device **104** may include processor **108**. Processor **108** may provide processing resources to perform disclosed processes. For example, processor **108** may generate a digitized finite state machine based on probabilistic data (e.g., range data and/or weight data).

[0022] In some embodiments, device **104** may include memory **110**. Memory **110** may store data and/or instructions for performing disclosed processes. In some embodiments, device **104** may include storage **106**. Storage **106** may store digitized instructions and computerized data. For example, storage **106** may include non-transitory computer-readable storage medium including instructions to perform disclosed processes. Device **104** may generate instructions in storage **106** that are transmitted to configure specialized device **102** to perform disclosed processes.

[0023] In some embodiments, device **104** may include I/O **112** (input-output interface). I/O **112** may connect to I/O Device(s) **114** and specialized device **102**. For example, device **104** may receive input (e.g., user input or network communication) from I/O Device(s) **114**. Device **104** may transmit programming instructions to specialized device **102**, as well as, receive computing results from specialized device **102**. For example, specialized device **102** may transmit pseudo-random numbers to device **104**.

[0024] Device **104** may connect to specialized device **102**. Specialized device **102** may be a specialized pattern-matching device for implementing digitized finite state machines. For example, specialized device **102** may be an Automata Processor, such as an Automata Processor PCIe board.

[0025] Specialized device **102** (e.g., an Automata Processor (AP)) may reproduce the power of a theoretical non-deterministic finite automata to non-deterministic parallelism. In this context, non-determinism may not imply stochastic behavior, but instead may denote an exploration of all possible parallel paths through an automata at once. For problems with large, combinatorially difficult search spaces, non-determinism may be an extremely powerful tool, enabling a fast, parallel exploration of an exponential number of problem instances.

[0026] Efficient implementations of non-deterministic finite automata in hardware may fall into two broad categories: specialized dynamically reconfigurable hardware for deterministic finite automata (DFA) and non-deterministic finite automata (NFA) execution, and static, circuit-based field-programmable gate array (FPGA) implementations. Specialized hardware to execute DFAs and NFAs may accelerate regular expression matching. However, existing architectures are application-specific and can only solve problems framed as regular expression matching. Static, circuit-based FPGA implementations of NFAs and DFAs may be much more flexible in their capabilities, but may suffer from density, scalability, and throughput limitations. Both specialized hardware and static logic solutions may not expose automata level programmability to the application developer, which may prevent the creation of automata that are either not convenient or even able to be expressed as regular expressions.

[0027] Specialized device **102** (e.g., an AP) may include a unique memory arrangement. For example, Micron's AP

may include unique, memory-derived architecture may take advantage of the bit-level parallelism inherent in synchronous dynamic random-access memory (SDRAM) arrays to gain improvements in state density over previous NFA and DFA implementations. In another example, Micron's AP may be configured using both pearl-compatible regular expressions (PCRE) and Automata Network Markup Language (ANML), which may offer programmers finegrained control over automata construction.

[0028] In some embodiments, specialized device **102** may include two AP cores that are combined to form an AP chip package and each core in the chip currently connects to the system via a shared double data rate type three (DDR3) interface. For example, 8 AP chips may be combined on a dual in-line memory module (DIMM) package, and up to 4-6 small outline dual in-line memory modules (SO-DIMMs) may be supported on a single PCIe accelerator board. Therefore, specialized device **102** may include a single AP board with a base configuration having 64 AP cores. In some embodiments, specialized device **102** may include an accelerator board with an Altera Stratix IV FPGA, which may include memory controllers and PCIe hardware to support AP DIMM modules. All STEs on an AP chip may be reconfigured in approximately 45 ms.

[0029] In some embodiments, STEs of specialized device **102** may trigger output. For example, when an STE on an AP chip reports, the AP may generate a report vector. Each report vector may be a bit-vector representation of all reporting STEs that activated at that particular cycle, and may contain up to 1,024 bits. Each chip may contain 6 reporting capable of exporting 1,024 output vectors in 1.8 ms. Therefore, a best-case upper-bound for the full AP output throughput may be approximately 437 MB/s per AP chip, or 14 GB/s per board.

[0030] The above metrics may be representative of first-generation AP architecture and implementation. Future AP system architectures may enable direct reads and writes to AP memories via a CPU's front-side bus, or other inter-processor interconnect, which may permit much lower AP reconfiguration times and much higher output throughput.

[0031] FIG. 2 illustrates an exemplary finite state machine for generating pseudo-random numbers.

[0032] A simple Markov chain that simulates an unfair coin toss with two states: Heads and Tails. Transition probabilities between these states are unfair meaning that the probability of transitioning to, or flipping, Heads is different than Tails.

[0033] In informal terms, Markov Chains are automata with probabilistic transitions between states. To be formally considered a Markov chain, transitions in the automaton may be stochastic processes (e.g., they occur with some probability), and respect the Markov property, which states that every probabilistic transition depends only on the current state, and is not influenced by memory of prior states. An example Markov Chain describing the behavior of tosses of an unfair coin is illustrated by diagram **200**.

[0034] Markov chains are defined by stochastic transition matrices which hold all transition probabilities from a start state (row) to a transition state (column). Each row of the transition matrix may be stochastic. For example, each stochastic row may add up to 1. The state may make some transition in each time step, even if it is to the current node.

[0035] A Markov chain implemented on the AP corresponding to the Markov chain in diagram **202**, with two "star

states" representing Heads and Tails. In an embodiment, "star states" may match on any character. For example, a "star state" may activate on any 8-bit input symbol, making the probability of transitioning to a "star state" from a previous state is 100%. Transition probabilities between these states are unfair and are modeled by dividing the possible input symbols [0-9] into random groups, proportional to the transition probabilities as those of diagram **200**. Diagram **202** may represent programming instructions for specialized device **102** (e.g., an AP).

[0036] AP automata may be made up of a directed graph of state transition elements (STEs), which can recognize an arbitrary character set of 8-bit symbols. An STE may "activate" when it recognizes the current input symbol and it is "enabled." An STE may be considered enabled when it is either configured to consume input from the input stream (a "start" STE), or an STE connected to it activated on the previous cycle. STEs can be configured to report on activation, which may produce a 1-bit output, analogous to accepting an input string in an NFA.

[0037] FIG. 3 illustrates an exemplary process for generating pseudo-random numbers in accordance with disclosed embodiments.

[0038] In step **302**, routine **300** may receive probabilistic definitions. System **100** may receive data indicating the parameters for pseudo-random number generation. For example, device **104** may receive digitized instructions describing the desired range of pseudo-random numbers to produce and/or the desired distribution for the pseudo-random numbers (e.g., the probabilistic transitions). System **100** may receive the state data and the weight data in the form of a stochastic transition matrix.

[0039] In some embodiments, probabilistic definitions may include the range of desired outputs. System **100** may receive a number of states for which random output is desired. For example, system **100** may receive input such as "2" when binary output is required, mimicking results for a coin flip. In another example, system **100** may receive "6" as input to mimic the roll of a six-sided die. Additional numbers of states may be used depending on the desired output. The range of desired output may further represent the output numbers desired to correspond to each of the states. For example, when there are two states to mimic a coin toss, probabilistic definitions may further detail that the two states should be labeled "0" and "1." In another example, a simulation of a six-sided die having six states, probabilistic definitions may indicate that the states should range from one ("1") to six ("6") to correspond to traditional numbers on a six-sided die.

[0040] In some embodiments, probabilistic definitions may include weight data. Weight data may indicate the desired probability or probabilistic distribution for each state. For example, for a fair coin toss (e.g., having an even or uniform distribution), each state (e.g., "0" and "1") would have equal probability (e.g., "0.5" for each). In the example of a fair six-sided die, each state would have a weight of one sixth (e.g., approximately "0.167"). In some embodiments, uneven (e.g., not uniform) or "unfair" distributions may be desired. For example, weight data may describe an unfair coin toss by indicating different weights for each state. In such an example, state "0" (e.g., "heads" for the coin) may have a probability of "0.9" while state "1" has a probability of "0.1". Weight data may assign any desired probability target so long as the weights total "1.0". In some embodi-

ments, process 300 may normalize the weights to “1.0” when the entered weights do not total “1.0”.

[0041] In step 304, routine 300 may generate a digitized finite state machine. For example, system 100 may generate a digitized finite state machine based on probabilistic definitions. To communicate the concept of probabilistic transitions and implement Markov chains on an AP, system 100 may map the probabilistic definitions to a digitized finite state machine. In some embodiments, system 100 may generate a Markov chain for the AP that utilizes an input symbol stream having uniformly distributed random symbols. Each Markov chain may be constructed using a stochastic transition matrix.

