|  |  |
| --- | --- |
|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ  *Информатика и системы управления*

КАФЕДРА *Программное обеспечение ЭВМ и информационные технологии*

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ***

***НА ТЕМУ:***

***Метод генерации псевдослучайных чисел на основе клеточного автомата «Жизнь»***

Студент ИУ7-41М **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Мирзоян С.А.**

(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель ВКР **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ковтушенко А.П**

(Подпись, дата) (И.О.Фамилия)

Консультант **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

Нормоконтролер **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

*2023 г.*

# РЕФЕРАТ

Расчетно-пояснительная записка \_ с., \_ рис., \_ табл., \_ источников, \_ прил.

Объектом разработки является метод для генерации псевдослучайных чисел.

Цель работы – продолжить исследования в методе генерации псевдослучайных чисел на основе клеточного автомата, изучить новые способы применения, а также предложить модификации для улучшения качественно-скоростных характеристик.

Поставленная цель достигается за счет решения следующих задач:

* изучение методов генерации случайных и псевдослучайных чисел;
* изучение новых методик представления карты вселенной игры «Жизнь»;
* Модернизация правил игры «Жизнь»;
* Модернизация последних реализаций данного метода генерации.
* Исследование качественных характеристик приведенного метода генерации псевдослучайных чисел;
* Сравнительный анализ модернизированного метода генерации с изначальной итерацией на предмет улучшения качественно-скоростных характеристик;
* Сравнительный анализ приведенного метода с существующими, в том числе на основе технологии клеточных автоматов;
* разработка приложения, визуально демонстрирующая работу предложенного метода, а также выдающего на выход последовательность псевдослучайных чисел.

**СОДЕРЖАНИЕ**

[**РЕФЕРАТ 2**](#_Toc136267652)

[**ВВЕДЕНИЕ 5**](#_Toc136267653)

[**1 Аналитический раздел 6**](#_Toc136267654)

[**1.1 Генераторы случайных и псевдослучайных чисел 6**](#_Toc136267655)

[**1.2 Описание предметной области** Ошибка! Закладка не определена.](#_Toc136267656)

[**1.3 Генераторы псевдослучайных чисел 7**](#_Toc136267657)

[**1.4 Криптостойкие генераторы псевдослучайных чисел 16**](#_Toc136267658)

[**1.5 Типы криптостойких ГПСЧ 17**](#_Toc136267659)

[**1.6 Клеточные автоматы 21**](#_Toc136267660)

[**1.7 Игра «Жизнь» 22**](#_Toc136267661)

[**1.8 Вывод 23**](#_Toc136267662)

[**2 Конструкторский раздел 24**](#_Toc136267663)

[**2.1 IDEF0 диаграмма 24**](#_Toc136267666)

[**2.2 Сложности при проектировании 24**](#_Toc136267667)

[**2.3 Модификация правил игры «Жизнь» 27**](#_Toc136267668)

[**2.3.1 Классические правила игры «Жизнь» 27**](#_Toc136267669)

[**2.3.2 Модифицированная игра «Жизнь» 29**](#_Toc136267670)

[**2.4 Выбор начального положения клеток 30**](#_Toc136267671)

[**2.5 Выбор продолжительности «Жизни» 39**](#_Toc136267672)

[**2.6 Вывод 40**](#_Toc136267673)

[**3 Технологический раздел 41**](#_Toc136267674)

[**3.1 Обоснование выбора языка, среды программирования 41**](#_Toc136267676)

[**3.2 Описание используемых модулей 42**](#_Toc136267677)

[**3.2.1 Используемые библиотеки 42**](#_Toc136267678)

[**3.2.2 Реализация модулей 43**](#_Toc136267679)

[**Работа клеточного автомата 44**](#_Toc136267680)

[**Отбор строки 46**](#_Toc136267681)

[**3.3 Реализация критериев для исследования 48**](#_Toc136267682)

[**3.4 Вывод 52**](#_Toc136267683)

[**4 Исследовательский раздел 54**](#_Toc136267684)

[**4.1 Описание среды и устройства для исследования 54**](#_Toc136267689)

[**4.2 Пример работы 55**](#_Toc136267690)

[**4.3 Постановка экспериментов 57**](#_Toc136267691)

[**4.3.1 Критерии оценки качества 57**](#_Toc136267692)

