Anarchy

A Library for Incremental Chaos

Peter Mawhorter
Wellesley College
106 Central St.
Wellesley, Massachusetts 02481
pmawhorter@gmail.com

ABSTRACT

Pseudo-random number generators are ubiquitous components of content generation systems, because their outputs are difficult to predict but also repeatable given an initial seed. These properties make them especially useful as the basis for "random" decisions during a generative process, as they allow the process to be chaotic but also repeatable. This paper describes an open-source family of pseudo-random algorithms which allow for shuffling and distributing items in a reversible and incremental manner. To demonstrate the applicability of these algorithms, I show how they have been used in the creation of a word-search game which includes strong guarantees about the distribution of words that can be discovered.

CCS CONCEPTS

•Theory of computation → Pseudorandomness and derandomization; Generating random combinatorial structures; •Applied computing → *Media arts;* **•Security and privacy →** Hash functions and message authentication codes;

KEYWORDS

Generative Algorithms, Computational Creativity, Pseudo-random Number Generation, Noise Functions, Chaos, Procedural Content Generation

ACM Reference format:
Peter Mawhorter. 2019. Anarchy. In *Proceedings of Foundations of Digital Games, San Luis Obispo, CA USA, August 2019 (FDG2019),* 8 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

INTRODUCTION

Pseudo-random number generators (PRNGs) are a fixture of generative algorithms, supplying unpredictable values that are used to disrupt undesirable regularity when generating anything from music to poetry. Most generative algorithms must make decisions as part of their execution, and while some decisions may be made based on principles or encoded theories of the medium in question, others are usually made arbitrarily, and these decisions give rise to additional variety in the space of generated artifacts. If these arbitrary decisions were to be made using a uniform policy, the artifacts produced by the process would always be the same given the same input parameters, but in order to more closely mimic human creative processes (which rarely result in identical outputs even with the same inputs), these decisions are more often made using some random or pseudo-random process. Pseudo-random processes and PRNGs in particular are often the go-to choice because unlike truly random processes, they allow for a repeatable performance: if a particular output is especially useful or aesthetically pleasing, it can be recreated easily.

These algorithms typically include the ability to generate a dependable sequence given an initial seed, and therefore effectively group an indefinite set of arbitrary decisions that are part of a creative process and tie them to a single input value. To give a concrete example, imagine an AI painter which must somehow decide exactly how each digital stroke of a painting affects its digital canvas. In a human painting, the chaotic interactions between paint viscosity and the physical systems of the brush and canvas give a slightly different character to each stroke, and this variety is important to the overall impression of the end result, although it is by no means the result of conscious decisions on the part of the painter, nor is it important exactly how each brush stroke turns out: it only matters that they do not all look the same. To emulate this chaotic system digitally, a computer may use a PRNG to decide exactly which pixels should be colored when the AI artist adds a digital stroke to the virtual canvas. These micro-level decisions must be made arbitrarily, but instead of treating these decisions as millions of individual inputs to the creative process, they are effectively tied together to depend on a single parameter: the seed of the PRNG. With the same seed (and holding other parameters constant), a painting may be exactly reproduced, down to the "random" virtual splatters of ink. This reproducibility might be especially useful in other domains, for example, in games to allow different users to experience the same content (cf. e.g., Noctis [8]).

Although PRNGs are typically irreversible, reversible versions do exist, and have been applied in cryptography. Whereas a normal PRNG allows the 'next' number in a pseudo-random sequence to be generated given a seed value, a *reversible* PRNG also enables the efficient computation of the previous sequence value. Using reversible PRNGs for generative purposes allows a number of interesting derivative algorithms to be created, including incremental shuffling and distribution algorithms, and these applications are the focus of this paper. Unlike a normal algorithm for shuffling items randomly, an *incremental* algorithm can generate a small part of a complete shuffled sequence without computing the entire sequence, and in particular, an *isotropic incremental* algorithm can generate

FDG2019, San Luis Obispo, CA USA
 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

parts of the sequence in any order. Both the reversibility and the incremental nature of the algorithms developed here open up new possibilities in the space of generative games.

Anarchy is an open-source library available in C, Python, and Javascript that makes use of a reversible PRNG to provide a suite of low-level functions for building generative algorithms. To demonstrate how these algorithms can be used as components of more complex generative algorithms, I describe an application that generates a hex grid of letters which includes all of the words from an arbitrary corpus. Whereas a human designer of a word-search game might carefully choose a thematically consistent set of words and then lay them out to create interesting juxtapositions, the algorithm described here enables a categorically different design space (that of indefinite word grids) but approaches the task with much more modest goals (no considerations for thematic relevance or juxtaposition are made). Accordingly, this is an example of hybrid machine-human creativity: the machine part of the design system is responsible only for very low-level decisions in the creative process (most of the game is designed by the human *around* the algorithm's capabilities), but enables the combined system to achieve a result that would be effectively impossible for a human alone (simply because of the sheer size of the generated game board, which lends a unique feeling to the game from the player's perspective).