[0042] An example Markov chain for an unfair coin example is shown in FIG. 2. For example, diagrams 200 and 202, as shown, are based on the input symbols being within the character class [0-9]. In some embodiments, a single state out of all possible states may be chosen arbitrarily to act as the start state. In some embodiments, fully connected fair Markov chains having transitions to all states are equally likely may not need a randomly chosen start state, as steady state behavior may be reached after the first cycle. As shown in FIG. 2, the construction may take two cycles to generate an output, one to transition to a transition node, and another to transition to the star state. Other embodiments may modify the state machine to generate an output on every cycle by also setting a randomly selected transition node, along with an arbitrary state node, to act as a start state. In such an embodiment, one state node and one transition node may be active on any given cycle, which may act as a pipeline for two probabilistic transitions.

[0043] To construct a PRNG from a single Markov chain, process 300 may build a fair Markov chain of a predetermined number of states. For example, a two-state chain may produce a single bit output on every cycle. In other examples, any number of states may be used to construct a Markov chain as long as transitions to all states are equally likely. When the state output is in binary bits, the number of states in the Markov chain may be a power of two to ensure a uniformly distributed output bits. On every cycle, a single chain may report which state it randomly transitioned to, which may emit output corresponding to $\log_2(\text{states})$ bits of random output per machine per cycle. In some embodiments, multiple Markov chains may be used. Additional Markov chains may be added, and their output may be interleaved, to increase the total amount of pseudo-random output relative to the input symbols used to drive random transitions. For example, a single 2-state Markov chain may emit a single random bit per random input byte, while eight 2-state chains create the same amount of random output as input.

[0044] In step 306, routine 300 may provide, to a specialized pattern-matching device, programmable instructions to implement the digitized finite state machine on the specialized pattern-matching device.

[0045] In some embodiments, specialized device 102 (e.g., an AP) may be programmed using automata, such as those described using a directed graph of state transition elements (STEs) corresponding to states of a digitized finite state machine, which can recognize an arbitrary character set of 8-bit symbols. An STE may “activate” when it (1) recognizes the current input symbol and (2) it is “enabled.” An STE may be considered enabled when it is either configured to consume input from the input stream (a “start”

STE), or a STE connected to it activated on the previous cycle. STEs may be configured to report on activation, producing a 1-bit output, similar to accepting an input string in a NFA. Device 104 may receive such output using I/O 112.

[0046] Specialized device 102 (e.g., an AP) may implement STEs using 256-bit memory columns AN Ded with an enable signal. Each 256-bit column vector may represent a character set of 256 possible 8-bit characters that this STE could recognize. Any character, supplied as a row address will then force all STE columns that recognize that character set to read out a 1 in parallel. For example, the Kleene star operator would simply fill all bit rows in the STE column with 1s. Thus, an STE may be capable of recognizing an arbitrary character set of possible input symbols on every cycle. If a column reads a “1” and the STE is enabled, the STE may activate and send its output signal to the routing matrix. The routing matrix may allow STEs to connect to and enable any other STEs within the same AP core, and may be pre-configured (placed and routed) based on the compiled AP application and automaton design. Columns of STEs are organized into blocks and a number of blocks makes up an AP core. Because the routing matrix only exists within cores, STEs may be prevented from enabling other STEs across cores. In the current generation AP hardware, a block may contain 256 STEs, 32 of which can report. AP cores may contain 96 blocks, offering a total of 24,576 STEs per core. The first generation AP hardware may operate at a constant frequency of 133 MHz, consuming a symbol every 7.5 ns, thus providing a throughput of 133 MB/s per core.

[0047] In step 308, device 104 may transmit a pseudo-random bit stream to a digitized finite state machine. Device 104 may produce a random stream of input with a predetermined level of randomness. For example, device 104 may transmit a stream of random characters to the PRNG of specialized device 102. Specialized device 102 may receive and process the random input on the digitized finite state machine. For example, in the example illustrated in FIG. 2, the digitized finite state machine may receive a stream of characters ranging from zero to nine. Specialized device 102 (e.g., an AP) may process each digit of input by transitioning to the appropriate state of the digitized finite state machine. Depending on the configuration of the digitized finite state machine, the state transition may include reporting functionality.

[0048] In step 310, routine 300 may include device 104 receiving pseudo-random output from a specialized pattern-matching device. In some embodiments, specialized device 102 may process a bit stream using a digitized finite state machine (e.g., a Markov chain) to produce pseudo-random output. The digitized finite state machine may include reporting functionality to produce output based on the current state transition of the digitized finite state machine. For example, the state transition may include reporting instructions that may cause an AP to generate pseudo-random output. The AP may transmit the reporting output to device 104.

[0049] Because only 32 memory elements (MEs) out of 256 in an AP block are capable of reporting, each Markov chain may be limited by either reporting elements or total STEs per block. An N-state chain requires N reporting elements, thus system 100 may instantiate a maximum of 16, 8, and 4 chains per block for 2, 4, and 8-state chains on an AP, respectively. An N-state chain may need N^2+N STEs,

thus system 100 may instantiate a maximum of 42, 12, and 3 chains per block for 2, 4, and 8-state chains respectively. While reporting elements may limit how many 2- and 4-state chains an AP may fit onto a given block, the total STEs may limit the number of 8-state chains. Given that an AP core has 96 blocks, 2 and 4-state chains may provide a 384× increase in throughput, while 8-state chains may provide a 288× increase in throughput per input symbol.

[0050] FIG. 4 illustrates a block diagram of an exemplary machine 400 upon which one or more embodiments (e.g., discussed methodologies) can be implemented (e.g., run). Examples of machine 400 can include logic, one or more components, circuits (e.g., modules), or mechanisms. Circuits are tangible entities configured to perform certain operations. In an example, circuits can be arranged (e.g., internally or with respect to external entities such as other circuits) in a specified manner. In an example, one or more computer systems (e.g., a standalone, client or server computer system) or one or more hardware processors (processors) can be configured by software (e.g., instructions, an application portion, or an application) as a circuit that operates to perform certain operations as described herein. In an example, the software can reside (1) on a non-transitory machine readable medium or (2) in a transmission signal. In an example, the software, when executed by the underlying hardware of the circuit, causes the circuit to perform the certain operations.

[0051] In an example, a circuit can be implemented mechanically or electronically. For example, a circuit can comprise dedicated circuitry or logic that is specifically configured to perform one or more techniques such as discussed above, including, for example, a special-purpose processor, a field programmable gate array (FPGA), or an application-specific integrated circuit (ASIC). In an example, a circuit can comprise programmable logic (e.g., circuitry, as encompassed within a general-purpose processor or other programmable processor) that can be temporarily configured (e.g., by software) to perform the certain operations. It will be appreciated that the decision to implement a circuit mechanically (e.g., in dedicated and permanently configured circuitry), or in temporarily configured circuitry (e.g., configured by software) can be driven by cost and time considerations.

[0052] Accordingly, the term “circuit” is understood to encompass a tangible entity, be that an entity that is physically constructed, permanently configured (e.g., hardwired), or temporarily (e.g., transitorily) configured (e.g., programmed) to operate in a specified manner or to perform specified operations. In an example, given a plurality of temporarily configured circuits, each of the circuits need not be configured or instantiated at any one instance in time. For example, where the circuits comprise a general-purpose processor configured via software, the general-purpose processor can be configured as respective different circuits at different times. Software can accordingly configure a processor, for example, to constitute a particular circuit at one instance of time and to constitute a different circuit at a different instance of time.

[0053] In an example, circuits can provide information to, and receive information from, other circuits. In this example, the circuits can be regarded as being communicatively coupled to one or more other circuits. Where multiple of such circuits exist contemporaneously, communications can be achieved through signal transmission (e.g., over appro-

priate circuits and buses) that connect the circuits. In embodiments in which multiple circuits are configured or instantiated at different times, communications between such circuits can be achieved, for example, through the storage and retrieval of information in memory structures to which the multiple circuits have access. For example, one circuit can perform an operation and store the output of that operation in a memory device to which it is communicatively coupled. A further circuit can then, at a later time, access the memory device to retrieve and process the stored output. In an example, circuits can be configured to initiate or receive communications with input or output devices and can operate on a resource (e.g., a collection of information).

[0054] The various operations of method examples described herein can be performed, at least partially, by one or more processors that are temporarily configured (e.g., by software) or permanently configured to perform the relevant operations. Whether temporarily or permanently configured, such processors can constitute processor-implemented circuits that operate to perform one or more operations or functions. In an example, the circuits referred to herein can comprise processor-implemented circuits.

[0055] Similarly, the methods described herein can be at least partially processor implemented. For example, at least some of the operations of a method can be performed by one or more processors or processor-implemented circuits. The performance of certain of the operations can be distributed among the one or more processors, not only residing within a single machine, but deployed across a number of machines. In an example, the processor or processors can be located in a single location (e.g., within a home environment, an office environment or as a server farm), while in other examples the processors can be distributed across a number of locations.

[0056] The one or more processors can also operate to support performance of the relevant operations in a “cloud computing” environment or as a “software as a service” (SaaS). For example, at least some of the operations can be performed by a group of computers (as examples of machines including processors), with these operations being accessible via a network (e.g., the Internet) and via one or more appropriate interfaces (e.g., Application Program Interfaces (APIs)).