[**4.3.1.1 Частотный побитовый тест 57**](#_Toc136267693)

[**4.3.1.2 Частотный блочный тест 58**](#_Toc136267694)

[**4.3.1.3 Частотный блочный тест 60**](#_Toc136267695)

[**4.3.1.4 Спектральный тест 62**](#_Toc136267696)

[**4.3.1.5 Тест с неперекрывающимися непериодическими шаблонами 64**](#_Toc136267697)

[**4.3.1.6 Тест на периодичность 68**](#_Toc136267698)

[**4.3.1.7 Тест приблизительной энтропии 71**](#_Toc136267699)

[**4.3.1.8 Тест кумулятивных сумм 74**](#_Toc136267700)

[**4.3.1.9 Тест на произвольные отклонения 77**](#_Toc136267701)

[**4.3.1.10 Другой тест на произвольные отклонения 81**](#_Toc136267702)

[**4.4 Анализ результатов исследования 83**](#_Toc136267703)

[**4.5 Сравнительный анализ ГПСЧ с аналогами 85**](#_Toc136267704)

[**4.6 Вывод 86**](#_Toc136267705)

[**ЗАКЛЮЧЕНИЕ 87**](#_Toc136267706)

[**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 89**](#_Toc136267707)

# ВВЕДЕНИЕ

Окружающий нас мир полон неизведанного. Каждый день миллионы ученых и просто любопытных людей трудятся во благо науки. Одни изучают живые организмы, других химические соединения. Результатом многовековых испытаний и исследований всего и вся путем проб, ошибок, изобретений новых и все более совершенных методик изучения природы окружающего мира является современный уровень развития человечества, науки и прочих институтов жизни.

От пещеры до космоса человека вела череда великих открытий и изобретений. Как когда-то человечество изменило навсегда изобретение колеса, так и изобретение компьютера расширило человеческие возможности во всех сферах жизни до небывалых широт, границы которых не изведаны до сих пор.

Компьютер помог завершить Вторую Мировую Войну, отправить человека в космос, реализовать в виртуальной реальности все самые невероятные теории и изучать их.

В многих научных исследованиях, будь то исследования в сфере медицины, моделирования, защиты информации и прочих особую роль занимает поиск случайных чисел или последовательностей случайных чисел. Есть разные способы получения таких чисел, со своими достоинствами и недостатками.

В данной работе предложен метод генерации псевдослучайных чисел, в основе которого лежит технология клеточных автоматов. Проведен обзор существующих генераторов псевдослучайных чисел, их преимуществ и недостатков, выбраны критерии оценки качества генераторов псевдослучайных чисел и проведен сравнительный анализ предложенного генератора с уже существующими, а именно с линейно-конгруэнтным генератором и генератором, также основанным на технологии клеточных автоматов.

# Аналитический раздел

В аналитическом разделе проведен обзор и сравнительный анализ существующих методов генерации псевдослучайных чисел с выделением их преимуществ и недостатков. Сформулирована постановка задачи для разработки метода генерации псевдослучайных чисел.

## **Генераторы случайных и псевдослучайных чисел**

Генератор псевдослучайных чисел (ГПСЧ) — это алгоритм, порождающий последовательность чисел, которая подчиняется заданному распределению и элементы которой почти независимы друг от друга.

В прошлом, для получения последовательности случайных чисел исследователям приходилось прибегать к неудобным и времязатратным способам, таким как перемешивание костей в мешочке и вынимание их оттуда по очереди. Однако с развитием технологии стали возможны способы получения истинно случайных чисел, используя микрофон для захвата окружающего шума или распад радиоактивного вещества в течение определенного времени.

Несмотря на очевидные достоинства, такие методы требуют наличия специального оборудования, содержащего генераторы энтропии, которые могут генерировать не коррелированные и статистически независимые числа, что является нетривиальной задачей. К тому же, для использования истинно случайных чисел в криптографии они должны быть изолированы от внешних влияний, что может быть проблематично.

Кроме аппаратного метода генерации случайных чисел, можно также составить таблицы, которые будут содержать некоррелированные последовательности чисел. Этот метод называется табличным. Однако у такого метода есть недостатки, такие как ограниченность последовательности, занимаемая память, предопределенность значений в случае утечки таблиц и т.д.