RELATED WORK

This work is inspired by work in pseudo-random number generation and texture synthesis. There are also parallels to work in cryptography, although those algorithms operate under a very different set of constraints from algorithms designed for creative uses.

Pseudo-Random Number Generation

The basis of the Anarchy library is a reversible pseudo-random number generator, similar conceptually to such well-know systems as the Mersenne Twister [14]. However, to build a reversible generator, much simpler techniques were used (not worrying about cryptographic applications helps), including ideas derived from the linear feedback shift register [3]. As mentioned already, existing reversible PRNGs for use in cryptography demonstrate some of the advantages of such systems [5], but to our knowledge, reversible PRNGs have not yet been widely used in generative algorithms. For those interested in practical advice on PRNG construction, Jones' guidelines for PRNG use in bioinformatics is informative [10], although many of the restrictions necessary for that work do not necessarily apply to generative algorithms.

It is worth noting that the usefulness of PRNGs in generative algorithms is contested, and researchers concerned with human ascriptions of creativity have pointed out that the use of "random" or even pseudo-random processes may decrease the layperson's perception of the creativity of a computational system [4]. Colton, Cook, Hepworth, and Pease state:

> *...any unexpected behaviour or output by software is cherished by observers. However, the bottom drops out of this experience when they realise that – rather than some inspired choices – a random number generator was largely responsible for the novelty.*

The point is well taken that pseudo-random processes may be a double-edged sword when it comes to building creative systems, but they are nevertheless an important part of the designer's toolbox. Also, the specialized PRNG systems described here are not deployed strictly in the service of unexpectedness, but rather in order to achieve variety in situations where order would be boring to the audience.

Texture Synthesis

Another inspiration for this work is texture synthesis algorithms, also sometimes referred to as procedural noise algorithms [13]. In particular, the work of Ken Perlin and others on Perlin noise [18] and simplex noise [9, 17] demonstrates the concept of an isotropic incremental algorithm. Simplex noise provides an N-dimensional everywhere-continuous (and differentiable) function, which can be computed at an arbitrary point in space without requiring knowledge of more than a finite neighborhood around that point. This means that it is isotropic, in the sense that it does not matter which order different regions are computed in, and incremental, in that any particular region of interest may be evaluated without requiring the evaluation of the function over a larger region. Note that it is very much not reversible, however: given a particular height value, finding all of the regions which are at that height requires enumeration of the space until such a region is found, with no guarantees about how long that process might take or whether there is even *any* point in the space at that height.

The key idea here is that given limited local dependence between values, a property like continuity can be guaranteed everywhere while allowing the function to be computed quickly for arbitrary inputs. In simplex noise, for example, the noise value at each point is determined by a combination of finite-domain basis functions, so for any particular point, the algorithm only needs to know the values for a fixed number of local basis functions, and basis functions elsewhere in the space are guaranteed by their finite domains to be irrelevant at that point. I use the term "incremental" to refer to this because although the function is defined over a large region of 2D space (it repeats when the seed values for the PRNG that determines basis functions cycles), it can be computed for a portion of that space without computing the value of the function everywhere. Furthermore, the word "isotropic" implies that the order in which regions are generated is irrelevant to the algorithm, as opposed to an incremental algorithm where sub-regions could be computed independently, but only in a particular order (for example if there was some dependency between subregions).

Simplex noise and related isotropic incremental algorithms are crucial enablers for vast procedural worlds such as the galaxy of *Noctis* or the near-endless terrain of *Minecraft* [8, 16]. Seeding allows these worlds to be shared, but as detailed in the following section, augmenting these algorithms using reversible PRNGs offers additional benefits. The sheer amount of virtual space that can be generated by these algorithms makes a categorical difference to the player's experience and usually also ends up influencing many other aspects of the design of these games.

It is important to mention here that the resulting aesthetic and thematic content of these games is often colonialist: by using generative techniques to create a "vast, untamed wilderness" or "a near-infinite galaxy of worlds to explore," these games reinforce the harmful colonialist myths of *terra nullis* and promote the twisted idea that brutal subjugation/harvesting of vast game territories is both enjoyable and unproblematic (see for example, [6]). These kinds of generative systems have sometimes been used in less problematic ways, and Kreminski and Wardrip-Fruin have identified gardening games as an example genre that puts players a different relationship with generated artifacts to ultimately serve very different aesthetic and thematic ends [12].

Exhaustive PCG

Sturtevant and Ota's work [19] on exhaustive PCG, or EPCG, is quite relevant to this work, both because their approach (exhaustive enumeration of possible solutions) enables similar design spaces, and because reversible isotropic incremental algorithms can be used as part of an EPCG approach for ranking and unranking solutions. In fact, given a arbitrary ranking/unranking algorithm which indexes solutions in some domain-specific order, Anarchy's incremental shuffling capabilities can produce an augmented ranking/unranking algorithm which indexes solutions in pseudo-random order, which could be used to promote variety in systems which use early stopping criteria during generation. Whereas the most significant limitation of EPCG is the difficulty in applying it to problems with large design spaces, indexable algorithms are specifically designed to tackle that problem by only ever generating the part of the space that the player has the time to explore.