[0057] Exemplary embodiments (e.g., apparatus, systems, or methods) can be implemented in digital electronic circuitry, in computer hardware, in firmware, in software, or in any combination thereof. Example embodiments can be implemented using a computer program product (e.g., a computer program, tangibly embodied in an information carrier or in a machine readable medium, for execution by, or to control the operation of, data processing apparatus such as a programmable processor, a computer, or multiple computers).

[0058] A computer program can be written in any form of programming language, including compiled or interpreted languages, and it can be deployed in any form, including as a stand-alone program or as a software module, subroutine, or other unit suitable for use in a computing environment. A computer program can be deployed to be executed on one computer or on multiple computers at one site or distributed across multiple sites and interconnected by a communication network.

[0059] In an example, operations can be performed by one or more programmable processors executing a computer

program to perform functions by operating on input data and generating output. Examples of method operations can also be performed by, and an exemplary apparatus can be implemented as, special purpose logic circuitry (e.g., a field programmable gate array (FPGA) or an application-specific integrated circuit (ASIC)).

[0060] The computing system can include clients and servers. A client and server are generally remote from each other and generally interact through a communication network. The relationship of client and server arises by virtue of computer programs running on the respective computers and having a client-server relationship to each other.

[0061] In embodiments deploying a programmable computing system, it will be appreciated that both hardware and software architectures require consideration. Specifically, it will be appreciated that the choice of whether to implement certain functionality in permanently configured hardware (e.g., an ASIC), in temporarily configured hardware (e.g., a combination of software and a programmable processor), or a combination of permanently and temporarily configured hardware can be a design choice. Below are set out hardware (e.g., machine 400) and software architectures that can be deployed in exemplary embodiments. In an example, the machine 400 can operate as a standalone device or machine 400 can be connected (e.g., networked) to other machines. In a networked deployment, machine 400 can operate in the capacity of either a server or a client machine in server-client network environments. In an example, machine 400 can act as a peer machine in peer-to-peer (or other distributed) network environments. Machine 400 can be a personal computer (PC), a tablet PC, a set-top box (STB), a Personal Digital Assistant (PDA), a mobile telephone, a web appliance, a network router, switch or bridge, or any machine capable of executing instructions (sequential or otherwise) specifying actions to be taken (e.g., performed) by machine 400. Further, while only a single machine 400 is illustrated, the term “machine” shall also be taken to include any collection of machines that individually or jointly execute a set (or multiple sets) of instructions to perform any one or more of the methodologies discussed herein.

[0062] Exemplary machine (e.g., computer system) 400 can include a processor 402 (e.g., a central processing unit (CPU), a graphics processing unit (GPU), or both), a main memory 404 and a static memory 406, some or all of which can communicate with each other via a bus 408. Machine 400 can further include a display unit 410, an alphanumeric input device 412 (e.g., a keyboard), and a user interface (UI) navigation device 414 (e.g., a mouse). In an example, the display unit 410, input device 417 and UI navigation device 414 can be a touch screen display. Machine 400 can additionally include a storage device (e.g., drive unit) 416, a signal generation device 418 (e.g., a speaker), a network interface device 420, and one or more sensors 421, such as a global positioning system (GPS) sensor, compass, accelerometer, or other sensor.

[0063] Storage device 416 can include a machine readable medium 422 on which is stored one or more sets of data structures or instructions 424 (e.g., software) embodying or utilized by any one or more of the methodologies or functions described herein. Instructions 424 can also reside, completely or at least partially, within main memory 404, within static memory 406, or within processor 402 during execution thereof by machine 400. In an example, one or any

combination of processor 402, main memory 404, static memory 406, or storage device 416 can constitute machine readable media.

[0064] While machine readable medium 422 is illustrated as a single medium, the term “machine readable medium” can include a single medium or multiple media (e.g., a centralized or distributed database, and/or associated caches and servers) that configured to store the one or more instructions 424. The term “machine readable medium” can also be taken to include any tangible medium that is capable of storing, encoding, or carrying instructions for execution by the machine and that cause the machine to perform any one or more of the methodologies of the present disclosure or that is capable of storing, encoding or carrying data structures utilized by or associated with such instructions. The term “machine readable medium” can accordingly be taken to include, but not be limited to, solid-state memories, and optical and magnetic media. Specific examples of machine readable media can include non-volatile memory, including, by way of example, semiconductor memory devices (e.g., Electrically Programmable Read-Only Memory (EPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM)) and flash memory devices; magnetic disks such as internal hard disks and removable disks; magneto-optical disks; and CD-ROM and DVD-ROM disks.

[0065] Instructions 424 can further be transmitted or received over a communications network 426 using a transmission medium via the network interface device 420 utilizing any one of a number of transfer protocols (e.g., frame relay, IP, TCP, UDP, HTTP, etc.). Exemplary communication networks can include a local area network (LAN), a wide area network (WAN), a packet data network (e.g., the Internet), mobile telephone networks (e.g., cellular networks), Plain Old Telephone (POTS) networks, and wireless data networks (e.g., IEEE 802.11 standards family known as Wi-Fi®, IEEE 802.16 standards family known as WiMax®, peer-to-peer (P2P) networks, among others. The term “transmission medium” shall be taken to include any intangible medium that is capable of storing, encoding or carrying instructions for execution by the machine, and includes digital or analog communications signals or other intangible medium to facilitate communication of such software.

[0066] Various embodiments or aspects of the disclosure, for example, can be implemented as software in a computing device, or alternatively, on hardware. An exemplary computing device in which disclosed embodiments, or a portion thereof, may be implemented is schematically illustrated in FIGS. 5 and 6.

[0067] Referring to FIG. 5, in its most basic configuration, device 500 may include at least one Processing unit 508 and Memory 504. Depending on the exact configuration and type of computing device, Memory 504 can be volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some combination of the two. Additionally, device 500 may also have other features and/or functionality. For example, the device could also include additional removable and/or non-removable storage including, but not limited to, magnetic or optical disks or tape, as well as writable electrical storage media. Such additional storage is the figure by removable storage 506 and non-removable storage 510. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any method or technology for storage of information such as

computer readable instructions, data structures, program modules or other data. The memory, the removable storage and the non-removable storage are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology CDROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can be accessed by the device. Any such computer storage media may be part of, or used in conjunction with, the device.

[0068] The device may also contain one or more communications connections 512 that allow the device to communicate with other devices (e.g. other computing devices). The communications connections carry information in a communication media. Communication media typically embodies computer readable instructions, data structures, program modules or other data in a modulated data signal such as a carrier wave or other transport mechanism and includes any information delivery media. The term "modulated data signal" means a signal that has one or more of its characteristics set or changed in such a manner as to encode, execute, or process information in the signal. By way of example, and not limitation, communication medium includes wired media such as a wired network or direct-wired connection, and wireless media such as radio, RF, infrared and other wireless media. As discussed above, the term computer readable media as used herein includes both storage media and communication media.

[0069] In addition to a stand-alone computing machine, embodiments of the invention can also be implemented on a network system comprising a plurality of computing devices that are in communication with a networking means, such as a network with an infrastructure or an ad hoc network. The network connection can be wired connections or wireless connections.

[0070] As a way of example, FIG. 6 illustrates a network system 600 in which embodiments of the invention can be implemented. In this example, the network system may include computer 602 (e.g., a network server), network connection 606 (e.g. wired and/or wireless connections), computer terminal 604, and PDA (e.g. a smartphone) 608 (or other handheld or portable device, such as a cell phone, laptop computer, tablet computer, GPS receiver, mp3 player, handheld video player, pocket projector, etc. or handheld devices (or non-portable devices) with combinations of such features). The embodiments of the invention can be implemented in any of the devices of the system.

[0071] For example, execution of the instructions or other desired processing can be performed on the same computing device that is any one of 602, 604, and 608. Alternatively, an embodiment of the invention can be performed on different computing devices of the network system. For example, certain desired or required processing or execution can be performed on one of the computing devices of the network (e.g. server 602), whereas other processing and execution of the instruction can be performed at another computing device (e.g. terminal 604) of the network system, or vice versa. In fact, certain processing or execution can be performed at one computing device (e.g. server 602); and the other processing or execution of the instructions can be performed at different computing devices that may or may not be networked. For example, the certain processing can

be performed at terminal 604, while the other processing or instructions are passed to device 608 where the instructions are executed. This scenario may be of particular value especially when the PDA device, for example, accesses to the network through computer terminal 604 (or an access point in an ad hoc network). For another example, software to be protected can be executed, encoded or processed with one or more embodiments of the invention. The processed, encoded or executed software can then be distributed to customers. The distribution can be in a form of storage media (e.g. disk) or electronic copy.

[0072] Practice of an aspect of an embodiment (or embodiments) of the invention is presented herein for illustration only and should not be construed as limiting the invention in any way.

[0073] An approach of the present invention systems and designs and optimization system and techniques may be based on the tools, programs and operating systems as discussed throughout this disclosure, such techniques can be applied to various hardware, tools, operating systems, virtual machines, parallel virtual machines (PVMs), or executable formats.