Существует также третий тип генераторов - алгоритмический. В основном он представляет собой комбинацию физического генератора и детерминированного алгоритма. Он использует ограниченный набор входных данных от физического генератора и преобразует его в новое значение согласно заданному алгоритму. Хотя этот метод работает быстро, не занимает много памяти и не требует специального оборудования, он генерирует псевдослучайные последовательности чисел, которые имеют свои недостатки, такие как цикличность и зависимость каждого последующего числа от предыдущих.

Стоит отметить, что в России существует ГОСТ для генераторов псевдослучайных чисел. В стандарте установлены методы генерации случайных чисел, подчиняющихся равномерному распределению и другим законам распределения, используемых при применении метода Монте-Карло. Однако криптографические методы генерации в нем отсутствуют. [6]

## **Генераторы псевдослучайных чисел**

Генераторы псевдослучайных чисел широко используются в различных приложениях, таких как игры, криптография, научные и инженерные вычисления, моделирование и другие программы, где требуется случайный элемент.

Криптостойкие генераторы псевдослучайных чисел используются в криптографических протоколах для обеспечения безопасности коммуникации и защиты данных от несанкционированного доступа.

В текущей подразделе рассматриваются различные виды методов генерации псевдослучайных чисел. Представленные ниже методы – это лишь малая часть от общего пантеона ГПСЧ.

В качестве примера можно рассмотреть следующие генераторы псевдослучайных чисел:

* Линейный конгруэнтный;
* Регистр сдвига с обратной линейной связью (РСЛОС);
* Вихрь Мерсенна;
* ГПСЧ на базе клеточного автомата (правило 30);
* ГПСЧ на базе клеточного автомата (NESW)

**Линейный конгруэнтный**

Линейная конгруэнтная схема генерации псевдослучайных чисел была предложена Д. Г. Лехмером в 1949 году и считается наиболее популярной. Согласно этой схеме последовательность случайных чисел можно получить, если положить, что

,

где

В приведенной выше формуле особое значение имеет выбор параметров. Например, для получится последовательность:

7, 5, 9, 1, 7, 5, 9, 1, …

Как можно заметить, данную последовательность с трудом можно назвать «случайной». В этом примере кроется проблема данного метода – линейный конгруэнтный метод всегда выдает последовательность, которая зацикливается.

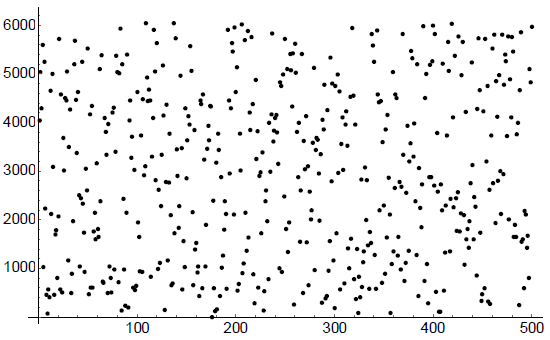
****

Рисунок 1.1. График ЛКП для X¬0= 7,a=106,c= 1283,m = 6075.

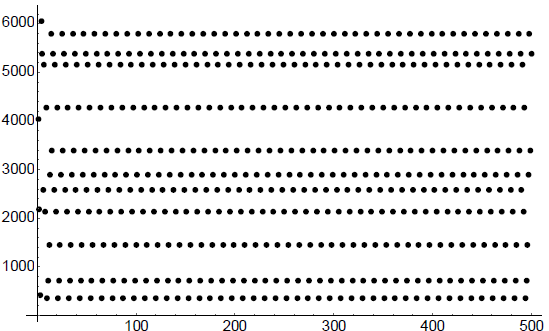


Рисунок 1.2. График ЛКП для X¬0= 7,a=105,c= 1283,m = 6075.