Cryptography

Previously mentioned systems such as reversible-cellular-network-based PRNGs [5] have typically been applied in the domain of cryptography, which has stringent requirements for pseudo-random numbers. In addition, the general concept of encrypting and decrypting a message carries with it the notion of a reversible but chaotic transformation, and research in symmetric cryptography has investigated means of preserving certain properties through such transformations (e.g., [11]). As a consequence of ignoring cryptographic constraints, however, Anarchy can be better optimized for creative purposes and generally requires less memory and time than more secure algorithms.

THE ANARCHY LIBRARY

Anarchy is an open-source library with C, Python, and Javascript implementations available at github.com/solsword/anarchy. It includes a reversible PRNG which is used to power algorithms for incremental shuffling and incremental distribution.

Reversible PRNG

Anarchy's reversible PRNG produces sequences of numbers in either forward or reverse order which are chaotic and which have relatively long periods. The PRNG is constructed using a sequence of the following operations on unsigned 64-bit integers (unsigned 32-bit integers in Javascript):

```
FLOP_MASK = 0xf0f0f0f0f0f0f0f0

def flop(x):
  left = x & FLOP_MASK
  right = x & ~FLOP_MASK
  return ((right << 4) | (left >> 4))
```

Figure 1: Simplified Python code for the `flop` operation, which is its own inverse.

`swirl(n)`	Shifts each bit *n* units left, wrapping higher-order bits into lower-order bits so no information is lost.
`fold(n)`	XORs the first *n* bits with the bits to their left. Fold is its own inverse.
`flop`	Swaps every 4-bit sequence with the adjacent 4-bit sequence. Flop is its own inverse.
`scramble`	Operates like a linear feedback shift register, but uses a circular shift and an XOR mask that doesn't overlap the trigger bits, so that it is reversible.

The specific seed adjustments and operation sequence used for the PRNG are still being optimized for period and autocorrelation; refer to the source code for implementation details. These primitive reversible operations can be assembled into a number of possible PRNGs, and their individual reversibility is the most important property of the algorithm. The reverse PRNG is constructed by applying the inverse of each operation used for the forward direction in reverse order. So a PRNG that used `swirl(7)`, `fold(11)`, `flop`, `scramble` could be reversed by calling `rev_scramble`, `flop`, `fold(11)`, `rev_swirl(7)`. For convenience the individual parameters are determined from an input seed, so that a variety of value sequences can be generated using different seeds. Optimizing to minimize sequence correlation for sequences generated from sequential seeds is a current area of development for the library.

It is worth noting that in this and all algorithms in this paper, reversibility comes at a cost: the quality of the pseudo-random numbers is far from cryptographic standards. Results from the `dieharder` test suite [2] for Anarchy's PRNG are mixed: it fails many of the tests that are sensitive to repeated bit sequences such as the `bitstream`, `oqso`, and `dna` tests, but passes others including `opso`, `rank_32x32`, and `birthdays`. Notably, it passes the Marsaglia and Tsang GCD tests from the `dieharder` suite, but more interesting is that while it passes 1-, 2-, and 3-bit STS `serial` tests, it fails for 4 bits and most (but not all) longer bit sequences, implying that the PRNG has a problem with some bit sequences either appearing too regularly or else some not appearing at all. Thankfully, what would be a significant flaw for application to a sensitive field such as statistical or cryptographic randomization is only relevant to generative algorithms insofar as it produces unwanted and human-observable regularities in generated output. The PRNG does pass tests such as the diehard `parking_lot` test and the RGB `permutations` test which both test the algorithm in ways similar to how it might be used to generate content for a

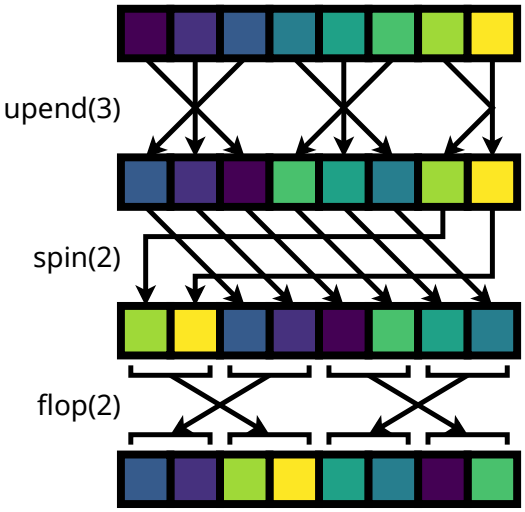

Figure 2: An example of how reversible parametrizable primitives can be combined to perform a shuffle operation. Note that the shuffled index of each individual element can be computed independently in constant time, regardless of the number of items being shuffled. Anarchy's shuffle algorithm uses a total of 15 applications of its seven primitive operations rather than just the three shown here.

game. Improving the algorithm to pass more of these tests under the constraint of reversibility is one area of future work.