What is claimed is:

1. A method for generating pseudo-random numbers, comprising:
 - receiving, by at least one processor, range data indicating a range of numbers;
 - generating, based on the range data and by the at least one processor, a digitized finite state machine configured to produce pseudo-random output within the range of numbers;
 - providing, by the at least one processor to a specialized pattern-matching device, programmable instructions to implement the digitized finite state machine on the specialized pattern-matching device;
 - transmitting, by the at least one processor to the specialized pattern-matching device, a pseudo-random bit stream for processing by the digitized finite state machine; and
 - receiving, by the at least one processor from the specialized pattern-matching device, pseudo-random output from the digitized finite state machine.
2. The method of claim 1, wherein the specialized pattern-matching device is an Automata Processor PCIe board.
3. The method of claim 1, wherein the digitized finite state machine includes a number of states corresponding to the range of numbers.
4. The method of claim 1, further comprising:
 - receiving, by the at least one processor, weight data indicating a distribution for the range of numbers;
 wherein the digitized finite state machine includes probabilistic transitions corresponding to the distribution for the range of numbers.
5. The method of claim 4, wherein
 - the weight data indicates that the distribution should be uniform; and
 - the probabilistic transitions each have an equal weight, based on the weight data indicating that the distribution should be uniform.
6. The method of claim 1, wherein the digitized finite state machine is formed from multiple Markov chains.
7. A non-transitory computer-readable storage medium for generating pseudo-random numbers, the computer-read-

able storage medium including instructions that when executed by at least one processor, cause the at least one processor to

receive, by the at least one processor, range data indicating a range of numbers;

generate, based on the range data and by the at least one processor, a digitized finite state machine configured to produce pseudo-random output within the range of numbers;

provide, by the at least one processor to a specialized pattern-matching device, programmable instructions to implement the digitized finite state machine on the specialized pattern-matching device;

transmit, by the at least one processor to the specialized pattern-matching device, a pseudo-random bit stream for processing by the digitized finite state machine; and receive, by the at least one processor from the specialized pattern-matching device, pseudo-random output from the digitized finite state machine.

8. The computer-readable storage medium of claim 7, wherein the specialized pattern-matching device is an Automata Processor PCIe board.

9. The computer-readable storage medium of claim 7, wherein the digitized finite state machine includes a number of states corresponding to the range of numbers.

10. The computer-readable storage medium of claim 7, wherein the instructions further configure the at least one processor to:

receive, by the at least one processor, weight data indicating a distribution for the range of numbers;

wherein the digitized finite state machine includes probabilistic transitions corresponding to the distribution for the range of numbers.

11. The computer-readable storage medium of claim 10, wherein

the weight data indicates that the distribution should be uniform; and

the probabilistic transitions each have an equal weight, based on the weight data indicating that the distribution should be uniform.

12. The computer-readable storage medium of claim 7, wherein the digitized finite state machine is formed from multiple Markov chains.

13. A computing apparatus for generating pseudo-random numbers, the computing apparatus comprising:

at least one processor; and

a memory storing instructions that, when executed by the at least one processor, cause the at least one processor to:

receive, by the at least one processor, range data indicating a range of numbers;

generate, based on the range data and by the at least one processor, a digitized finite state machine configured to produce pseudo-random output within the range of numbers;

provide, by the at least one processor to a specialized pattern-matching device, programmable instructions to implement the digitized finite state machine on the specialized pattern-matching device;

transmit, by the at least one processor to the specialized pattern-matching device, a pseudo-random bit stream for processing by the digitized finite state machine; and receive, by the at least one processor from the specialized pattern-matching device, pseudo-random output from the digitized finite state machine.

14. The computing apparatus of claim 13, wherein the specialized pattern-matching device is an Automata Processor PCIe board.

15. The computing apparatus of claim 13, wherein the digitized finite state machine includes a number of states corresponding to the range of numbers.

16. The computing apparatus of claim 13, wherein the instructions further configure the apparatus to:

receive, by the at least one processor, weight data indicating a distribution for the range of numbers;

wherein the digitized finite state machine includes probabilistic transitions corresponding to the distribution for the range of numbers.

17. The computing apparatus of claim 16, wherein the weight data indicates that the distribution should be uniform; and

the probabilistic transitions each have an equal weight, based on the weight data indicating that the distribution should be uniform.

18. The computing apparatus of claim 13, wherein the digitized finite state machine is formed from multiple Markov chains.

* * * * *

Appendix E

APPRNG Python Implementation

5.1 Moore Machine Simulator

```
class MooreMachine:
    #simulates a Moore Machine
    #automaton that outputs its state label after each transition

    def __init__(self, Q, dFunc, start):
        #Q: list of states
        #d_func: transition function
            #(source, destination, input character)
        #start: unique start state
        self.start=start
        self.output = []
        self.dFunc = dFunc
        self.states={}
        for state in Q:
            self.states[state]={}
        for trans in dFunc:
            self.states[trans[0]][trans[2]]=trans[1]
        self.current=start
```

```

def step(self, input_char):
    #advances computation by one character
    self.current=self.states[self.current][input_char]
    self.output.append(self.current)
    return self.current

def sim(self, inString):
    #runs the machine on entire string
    self.reset()
    walk = []
    for c in inString:
        walk.append(self.step(c))
    return walk

def reset(self):
    self.current = self.start
    self.output = []

```

5.2 APPRNG Creation

```

def apprngBuilder(n):
    #builds a APPRNG markov chain of n states
    Q = [] #state set
    f = [] #(source, destination, input character) list
    for i in range(n):
        Q.append(i)
    R = Q
    for i in range(n):
        rand.shuffle(R)
        for j in range(n):
            f.append((i, j, R[j]))
    g = MooreMachine(Q, f, 1)
    return g

```

```

def apprngSim(m, n, t):
    #m machines
    #n states each
    #run on t symbols
    markov_chains = []
    output = []
    sigma = []
    inputs = []
    for i in range(n):
        sigma.append(i)
    for i in range(m):
        markov_chains.append(apprngBuilder(n))
    for i in range(t):
        for j in range(m):
            x = rand.choice(sigma)
            inputs.append(x)
            output.append(markov_chains[j].step(x))
    return output

```

5.3 Sample Usage

```

def ToBits(lst):
    #A utility function for converting integers to ASCII binary
    output = []
    curr_num = ""
    for i in lst:
        if i == 0:
            curr_num = curr_num + '0'
        while i > 0:
            if i % 2 == 0:
                curr_num = curr_num + '0'

```

```

        else:
            curr_num = curr_num + '1'
            i = math.floor(i/2)
            output.append(curr_num[::-1])
            curr_num = ""
string_length = 0
for i in output:
    if string_length < len(i):
        string_length = len(i)
for i in range(len(output)):
    while len(output[i]) < string_length:
        output[i] = "0" + output[i]
return output

def apprngSimString(m, n, t):
    #runs the PRG
    #then converts the integer state labels to ASCII binary
    out = ToBits(apprngSim(m, n, t))
    string_out = ""
    for i in out:
        string_out = string_out + i
    return string_out

def apprngGenRand(m, n, t, output_bits):
    #handles the file writing repeated invocations of the function
    #m: number of machines
    #n: number of states in each machine
    #t: number of inputs to all machines
    #output_bits: desired number of output bits
    iters = math.ceil(output_bits/(t*m*math.ceil(math.log(n, 2))))
    accumulator = ""

```

```
for i in range(iters):
    accumulator = accumulator + apprngSimString(m, n, t)
f = open(str(m)+"Machine_"+str(n)+"State_"+str(t)+"Input.txt",'w')
f.write(accumulator)
f.close()

# 8 machines
# 4 states
# 5000 symbols before reconfiguration
# 100,000 total output bits
# result written to 8Machine_4State_5000Input.txt
apprngGenRand(8,4,5000,100000)
```


Appendix F

AP Bloom Filter Python Implementation

6.1 Finite Automata Simulator

```
class FSA:
    #simulates a Deterministic Finite State Automaton

    def __init__(self, q, dFunc, start, finals):
        #Q: list of states
        #d_func: transition function
            #(source, destination, input character)
        #start: unique start state
        #finals: set of final states
        self.start=start
        self.finals = finals
        self.dFunc = dFunc
        self.states={}
        for state in q:
            self.states[state]={}
        for trans in dFunc:
            self.states[trans[0]][trans[2]]=trans[1]
```

```

        self.current=start

    def step(self , inChar):
        #advances computation by one character
        self.current=self.states[ self.current ][ inChar]

    def sim(self , inString):
        #runs the machine on entire string
        for c in inString:
            self.step(c)
        status=self.current in self finals
        self.current=self.start
        return status

```

6.2 Bloom Filter Data Structure Implementation

```

class Bloom_Filter:
    def __init__(self , sizeV , sizeAlph , theSet):
        self.sizeV = sizeV
        self.numHash = (math.log(2) * sizeV) / len(theSet)
        self.sizeAlph = sizeAlph
        self.sizeSet = len(theSet)
        self.numMachines = 2 * self.sizeV * math.log(2)
        self.chains = self.buildChains()
        self.err = self.setup(theSet)
        self.probAccept = probAccept(self.numHash, self.sizeV)

    def buildChains(self):
        #build a two-state MC
        #acceptance probability based on size_v and num_hash
        bloom = []
        probAccept = probAccept(self.numHash, self.sizeV)