Также, ввиду свойств операций в конечном поле, применяемых в формуле (1), возникает «решетчатая структура» в последовательностях, что также является недостатком. Данный эффект можно наблюдать на рисунке 1.2.1.3

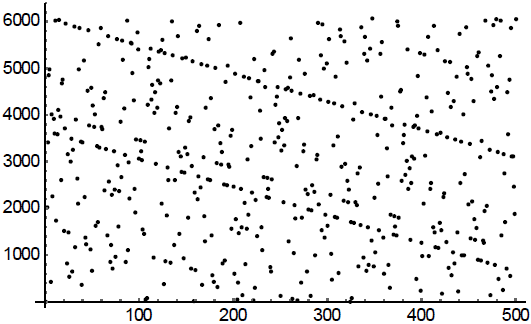
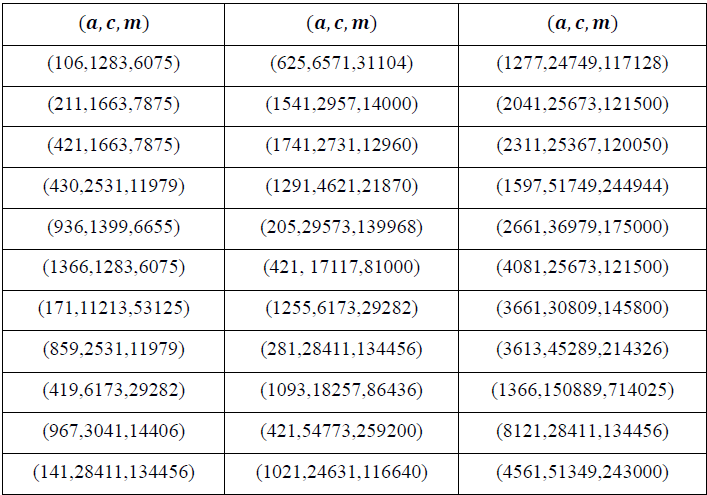


Рисунок 1.3. График ЛКП для X¬0= 7,a=106,c= 1284,m = 6075.

Ниже приведена таблица констант для линейных конгруэнтных генераторов. [4]

Таблица 1.1. Параметры ЛКГ для формулы 1.



Если подытожить, то можно в качестве преимуществ выделить простоту реализации и скорость работы данного генератора. Недостатком же будет являться цикличность значений последовательности.

**Регистр сдвига с обратной линейной связью**

Данный класс ГСПЧ основан на преобразовании бинарного представления некоторого числа. Такие генераторы имеют некоторые преимущества, как, например, скорость генерации чисел, хорошие статистические свойства псевдослучайных чисел, а также возможность простой реализации на аппаратном уровне.

Регистр сдвига с обратной линейной связью (РСЛОС) – регистр сдвига битовых слов, у которого входной (вдвигаемый) бит является линейной функцией остальных битов. Вдвигаемый вычисленный бит заносится в ячейку с номером 0. Количество ячеек p называют длиной регистра. [4]

Для натурального числа и , принимающих значения 0 или 1, определяют рекуррентную формулу

,

Из формулы можно заключить, что для РСЛОС функция обратной связи является линейной булевой функцией от состояний всех или некоторых битов регистра.

Одна итерация алгоритма, генерирующего последовательность, состоит

из следующих шагов:

1. Содержимое ячейки − 1 формирует очередной бит ПСП битов.

2. Содержимое ячейки 0 определяется значением функции обратной связи, являющейся линейной булевой функцией с коэффициентами . Его вычисляют по вышеприведенной формуле.

3. Содержимое каждого i-го бита перемещается в (*i* + 1)-й, 0 ≤ *i* < − 1.

4. В ячейку 0 записывается новое содержимое, вычисленное на шаге 2.

Наименьшее положительное целое N, такое, что для всех значений называют *периодом последовательности*. Эту последовательность называют М-последовательностью.

**Вихрь Мерсенна**

В 1997 году японскими учеными Макото Мацумото и Такудзи Нимисура был предложен метод генерации случайных чисел, основанный на свойствах простых чисел Мерсенна. Данный метод получил название «Вихревой генератор». «Вихрь» – это преобразование, которое обеспечивает равномерно распределение ПСЧ.