Incremental Shuffling

A common operation in generative algorithms is shuffling, typically using a memory-based swapping algorithm such as the Fisher-Yates shuffle [7]. A shuffled set can be used to select a subset from a larger set at random without replacement, and in this case, only a prefix of the shuffled set is needed. When the required prefix size is small and the superset is large, Batagelj and Brandes have proposed a virtual Fisher-Yates shuffle [1] which takes time proportional to the desired subset size. Anarchy takes this further, using its reversible PRNG to provide arbitrary access to shuffled elements in constant time per element, thus doing away with the requirement that the portion of the shuffled set being used is a prefix (in other words, it is not only indexable, but also isotropically indexable).

To achieve this, reversible operations can be applied to the items to be shuffled which change their position in a virtual "cohort" with a predetermined size (see Fig. 2). All of the component operations can be computed independently for each item in a cohort, so the algorithm is both indexable and isotropic. Additionally, any index in the shuffled cohort can be reverse-shuffled to find its index in the original cohort, which opens up new design possibilities.

The drawback of this approach is not immediately apparent, but is revealed by consideration of the combinatorics of the situation. No matter how many elements are being shuffled, this reversible shuffle algorithm produces a deterministic outcome for each distinct seed. Even ignoring the potential for two different seeds to result in the same shuffle, in a 32-bit implementation, there are only 2^{32} distinct seeds available, and so at most there are that many possible shuffle results (and likely there are fewer). However, when doing a virtual shuffle of 10,000 elements, the number of possible permutations (abstractly $O(n!)$) is *much* greater than 2^{32}, or even 2^{64}. Because the number of possible shuffled configurations grows so fast ($21! > 2^{64}$), even when shuffling a moderate number of elements, this reversible/indexable shuffle algorithm is capable of producing only a small subset of all possible output orderings. Although there is no reason to expect that the subset of orderings produced has any kind of systematic bias (and in fact the performance of the underlying PRNG on some of the relevant `diehard` tests suggests that it might be fine), I am still investigating the distribution of the produced orderings among the possible orderings, and intend to optimize the specific operations and parameters of the reversible shuffle to minimize biases that might occur.

The seven operations used for indexable shuffling are as follows:

`interleave`	Puts the first half of the cohort into even indices and the second half into odd indices in reverse order.
`fold(n)`	Swaps *n* items from the end of the cohort with items from the middle.
`flop(n)`	Divides the cohort into sections of size *n*, and swaps each even section with its odd neighbor. Elements that would end up outside the cohort are left in place.
`spin(n)`	Moves every item *n* spaces towards the end of the cohort, wrapping around at the end.
`mix(n)`	Treats even and odd elements as separate half-sized cohorts and performs a spin operation on each using different *n* values.
`spread(n)`	Divides the cohort into sections of size *n*, and puts the *i*th member of the *j*th section into the *j*th spot in the *i*th section.
`upend(n)`	Divides the cohort into sections of size *n*, and reverses ordering within each section.

```
def cohort_interleave(inner, cohort_size):
  if inner < (cohort_size + 1) // 2:
    return (inner * 2)
  else:
    return (cohort_size - 1 - inner) * 2 + 1

def rev_cohort_interleave(inner, cohort_size):
  if inner % 2:
    return cohort_size - 1 - inner//2
  else:
    return inner//2
```

Figure 3: Simplified Python code for the `interleave` cohort operation and its inverse.

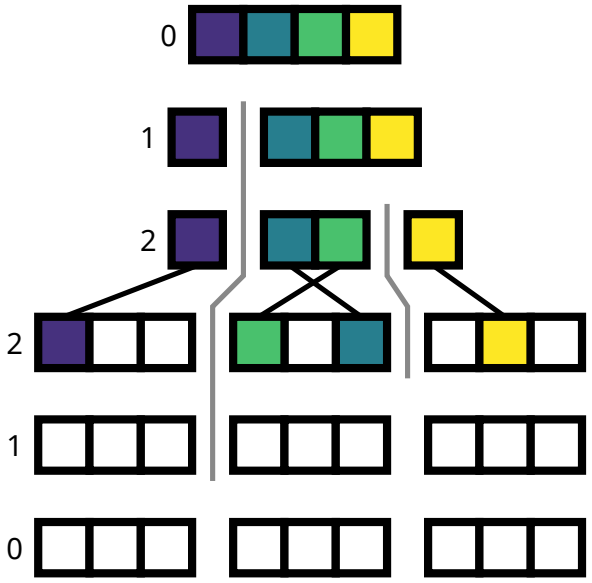

Figure 4: An example showing random distribution of four items among three three-item sections (result shown in the fourth row of the figure). Using recursion, the sections are evenly divided while the items are randomly divided at each step, working from the top and bottom of the figure towards the middle (numbers indicate recursion depth). When a single section has been identified, items are mapped to indices using a reversible shuffle.