```

```

        for i in range((int)(round(self.numMachines))):
            bloom.append(self.genMC(probAccept))
        return bloom

def genMC(self, prob):
    #build a 2-state automaton which accepts with probability prob
    #sizeAlph is the number of symbols in the alphabet
    if prob >= 1: #ill-defined will cause infinite loop
        return False
    sizeAlph = self.sizeAlph
    machineQ = [0, 1]
    machineD = []
    machineStart = 0
    machineFinals = [1]
    #to generate the transition function:
    #pick a random set of size p*sizeAlph w/o replacement
    trans0to1 = []
    trans1to1 = []
    for x in range((int)(sizeAlph*prob)):
        i = rand.randint(0, sizeAlph-1)
        j = rand.randint(0, sizeAlph-1)
        while i in trans0to1: #draw without replacement
            i = rand.randint(0, sizeAlph-1)
        while j in trans1to1:
            j = rand.randint(0, sizeAlph-1)
        trans0to1.append(i)
        trans1to1.append(j)
    for i in range(sizeAlph):
        if i in trans0to1:
            machineD.append((0, 1, i))
        else:
            machineD.append((0, 0, i))

```

```

        if i in trans1to1:
            machineD.append((1,1,i))
        else:
            machineD.append((1,0,i))
    fsa = FSA(machineQ, machineD, machineStart, machineFinals)
    return fsa

def setup(self, theSet):
    #remove all chains which accept something from the_set
    #return probability of false positive error
    for s in theSet:
        self.chains = filter(lambda c: not c.sim(s), self.chains)
    return self.p_error(len(theSet))

def p_error(self, sizeSet):
    #calculates the probability of a false positive
    numHash = (float)self.numHash
    sizeV = (float)self.sizeV
    p = math.pow(1-math.exp(-numHash*sizeSet/sizeV),numHash)
    return p

def query(self, x):
    #return true if all chains reject
    allReject = True
    for chain in self.chains:
        if chain.sim(x):
            allReject = False
    return allReject

```

6.3 Experiment Implementation

```

def random_set_gen(str_length, set_length, size_alpha):
    output_list = []

```

```

    for i in range(int(set_length)):
        output_list.append(random_str_gen(str_length, size_alph))
    return output_list

def random_str_gen(str_length, size_alph):
    out = []
    for i in range(str_length):
        out.append(rand.randint(0, size_alph - 1))
    return out

def comp_error(bfilter, input_set, bloom_set):
    error_count = 0
    #add true positives
    # error rate: false positives / true negatives
    true_positives = 0
    for i in input_set:
        q = bfilter.query(i)
        if i in bloom_set:
            true_positives = true_positives + 1
        if q and (i not in bloom_set):
            error_count += 1
        if not q and (i in bloom_set):
            print("this should never print ")
    #(expected errors, actual errors)
    return bfilter.err*(len(input_set)-true_positives), error_count

def testGen(nTrls, mTrls, lTrls, reps):
    alph = 512
    tot = []
    for t in [(n,m,l) for n in nTrls for m in mTrls for l in lTrls]:
        results = []
        for r in range(reps):

```

```

        bSet = random_set_gen(t[2], t[0]*t[1], alph)
        bloom = Bloom_Filter(t[1], alph, bSet)
        qSet = random_set_gen(t[2], 1000, alph)
        res = comp_error(bloom, qSet, bSet)
        results.append(res)
        tot.append(res)
    totals = reduce(lambda x, y: (x[0]+y[0], x[1]+y[1]), tot, (0.0,0.0))
    totals = (totals[0]/len(tot), totals[1]/len(tot))
    print "trial: ALL"
    print "averages: " + (str)(totals)

def test():
    nTrials = [.05, 0.1, 0.2, 0.5] #varies input set size
    mTrials = range(100, 500, 100)
    lTrials = range(1,50,7)
    reps = 10
    testGen(nTrials, mTrials, lTrials, reps)
    testGen([0.5], [500], lTrials, reps)
    testGen(nTrials, [500], [8], reps)
    testGen([0.5], mTrials, [8], reps)

test()

```

Appendix G

PhD Defense Presentation Video and Slides

The PhD defense presentation of this dissertation took place on July 31, 2017, at the University of Virginia in Charlottesville.

The video of this presentation can be viewed at:

<https://www.youtube.com/watch?v=GP2rm0z3ebI>

The PDF version of this dissertation is available at:

http://www.cs.virginia.edu/~njb2b/Brunelle_phdDissertation_UVACS_2017.pdf

The PowerPoint slides of this dissertation defense are available at:

http://www.cs.virginia.edu/~njb2b/Brunelle_phdDefense_UVACS_July2017.pdf.

The PDF version of the dissertation defense slides are available at:

http://www.cs.virginia.edu/~njb2b/Brunelle_phdDefense_UVACS_July2017.pptx

The dissertation defense slides are also reproduced below as follows.

Super-Scalable Algorithms

PhD Dissertation Defense

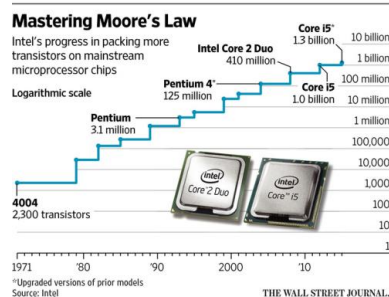
[Nathan Brunelle](#)

July 31, 2017

Video of this presentation viewable at <https://www.youtube.com/watch?v=GP2rmOz3ebi>
Dissertation PDF: http://www.cs.virginia.edu/~njb2b/Brunelle_phdDissertation_UVACS_2017.pdf
Slides PDF: http://www.cs.virginia.edu/~njb2b/Brunelle_phdDefense_UVACS_July2017.pdf
Slides PPT: http://www.cs.virginia.edu/~njb2b/Brunelle_phdDefense_UVACS_July2017.pptx

1

Overview



2016: Intel announced that Moore's Law is slowing down

Thesis: "Life-extension" strategies for Moore's Law:

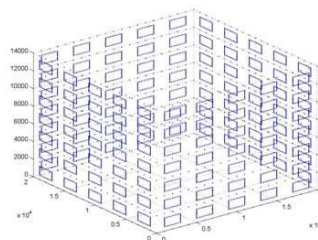
- Compression-aware algorithms
- Automata-based hardware accelerators

2

Compression Aware Algorithms



A real world building



Corresponding highly compressible CAD model

3

Compression-Aware Benefits

- Compression's size grows more slowly than the volume of the data it represents

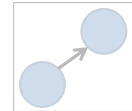


- Less data for the algorithm to manage



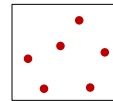
4

Data Domains



- Graph
 - Literature is on compression, not on algorithms
- Text
 - Well-studied
 - Impossibility result
- Geometric
 - Literature sparse on compression
 - Algorithmically-aware compressions

It was the best of times
It was the worst of times...



5

Compression-aware Sorting

- Context-free grammar compression
 - Set of **terminals**, **variables**, **substitution rules**
 - Begin with start variable
 - apply rules until no variables remain

Grammar

$A_0 \rightarrow aA_1A_2A_3$
 $A_1 \rightarrow ab$
 $A_2 \rightarrow A_1b$
 $A_3 \rightarrow A_2b$

Parsing

A_0
 $aA_1A_2A_3$
 $aabA_1bA_2b$
 $aababbA_1bb$
 $aababbabbbb$

6

Compression-aware Sorting

- Compress list as a string of CSV
- Requires $\Omega(n \log n)$

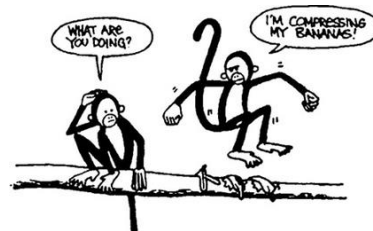
$A_0 \rightarrow A_1 A_2 A_3 \dots A_n$ Permute these
 $A_1 \rightarrow \dots,$ to permute list
 $A_2 \rightarrow \dots,$ List is arbitrarily compressible
 $A_3 \rightarrow \dots,$
 \dots