Числом Мерсенна называется натуральное число , определяемое формулой

Пример. Первые 17 чисел последовательности: 1, 3, 7, 15, 31, 63, 127, 255, 511,1023, 2047, 4095, 8191, 16 383, 32 767, 65 535, 131 071. [4]

Часто числами Мерсенна называют числа с простыми индексами n. Важным свойством чисел Мерсенна является то, что если является простым, то значит, что и n – также простое число. В общем случае обратное не верно, что не дает возможности просто и эффективно генерировать простые числа, но зато позволяет эффективно проверять число на простоту. Данное свойство лежит в основе теста на простоту Люка-Лемера. [1]

Существует несколько вариантов этого ГПСЧ. Мы рассмотрим наиболее распространенный, который имеет обозначение MT19937. По сути, данный ГПСЧ является РСЛОС, состоящим из 624 ячеек по 32 бита. Метод Вихрь Мерсенна позволяет генерировать последовательность двоичных псевдослучайных целых w-битовых чисел в соответствии со следующей рекуррентной формулой

где – целые константы, – степень рекуррентности, ;

– w-битовое двоичное целое число;

– двоичное целое число, полученное конкатенацией чисел и , когда первые (w-r) битов взяты из , а последние битов из в том же порядке;

– матрица размера w×w, состоящая из нулей и единиц, определенная по-

средством ;

– произведение, при вычислении которого сначала выполняют операцию (сдвига битов на одну позицию вправо), если последний бит равен

0, а затем, когда последний бит , вычисляют .

Вихрь Мерсенна имеет огромный период, равный числу Мерсенна (). Этот период достаточен для большинства возможных применений

алгоритма.

Метод обеспечивает равномерное распределение генерируемых псевдослучайных чисел в 623 измерениях. Поэтому корреляция между последовательными значениями в выходной последовательности Вихря Мерсенна пренебрежимо мала. Метод также хорошо проходит статистические тесты на «случайность».

Однако данный ГПСЧ не предназначен для получения криптографически стойких последовательностей случайных чисел. [12]

**ГПСЧ на базе клеточного автомата («правило 30»)**

В работах [7],[8],[9] и [10] описан метод генерации псевдослучайных чисел, предложенный Стивеном Вольфрамом. В его основе лежит клеточный автомат «правило 30».

Клеточный автомат – это устройство, состоящее из n-мерного массива ячеек и правил изменения значений ячеек. Каждая из ячеек имеет начальное состояние и изменяет свое состояние в дискретные моменты времени.

Правило, в соответствии с которым ячейка изменяет состояние, – это рекуррентная формула, в которой новое значение ячейки определяется исходя из предыдущих значений этой и соседних ячеек. В простом одномерном случае, когда массив ячеек состоит из n битов , правило выглядит так:

Этот генератор показывает хорошие статистические свойства. Однако он сильно зависим от начальных значений ячеек: при неудачном выборе начального состояния, клеточный автомат порождает циклические структуры. Кроме того, для него существует успешное вскрытие с известным открытым текстом. В [8, 9] показано, что вскрытие выполнимо на персональном компьютере с размером клеточного автомата вплоть до 500 битов.