The specific combination of these operations used for shuffling can be optimized for purposes such as cycle length and adjacent-seed correlation avoidance, but the core idea of combining reversible operations to provide an indexable shuffle can be implemented using various schemes. Besides the example of word distribution given in the next section, this indexable/reversible shuffle algorithm is potentially useful for a number of generative tasks. For example, as outlined in [15], I am already exploring the potential of these operations for the generation of family tree structures.

Incremental Distribution

In addition to shuffling items, Anarchy includes routines for dividing n items randomly between s sections such that an item's section index can be computed in logarithmic time, and given a section and in-section index, the original item index can also be computed in logarithmic time. Furthermore, the total number of items preceding any section can be computed in logarithmic time, allowing for indexing of the non-distributed items in each section. Fig. 4 illustrates the algorithm: at each step, the items that need assignment are divided in two randomly, while the available sections are divided evenly. The algorithm then proceeds recursively with a smaller number of sections to assign to, ultimately performing $1 + \log_2(s)$ recursive calls to compute the section for an individual item. When dividing the items, a random index is chosen under the constraint that the total number of items on either side must not exceed the available section slots on that side of the section division. This random division can be biased towards the midpoint of the items to create a smoother distribution of items among sections.

A step-by-step description of this algorithm is as follows:

(1) Split the available segments in half.
(2) Split the items being distributed into two groups at a random point *(subject to the constraint that we cannot pick a split which assigns more items into one of the groups than there are slots in the corresponding segment group)*.
(3) Pick either the first or second group for both segments and items, and recurse if there is more than one segment.
    - If we are trying to find which segment an item is assigned to, we compare the item index to the random item split point to determine which groups to use for the recursive call.
    - If we are trying to find out which item(s) are in a particular segment, we use the first or second groups depending on whether the segment we're interested in is in the first or second half of the segments at this level.
(4) When a single segment remains, use an incremental shuffle to assign each item to a slot in the segment. In the extreme, a single segment could be entirely full or empty.

Note that computing the section assignments for any set of n items takes $O(n \log(s))$ time, independent of the size of the sections and the total number of items being assigned. Furthermore, by recursing according to the section division instead of the item division, the original item index of an arbitrary item within a section can be computed via the same method. Finally, by returning the sum of the items split off to the left of a given section, the algorithm can compute the total number of items that are distributed prior to a section (again in logarithmic time), allowing the items which are not part of the distribution to be indexed contiguously. The next section gives one example which shows why these properties are useful for procedural generation.

Incremental Embedding

To give an example of Anarchy in action, I have created a word-finding game that embeds letters into an indefinite hex grid (limited only by integer overflow issues). For each position in the grid, the system can quickly compute the corresponding letter, and these can be recomputed as necessary so that the grid contents do not have to be stored in memory. The computation is seeded, so multiple players on different machines can access equivalent grids, and the system does not need to save grid contents between play sessions. In addition, the grid is filled in such a way that in each 86016×86016 section of the grid, each word from a corpus of millions of words is present at least once, and extra slots are filled according to the frequency of each word in the corpus. Finally, some slots are excluded from the normal assignment mechanics, and are instead used to include words from other languages. To achieve all of this, the demo relies heavily on the features of the Anarchy library.

First, the grid is divided into 37-hex supertiles, which each include a central hex and all hexes within three units of it (see Fig. 6). These supertiles seamlessly tile the entire grid, and are divided into six six-hex triangular regions, with the center tile unassigned. Each