\Rightarrow Compression should be **matched** to algorithm

7

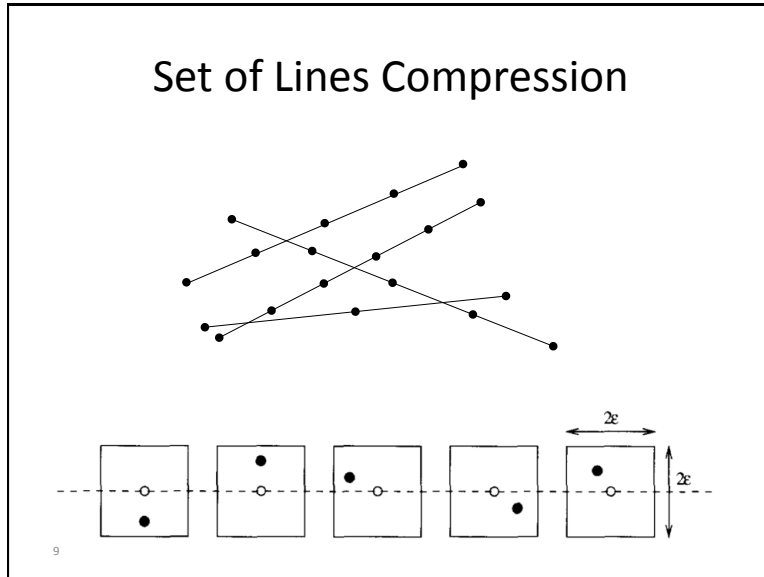
Algorithmically-Aware Compressions

- Dual of compression-aware algorithms
- Compressions designed for faster algorithms
- Focus on geometric data



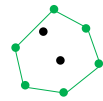
8

Set of Lines Compression



Algorithms on Set of Lines Data

- Convex Hull
 - $O(CH(\#Lines))$



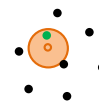
- Range Searches
 - Manhattan
 - $O(\#Lines)$



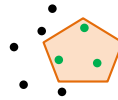
- Nearest Neighbor
 - $O(\#Lines)$



- Euclidean
 - $O(\#Lines)$

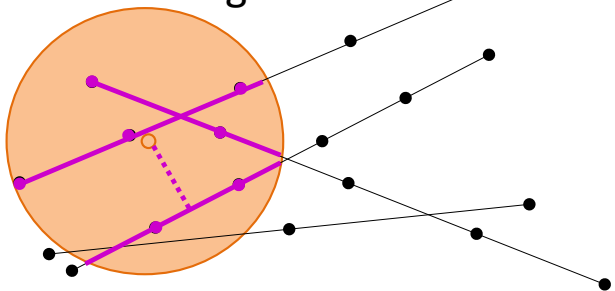


- Polytope Membership
 - $O(\#Lines \cdot \#faces)$



10

Range Search



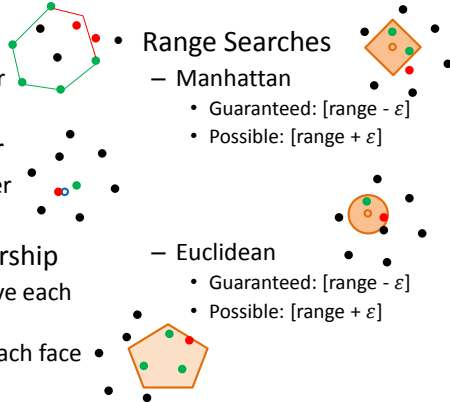
- Experimental results:
 - Highly compressible data has same run time as KD-Trees
 - Save 10 min. on decompression/data structure construction
 - Biggest advantage results returned in compressed format
 - Up to 300x speedups for our experiments

11

Lossy vs Lossless

- Convex Hull
 - Unbounded error
- Nearest Neighbor
 - May be 2ϵ farther
- Polytope Membership
 - Guaranteed: move each face in by ϵ
 - Possible: move each face out by ϵ

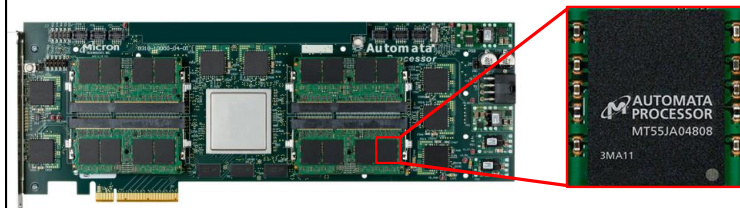
- Range Searches
 - Manhattan
 - Guaranteed: $[\text{range} - \epsilon]$
 - Possible: $[\text{range} + \epsilon]$
 - Euclidean
 - Guaranteed: $[\text{range} - \epsilon]$
 - Possible: $[\text{range} + \epsilon]$



12

Coprocessing

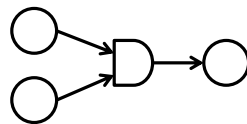
- Use clever hardware and algorithms to accelerate computation
- Solution: restrictive yet efficient hardware
 - GPUs
 - **Automata Processor**



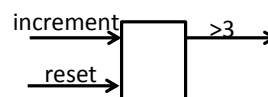
Micron Automata Processor

- Simulates NFAs in hardware
- Other utilities:

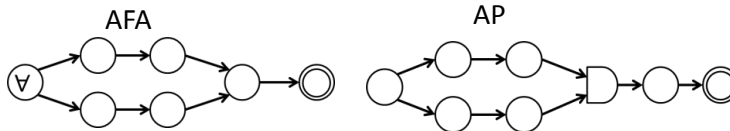
Boolean logic gates



Threshold counters



Eliminating Logic gates



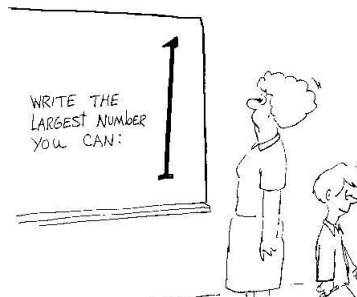
- Theorem:
 - Any Machine with Boolean gates can be converted to an Alternating Finite State Automaton (AFA)
- Corollary:
 - AP machines accept exactly the regular languages

15

Relation to Other Architectures

- Corollary:
 - Any AP machine can be simulated in log time with polynomial parallel Turing machines
- Space Cost:

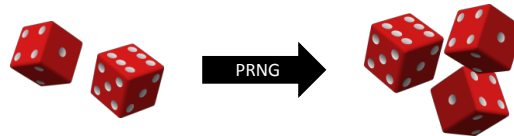
$$-(2^{2^k})^{2^{2^k}}$$



16

Pseudorandom Number Generators

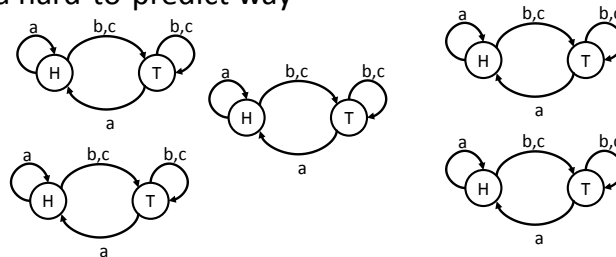
- Few bits of random input, more bits of “random-looking” output
- Formally:
 - $f: \{0,1\}^n \rightarrow \{0,1\}^m$ is a PRNG for $n < m$ if there is no polynomial-time algorithm which can distinguish the output distribution from uniform



17

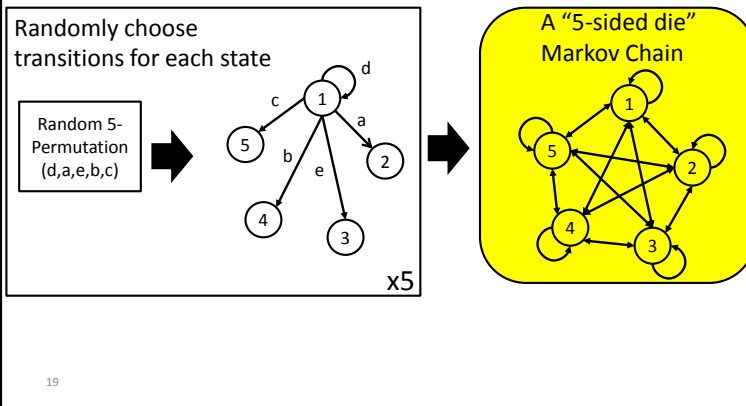
PRNG for APs

- Idea: Simulate many independent Markov Chains in parallel on the same input string
- Intuition: The Markov Chains are correlated in a hard-to-predict way



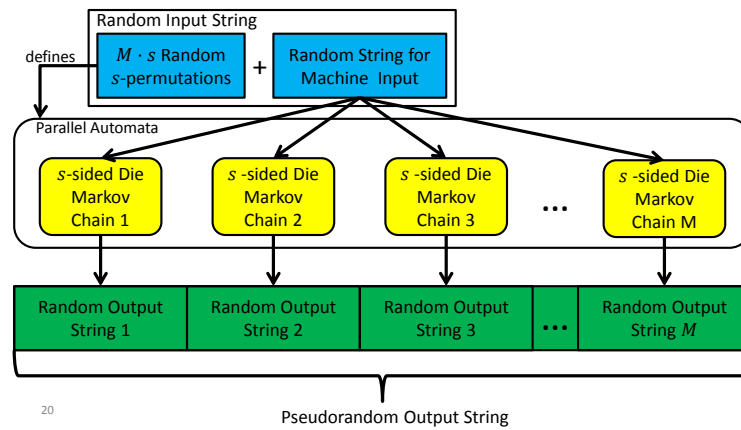
18

AP-PRNG Algorithm



AP-PRNG Algorithm

M machines with s states each



Evaluation*

- Evaluate using statistical tests
 - TestU01 test suite
- Compare with state-of-the-art
 - Philox
 - GPU: 145 GB/s
- APPRNG:
 - First Gen AP: 436.9 MB/s per chip
 - Modern DDR3: 12.8 GB/s per chip
 - 409.6 GB/s per board
 - HMC: 28.3 GB/s per chip
 - 905.6 GB/s per board
 - 6.8× energy efficiency per bit

THIS APP RANDOM NUMBER GENERATOR YOU WROTE CLAIMS TO BE FAIR, BUT THE OUTPUT IS BIASED TOWARD CERTAIN NUMBERS.