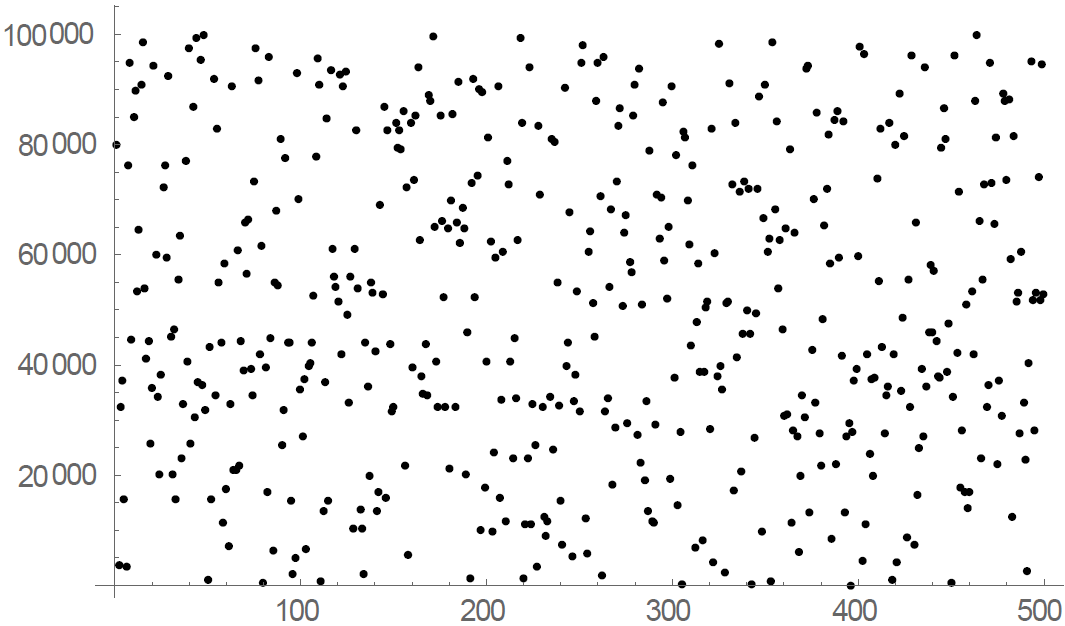


Рисунок 1.4. График ГПСЧ на базе клеточного автомата

**ГПСЧ на базе клеточного автомата (NESW)**

Одним из относительно молодых методов генерации псевдослучайных чисел, также основанном на технологии клеточного автомата, является метод, представленный университетом ИТМО за авторством Д. Д. Мухамеджанова, А. Б. Левиной. В сути своей данный метод является глубокой модификацией предыдущего метода. Сетка имеет размеры p и q , которые являются простыми числами (для улучшения периодичности). Изменения начинаются прямо с этого этапа: деление всей сетки на блоков, которые формируются как прямоугольники с размерами и , а каждый блок состоит из lbi wbi  ячеек (рис. 1.5).

Выбрав произведение сторон прямоугольников блоков lbi  wbi как начальную конфигурацию, необходимо заполнить ячейки таким образом, чтобы в каждой ячейке располагался один бит (состояние 0 или 1).

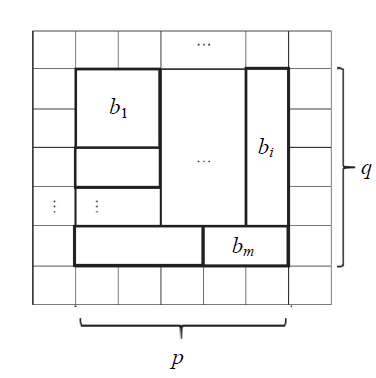


Рисунок 1.5. Разделение сетки клеточного автомата на блоки

Затем применяется алгоритм под названием NESW (North, East, South, West). Метод такого движения носит название по сторонам света (рис. 1.6). Суть способа в том, чтобы заполнять сетку блока согласно направлениям сторон света, т.е. мы циклически заполняем битами полученного числа сетку, начиная с левого нижнего угла, двигаясь сначала вверх, затем вправо, вниз и влево до конца сетки (или до заполненной ячейки).

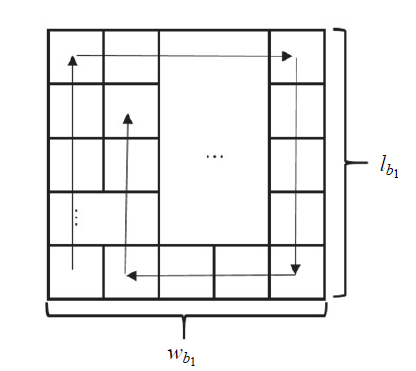


Рисунок 1.6. Метод NESW

Каждый блок может иметь свое собственное правило, что делает КА неоднородным и приводит к лучшим показателям критериев статистических свойств и лучшим периодическим свойствам. [11]

## **Криптостойкие генераторы псевдослучайных чисел**

Криптостойкие генераторы псевдослучайных чисел (КГПСЧ) используются в криптографии для генерации псевдослучайных чисел, которые могут использоваться для шифрования данных и защиты от несанкционированного доступа. КГПСЧ обладают особенными свойствами, которые делают их надежными и безопасными при использовании в криптографии.

## **Типы криптостойких ГПСЧ**

Существует несколько типов криптостойких генераторов псевдослучайных чисел, которые различаются по состоянию, которое они используют для генерации чисел.

**Односторонние хэш-функции**

Односторонние хэш-функции (ОХФ) являются одним из наиболее широко распространенных типов криптостойких генераторов псевдослучайных чисел. Они генерируют псевдослучайные числа, используя входной ключ и текущее состояние.

Ключ и текущее состояние используются для обновления состояния генератора. При этом применяется односторонняя хэш-функция, которая выполняет преобразование текущего состояния в новое состояние. Новое состояние используется для генерации следующего числа.