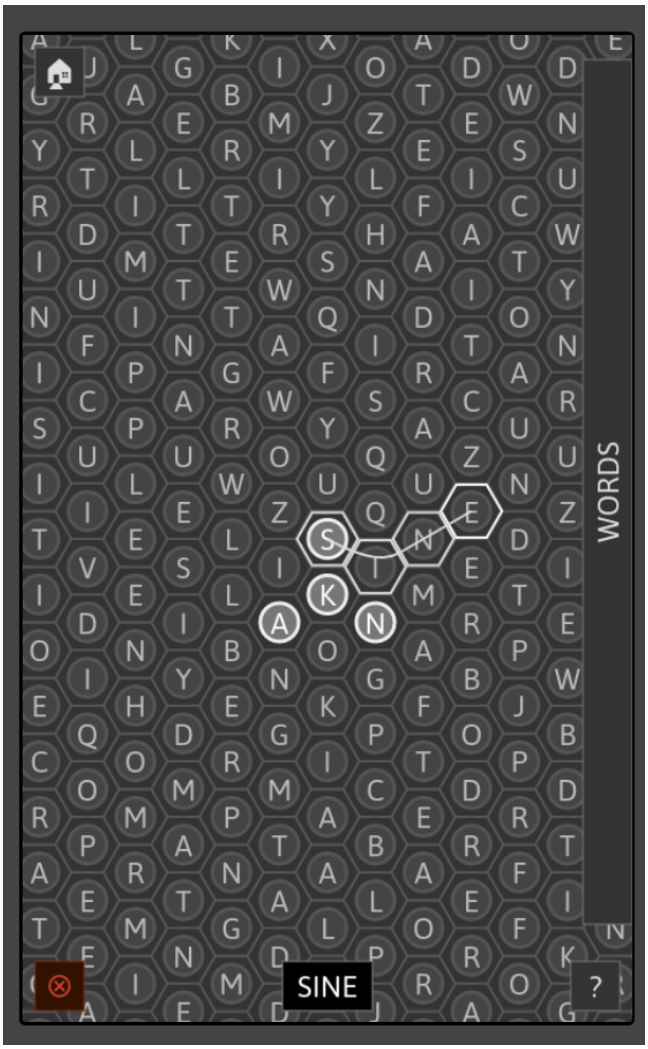

Figure 5: The interface for the word-finding example game. An indefinite hex grid is filled with letters, and the user can select valid words to unlock hexes which can then be used to select more words. The hex grid scrolls in all directions limited only by index integer overflow issues. The bar on the right labeled 'WORDS' can be expanded to show a list of all words found so far, while the current selection is shown at the bottom.

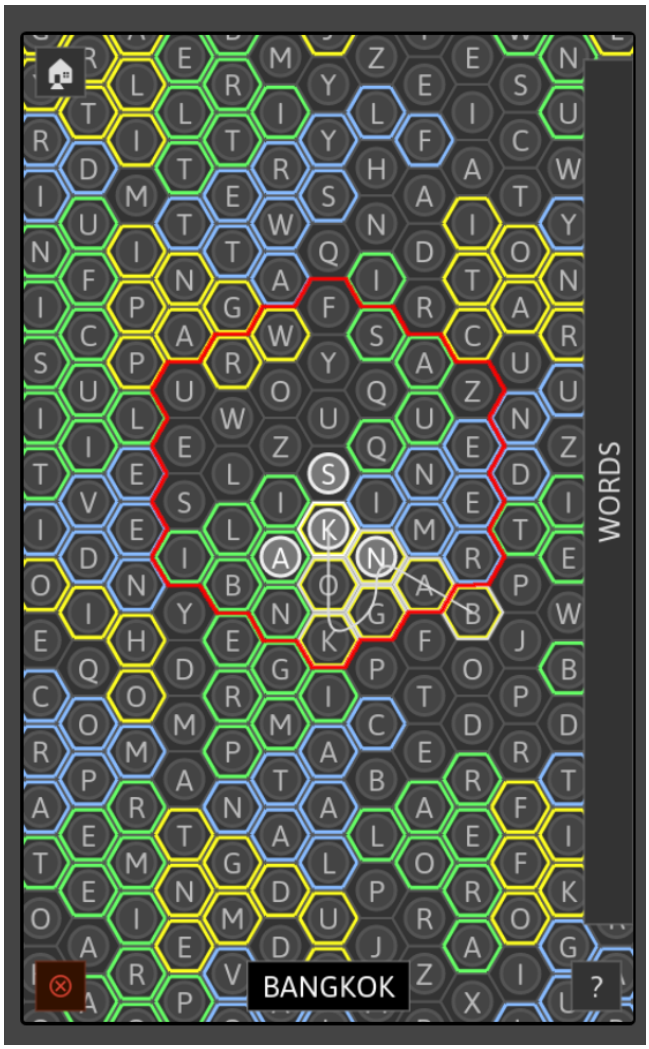

Figure 6: The same region as in Fig. 5, but with assigned words highlighted. The red border indicates a single supertile. Note how each word is packed into two triangular regions from adjacent supertiles (e.g., the green word at the bottom of the highlighted supertile), although not all words fill their assigned regions.

triangular region is paired with the adjacent region from a neighboring supertile to form an assignment socket, and the words from the corpus are distributed into these sockets. Fig. 6 shows the same region as Fig. 5, but with the socketed words highlighted. Each word is laid out randomly within its socket (words with more than 12 letters are handled using a different procedure). After assigning letters to socketed words, empty hexes are filled according to trigram, bigram, or unigram letter probabilities measured within the corpus.

This algorithm can thus take an assignment of words to sockets and produce a hex grid, and that assignment step is where Anarchy comes in. First, supertiles are grouped into 12×12 parallelogram regions called ultratiles, and 1024×1024-ultratile regions are designated as 'assignment regions.' Each assignment region therefore contains 452,984,832 sockets, which is more than enough to accommodate even a very large corpus. To assign words to each socket, as reversible shuffle is used on the 452 million sockets, and for a corpus of size n, the first n shuffled sockets are assigned to the n words of the corpus to ensure that each word occurs at least once. This leaves $452984832 - n$ sockets that need words assigned, and these assignments should be made according to the relative frequency of each word.