WELL, MAYBE THOSE NUMBERS ARE JUST INTRINSICALLY BETTER!

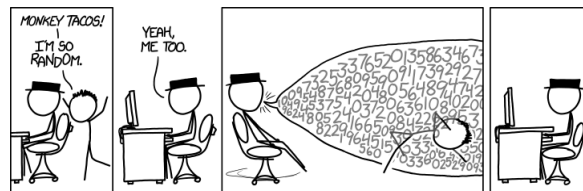


21

*Work done in collaboration with Jack Wadden

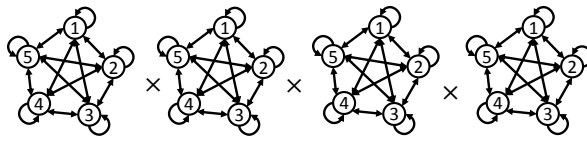
Evaluating Cryptographic Security

- D distinguishes random vs. pseudorandom implies D solves a “hard” problem
- Assumption:
 - It is difficult to distinguish a random walk on a large automaton from a small one.



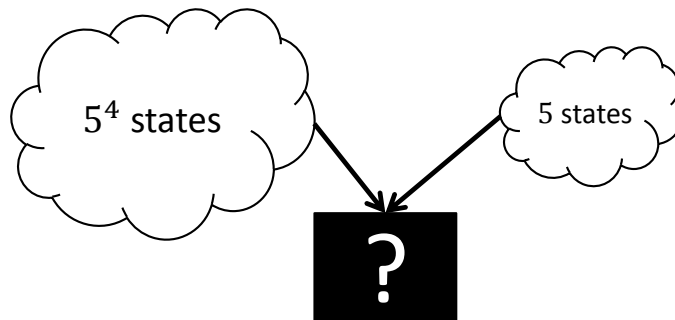
22

Two worlds





23

Two worlds



24

History of Automata Learning

| Year | Author | Result |
|------|--|--|
| 1978 | Angluin | Learning from examples is NP-Complete |
| 1987 | Angluin | L^* , learns by selecting input, requires “reset” capability |
| 1989 | Rivest & Schapire  | Learning with homing sequences, removes “reset” |
| 1993 | Freund et al. | Learning with random input |
| 1994 | Kearns & Valiant  | Learning acyclic automata is as hard as RSA |
| 2015 | Angluin | Efficiently learn random DFAs from random strings |

25

Automata-based Bloom Filters

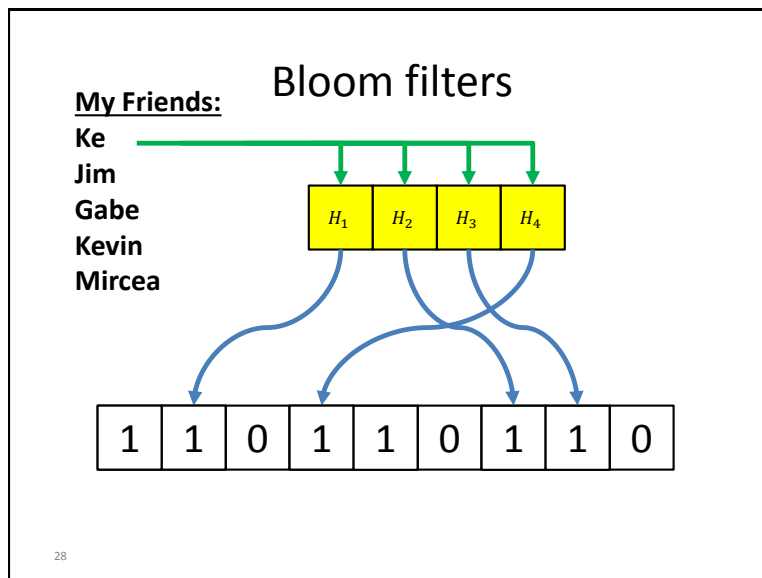
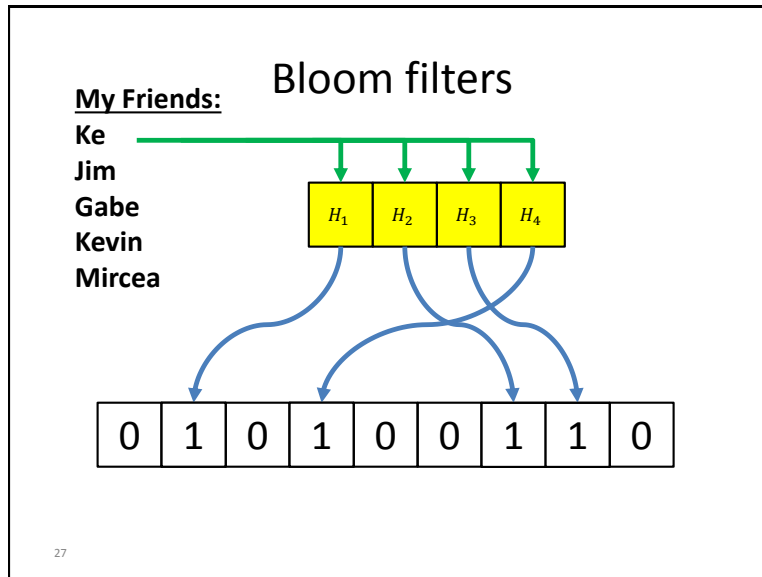
- Application enabled by APPRNG
- Set membership data structure
 - Insert(item): add item to data structure
 - Contains(item): True if item is in inserted set
- Bloom filter:
 - Insert: $O(1)$
 - Contains: $O(1)$

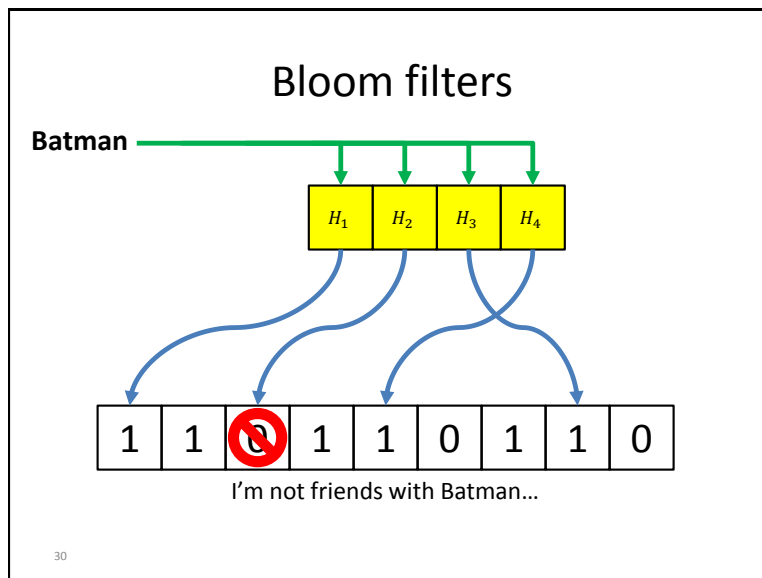
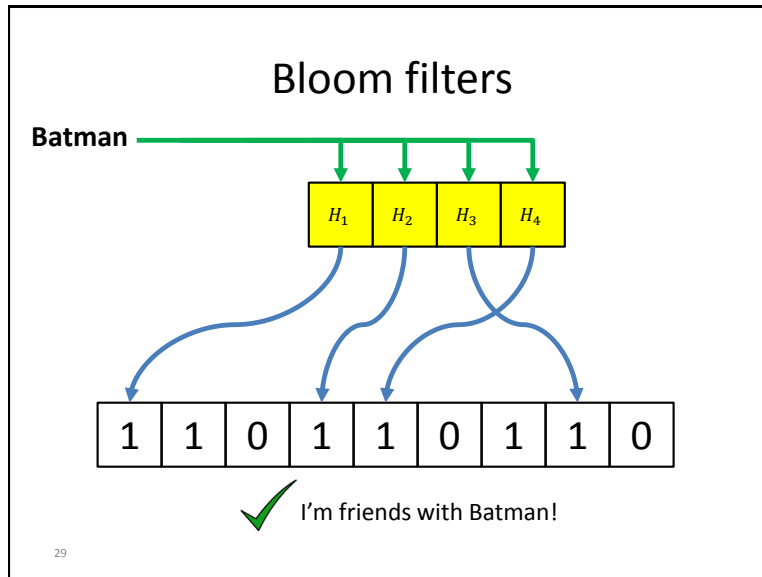
False Positives



Statisticians have a different version of 'The Boy Who Cried Wolf.'