Среди преимуществ данного метода можно выделить следующее:

* Несмотря на то, что исходное сообщение может быть произвольной длины, хеш-функция генерирует фиксированный хеш-код заданной длины. Это позволяет использовать хеш-функции для определения целостности данных и проверки наличия изменений;
* В зависимости от выбранной функции, хеш-значение может быть очень сложно предсказать. Это делает хеш-функции полезными для генерации псевдослучайных чисел;
* Хеши являются необратимыми - т.е. по значению хеша невозможно восстановить исходное сообщение. Это свойство является ключевым для обеспечения безопасной передачи данных и аутентификации клиентов.

Среди же недостатков следует отметить следующее:

* По определению хеш-функций, несколько сообщений могут иметь одинаковый хеш-значение — это называется коллизия. Чем сложнее функция хеширования, тем меньше вероятность возникновения коллизий, но на практике они могут происходить. Это может стать уязвимостью для систем, которые зависят от уникальности хеш-кода для каждого сообщения;
* Хеш-функции также могут быть подвержены "атакам по словарю" - методу перебора множества возможных входных данных до нахождения их хеш-значения. Для устранения этого недостатка может использоваться "соление" - добавление случайного секретного значения в исходное сообщение перед хешированием;
* Некоторые современные криптографические алгоритмы могут быть уязвимыми к атакам с использованием специальных аппаратных средств, которые могут находить коллизии с высокой скоростью.

Один из примеров криптостойкой ОХФ – алгоритм SHA-2.

**Генераторы на основе математических задач**

Генераторы псевдослучайных чисел на основе сложных математических задач используют сложные вычисления для создания чисел, которые будут выглядеть случайными и не поддадутся криптоанализу. Примерами таких генераторов являются алгоритм Блюма-Блюма-Шуба и алгоритм Блюма-Микали. Рассмотрим детальнее один из этих генераторов.

Алгоритм Блюм-Блюма-Шуба (BBS) является одним из наиболее известных криптографических генераторов псевдослучайных чисел. Он основывается на трудной задаче факторизации больших целых чисел и является стойким к атакам по крайней мере на сегодняшний день.

Применение BBS имеет следующие преимущества:

* 1. Простота: алгоритм BBS достаточно прост в реализации, а входные данные легко регулируемы, в связи с чем алгоритм может быть удобен для различного рода исследований;
  2. Псевдослучайность: числа, которые генерируются BBS, не могут быть предсказаны заранее, что делает их настоящими случайными числами.
  3. Быстродействие: BBS генерирует большое количество случайных чисел за короткое время;

Однако, есть и недостатки BBS:

1. Значительные вычислительные ресурсы: генерация чисел происходит путем выполнения сложных математических операций, что требует большого количества вычислительных ресурсов.
2. Низкая скорость генерации чисел: время генерации каждого числа зависит от количества бит в последовательности. Также для генерации большого количества случайных чисел требуется много времени.
3. Неудобство использования: BBS может быть неудобным для использования в реальном времени, поскольку время генерации каждого числа может достигать нескольких секунд.

Итак, преимущества BBS включают безопасность, псевдослучайность и эффективность, а недостатки - значительные вычислительные ресурсы, низкую скорость генерации чисел и неудобство использования.

**Энтропийные коллекторы**

Энтропийные коллекторы – это специализированные устройства или программы, которые собирают случайные данные из различных источников и используют их для генерации псевдослучайных чисел.

Один из примеров энтропийных коллекторов – алгоритм Fortuna.

Fortuna основывается на хэш-функции SHA-256 (Secure Hash Algorithm) и имеет структуру, состоящую из нескольких независимых генераторов, работающих параллельно. Каждый генератор использует отдельный вектор и периодически обновляется новыми случайными данными. Такая структура делает Fortuna стойким к атакам и обеспечивает высокую скорость генерации случайных чисел.

У данного алгоритма есть несколько преимуществ:

1. Стойкость к атакам: многие криптографические алгоритмы могут быть взломаны при наличии достаточной вычислительной мощности. Fortuna, однако, является стойким к атакам благодаря использованию нескольких независимых генераторов, работающих параллельно.
2. Высокая скорость генерации: благодаря параллельной работе нескольких генераторов Fortuna может генерировать большие объемы случайных чисел за короткое время.
3. Автоматическое обновление ключей: Fortuna автоматически обновляет свои ключи после каждой генерации случайных чисел, что улучшает безопасность генерируемых данных.
4. Энтропия: Fortuna использует множество источников энтропии для генерации случайных чисел, что улучшает их качество.