To accomplish this frequency-relative assignment, the words in the corpus are sorted by their frequencies, and the number of words of each frequency is tallied in a table (the ordering of words within

each frequency bucket is unimportant, since the goal is to sampling at random). This table creates a virtual index space where each word is assigned a number of index slots equal to its frequency in the corpus. For example, if the corpus contains three words with frequencies 3, 3, and 10, virtual indices 0–2 would map to the first frequency-3 word, indices 3–5 would map to the second frequency-3 word, and the remaining indices (6–15) would all map to the frequency-10 word. Sampling uniformly from this virtual space will sample words according to their frequencies, so another reversible shuffle is used to do so, shuffling the remaining sockets and aligning them against this virtual index space. Using the aforementioned table of number-of-words-for-each-frequency, this alignment can be done in time proportional to the number of distinct frequency values that appear in the corpus (typically less than 1000 even for very large corpora).

The algorithm as described so far can assign a word to each socket, but the demo includes an additional feature: 'inclusions' of foreign words are scattered throughout the space to make things more interesting. To avoid interference with the just-described assignment scheme, sockets taken up by inclusions are removed from the available sockets, and assignment proceeds as before using a reduced number of sockets for each assignment region. This is made possible using Anarchy's distribution procedures to distribute a fixed amount of 'inclusion' material among the ultratiles in an assignment region, thus reducing the available sockets throughout the region. For any given ultratile, the linear index among non-inclusion sockets can still be computed, however, because the number of prior inclusion sockets for that ultratile can be computed in logarithmic time, as described above. This linear index is used with the base assignment scheme, allowing inclusions to coexist with frequency-based word assignments. Within each ultratile, the specific location of inclusions and the resulting specific indices of each non-inclusion socket are cached in memory for quick access, but can be recomputed from the ultratile parameters if necessary.

Advantages of the Incremental Approach

The embedding scheme described above offers several advantages over a simpler scheme which could just use a normal PRNG to assign a word from the corpus probabilistically to each socket. First, by using a shuffle instead of probabilistic assignment, each word is guaranteed to appear somewhere, where using probabilistic assignment, many words would be "unlucky" and never appear. There is even a guarantee of the exact number of times each word appears, and this shows how shuffle-based assignment schemes can be helpful when dealing with models that include rare events or features. However, shuffle-based assignment using an in-memory shuffle would create a performance problem given the approximately 450 million sockets that need to be filled. This is where Anarchy's ability to provide an indexable shuffle shines: only the parts of the shuffle that show up on the player's screen must be computed, which can be done in constant time per unit.

The embedding is not only indexable, but it is also reversible, and this enables designs that are impossible under a simpler assignment scheme. For example, in this game it would be possible to assign a quest for the player to find a certain word, and not only would that word be guaranteed to be present, but the system could reverse the assignment algorithm to figure out where that word was located, without computing the locations of all assigned words. With classic irreversible assignment schemes, figuring out where a specific word appears (or even whether it appears at all) would in the worst case require enumerating the entire assignment space. However, with a reversible algorithm, each appearance of a given word can be tracked to its location on the grid in constant time by simply running the assignment algorithm backwards from the word index to a position.

In terms of broader applicability to different kinds of computationally creative systems, Anarchy's robust shuffling and distribution algorithms can potentially be applied in a number of ways. Examples include:

- Constant time reversible and indexable shuffling allows sampling from huge possibility spaces to be carried out in a manner that guarantees an *exhaustive* process. This could be leveraged to ensure that, for instance, each product of a generative algorithm is unique, without the chance that the same output occurs twice at random (up to limitations based on the finite generative space the algorithm is capable of producing). Of course, that can already be achieved by non-reversible techniques, but reversibility provides the additional property that seeds which will lead to specific output configurations can be located without searching the whole generative space. For example, if potentially millions of generated characters had pseudorandomly colored capes, a reversible assignment of these colors using Anarchy would allow efficient enumeration of all of the seeds which resulted in pink capes. This in turn could enable new ways of interacting with generative systems, for example the ability for the system to respond to a request to "Show me more like this." While such a design is achievable by other means, Anarchy makes such a query possible without much extra work on the part of the system designer because it includes reversibility by default.
- The distribution algorithms outlined above allow for arbitrary smoothing of output distributions across output spaces, which can provide guarantees that pseudorandomly-distributed content does not become too thick or thin just due to chance. For example, if one distributed an important feature such as villages in a *Minecraft*-like world using a fixed probability per unit of map area, arbitrarily-dense clusters of villages could occur. Although such clusters can in many systems be a good thing, as they represent a relatively novel outcome from the player's perspective, they can also be seen as unrealistic or a failure of the generative system if they are too frequent or too dense. Using Anarchy's distribution functions, an equivalent distribution in terms of villages-per-unit-area could be constructed with strict guarantees on the maximum possible density, without requiring extra computational resources or context when generating map segments piecewise.

CONCLUSION

This paper illustrates the advantages of reversible and indexable chaotic systems via an example word-finding game. By providing Anarchy as an open-source library (available at github.com/solsword/anarchy) and reporting on it here, I hope to encourage

its use and enlist others to find more applications for these algorithms. The word-finding demo app is also open-source, and can be accessed at github.com/solsword/words.