26





Automata Bloom filters

My Friends:

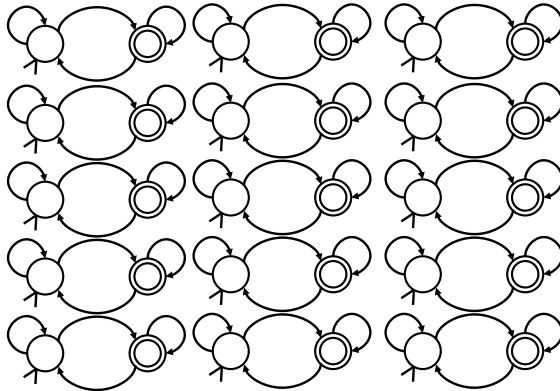
Ke

Jim

Gabe

Kevin

Mircea



31

Automata Bloom filters

My Friends:

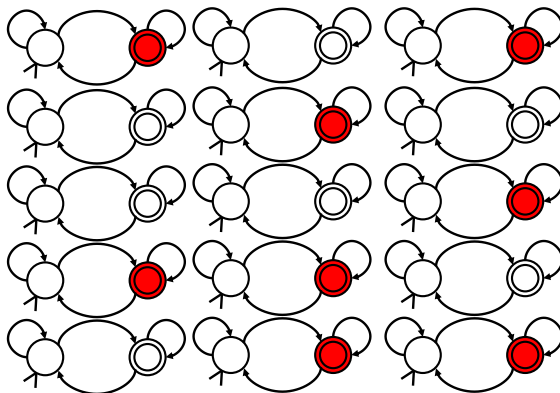
Ke

Jim

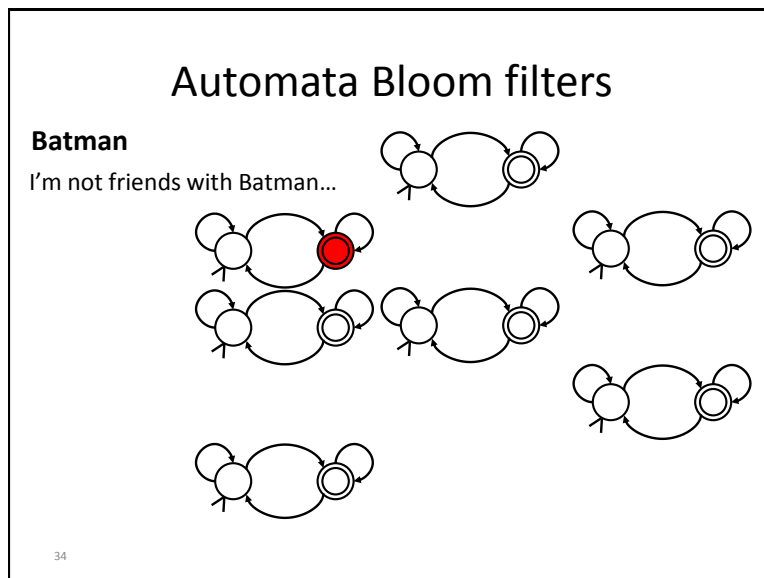
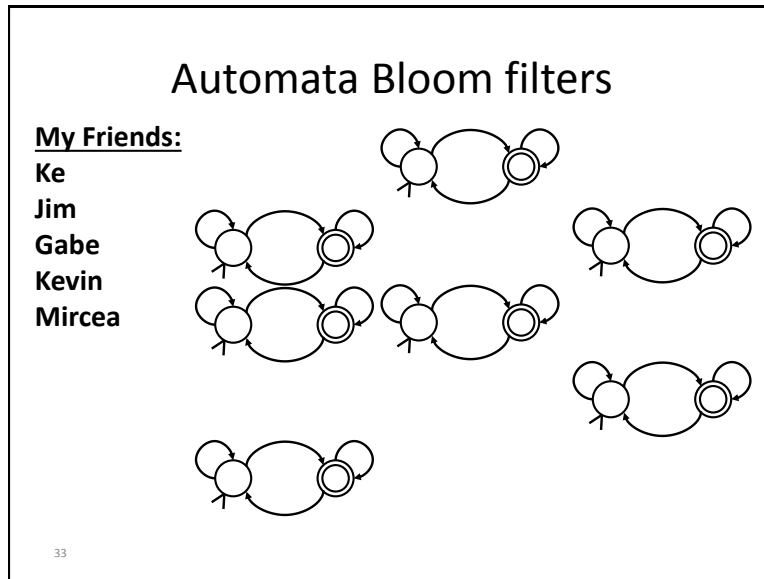
Gabe

Kevin

Mircea



32

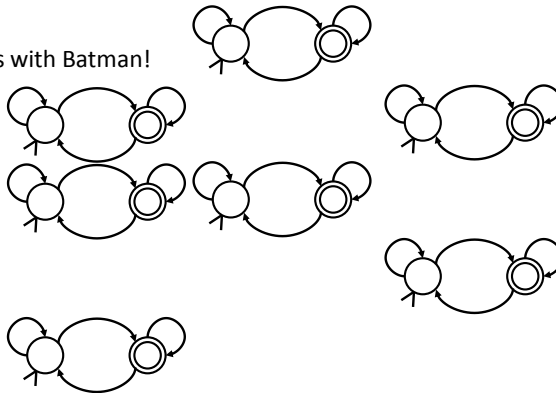


Automata Bloom filters

Batman



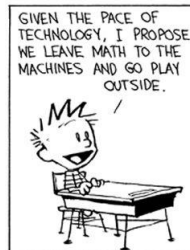
I'm friends with Batman!



35

How Many Machines?

- For a Bloom filter of m bits, n inserted items
 - $\frac{m}{n} \ln 2$ hashes
 - Must begin with $(2 \ln 2)m \approx 1.386m$ Machines
 - After removing all accepting automata
 - $(\ln 2)m \approx 0.693m$ machines remain



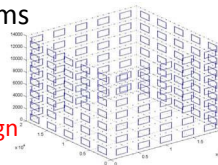
36

Conclusions

- **Life-extension** techniques for Moore's Law:

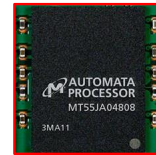
- **Software**: Compression aware algorithms

- Geometric and text data
 - Lossy and Lossless compressions
 - Compression scheme & Algorithm **Codesign**



- **Hardware**: Automata Processors

- Efficiently compute regular languages
 - Pseudorandom number generator
 - Bloom filters



- Longevity?

37

Future Directions

- Compression-aware **data structures**
 - Revisit classical data structures w.r.t. compression
 - E.g. Compression-aware KD-Trees
- **Combine** AP with compression-aware algorithms
 - Algorithms for schemes with AP-based decompression
 - Compressed pattern matching
 - E.g. Pattern matching on Huffman-coded data
- Explore **lossyness** in compressions
 - Leverage imprecise hardware
 - Markov Chains vs. imprecision vs. lossyness

38

Questions?

- J. Hott, N. Brunelle, J. Myers, J. Rassen and a. shelat. KD-Tree Algorithm for Propensity Score Matching With Three or More Treatment Groups. Technical Report Series. Division of Pharmacoepidemiology And Pharmacoeconomics, Department of Medicine, Brigham and Women's Hospital and Harvard Medical School, 2012.
- N. Brunelle, G. Robins, a. shelat. Compression-Aware Algorithms for Massive Datasets. Data Compression Conference (DCC), 2015.
- N. Brunelle, G. Robins, a. shelat. Algorithms for Compressed Inputs. DCC, 2013.
- T. Tracy, M. Stan, N. Brunelle, J. Wadden, K. Wang, K. Skadron, G. Robins. Nondeterministic Finite Automata in Hardware - the Case of the Levenshtein Automaton. Workshop on Architectures and Systems for Big Data (ASBD), in conjunction with ISCA, 2015.
- J. Wadden, N. Brunelle, K. Wang, M. El-Hadedy, G. Robins, M. Stan, K. Skadron. Generating efficient and high-quality pseudo-random behavior on automata processors. ICCD, 2016.
- J. Wadden, V.Dang, N. Brunelle, T.Tracy II, D.Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, K.Skadron . ANMLZoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures. IISWC, 2016.
- N. Brunelle, G. Robins, a. shelat, Algorithms for compressed inputs. In preparation for Journal of Discrete Algorithms.
- N Brunelle, J. Wadden, T. Tracy, M. Wallace, G. Robins, K. Skadron. Pseudorandom Number Generation using Parallel Automata. In preparation for Journal of Experimental Algorithmics.
- J. Wadden, N. Brunelle. System, Method, and Computer-Readable Medium for High Throughput Pseudorandom Number Generation. Patent Application no. 15/091925, Filed April 2015