The main advantage provided by these algorithms lies in removing memory- and time-related constraints from shuffling and distribution algorithms when only a subset of results is required at once. In generative systems where the user observes only part of a huge implicit possibility space, these criteria are met exactly, and such systems are likely to be the most natural users of this library. However, as evidenced by the work cited here, many areas of computer science find use for pseudo-random values, and it is my hope that the Anarchy library can also prove useful for simulation systems, although more work needs to be done on the exact distribution of the permutations it creates to avoid potential biases. The beginnings of the application of this work to family-tree generation for just that purpose has already been described in [15].

Future Work

As may already be apparent, both Anarchy and the word-finding demo app are still under active development. One key area of future work is the optimization of all of Anarchy's algorithms to avoid patterns in the sequences produced both via repeated invocations using a single seed and via single invocations using sequential seeds (an unfortunately common use-case). Establishing guarantees for the period of the various generation components is also important, and both of these properties can be tweaked by altering the specific reversible primitives used for the PRNG and for shuffling. The demo application is also in the process of being made into a fully-fledged game, with the idea to encourage language learning through various design choices. For example, since the corpus used is arbitrary, specific vocabulary from a lesson plan could be used to populate the space and provide language students with a game-based tool for vocabulary practice. For these purposes, the ability of the system to locate specific words allows it to do things like provide hints that would otherwise be extremely difficult to implement.

REFERENCES

[1] Vladimir Batagelj and Ulrik Brandes. 2005. Efficient Generation of Large Random Networks. *Physical Review E* 71, 3 (2005), 036113.

[2] Robert G. Brown. 2019. `dieharder`. (2019). https://webhome.phy.duke.edu/~rgb/General/dieharder.php

[3] Anne Canteaut. 2005. Linear Feedback Shift Register. In *Encyclopedia of Cryptography and Security*. Springer, 355–358.

[4] S Colton, M Cook, R Hepworth, and Alison Pease. 2014. On Acid Drops and Teardrops: Observer Issues in Computational Creativity. In *AISB 2014–50th Annual Convention of the Society for the Study of Artificial Intelligence and Simulation of Behavior*.

[5] KR Crounse, Tao Yang, and LO Chua. 1996. Pseudo-random sequence generation using the CNN universal machine with applications to cryptography. In *Cellular Neural Networks and their Applications, 1996. CNNA-96. Proceedings., 1996 Fourth IEEE International Workshop on*. IEEE, 433–438.

[6] Daniel Dooghan. 2019. Digital Conquerors: Minecraft and the Apologetics of Neoliberalism. *Games and Culture* 14, 1 (2019), 67–86. DOI: http://dx.doi.org/10.1177/1555412016655678 arXiv:https://doi.org/10.1177/1555412016655678

[7] Ronald Aylmer Fisher, Frank Yates, and others. 1949. Statistical Tables for Biological, Agricultural and Medical Research. *Statistical Tables for Biological, Agricultural and Medical Research* Ed. 3. (1949).

[8] Alessandro Ghignola. 2000. Noctis. Self-published. (2000). Microsoft Windows.

[9] Stefan Gustavson. 2005. Simplex noise demystified. *Linköping University, Linköping, Sweden, Research Report* (2005). http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

[10] David Jones. 2010. Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications. (2010). http://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf

[11] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. 2012. Dynamic Searchable Symmetric Encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 965–976. DOI: http://dx.doi.org/10.1145/2382196.2382298

[12] Max Kreminski and Noah Wardrip-Fruin. 2018. Gardening Games: An Alternative Philosophy of PCG in Games. In *Procedural Content Generation workshop at the 13th International Conference on Foundations of Digital Games*.

[13] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, David S Ebert, John P Lewis, Ken Perlin, and Matthias Zwicker. 2010. A survey of procedural noise functions. *Computer Graphics Forum* 29, 8 (2010), 2579–2600.

[14] Makoto Matsumoto and Takuji Nishimura. 1998. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.* 8, 1 (Jan. 1998), 3–30. DOI: http://dx.doi.org/10.1145/272991.272995

[15] Peter Mawhorter. 2017. Efficiency, Realism, and Representation in Generated Content. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*. 72:1–72:4.

[16] Mojang. 2011. Minecraft. Mojang; Microsoft Studios; Sony Computer Entertainment. (2011). Various platforms.

[17] Mark Olano, JC Hart, W Heidrich, B Mark, and K Perlin. 2002. Real-time shading languages. *Course Notes. ACM SIGGRAPH* 99 (2002).

[18] Ken Perlin. 1985. An image synthesizer. *ACM Siggraph Computer Graphics* 19, 3 (1985), 287–296.

[19] Nathan R. Sturtevant and Matheus Jun Ota. 2018. Exhaustive and Semi-Exhaustive Procedural Content Generation. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 109–115.