Однако, есть и некоторые недостатки Fortuna:

1. Низкая скорость восстановления: если входной источник энтропии прекращает свою работу, то требуется определенное время для восстановления необходимого уровня энтропии.
2. Сложность реализации: Fortuna является сложным и трудоемким для реализации алгоритмом, что может сделать его менее привлекательным для использования в некоторых приложениях.

Каждый из вышеупомянутых типов КСГПСЧ имеет свои преимущества и недостатки, которые могут помочь при выборе соответствующего генератора в зависимости от конкретного случая использования.

## **Клеточные автоматы**

Клеточные автоматы – это дискретные динамические системы, поведение которых полностью определяется в терминах локальных зависимостей, в значительной в значительной степени так же обстоит дело для большого класса непрерывных динамических систем, определенных уравнениями в частных производных. [2]

КА можно формально описать как четверку

, (1)

где – множество мерных векторов, = - множество состояний одной ячейки в , – окрестность или шаблон соседства (упорядоченное множество различных -мерных векторов из , – локальная функция переноса.[3-4]

Из данного выше описания становится понятно, что на очередном шаге состояние каждой клетки зависит от состояния клеток вокруг нее – ее окрестности. Обычно различают два вида окрестностей: окрестность Мура и окрестность фон Неймана (рисунок 1.7). [3]

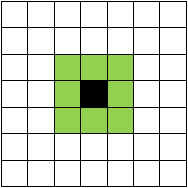
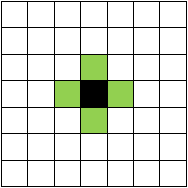


Рисунок 1.7 Окрестности фон Неймана (слева) и Мура (справа)

## **Игра «Жизнь»**

Игра «Жизнь» — это клеточный автомат, изобретенный в 1970 году английским математиком Джоном Конвеем, в котором соблюдается ряд определенных правил поведения системы.

Правила игры таковы:

* «Вселенная» игры — это размеченная на клетки поверхность или плоскость. Она может быть безграничная, ограниченная или замкнутая.
* Каждая клетка на этой поверхности может находиться в двух состояниях: быть «мёртвой» или «живой». Клетка имеет восемь соседей, окружающих её.
* Распределение живых клеток в начале игры называется первым поколением. Каждое следующее поколение рассчитывается на основе предыдущего по следующим правилам:
  + Если рядом с мёртвой клеткой рядом находятся ровно три живые клетки, то в данной клетке зарождается жизнь;
  + если вокруг живой клетки два или три живых соседки, то эта клетка не умирает;
  + клетка умирает, если соседей меньше двух («от одиночества») или больше трёх («от перенаселённости»);
* Игра прекращается, если
  + Во «вселенной» не останется ни одной живой клетки;
  + складывается периодическая конфигурация;
  + складывается стабильная конфигурация;

Благодаря этим простым правилам можно добиться большого разнообразия «форм жизни».

Стоит также отметить, что игрок занимает позицию наблюдателя и не принимает прямого участия в игре. Он может лишь формировать начальную конфигурацию живых клеток.

В результате выполнения классических условий игры через некоторое количество ходов по выбору методики генерирования получается последовательность псевдослучайных чисел из нулей и единиц. Эта особенность данного клеточного автомата и будет применяться в предложенном методе.

## **Вывод**

В разделе дано описание генераторов случайных и псевдослучайных чисел, а также клеточных автоматов. Рассмотрены особенности и проблемы генерации истинно случайных чисел, разновидности методов генерации псевдослучайных чисел. Среди разновидностей рассмотренных методов приведены два метода, также основанных на технологии клеточных автоматов. Рассмотрены обобщенный клеточный автомат, а также частный случай его реализации – игра «Жизнь», которая лежит в основе предложенного метода. Приведены правила игры «Жизнь». Определены ограничения и требования.

# Конструкторский раздел

В конструкторском разделе проведена формализация задачи – рассмотрены проблемы и способы их решения. Проведен разбор используемых алгоритмов, предложены модификации для улучшения их работы. Представлены IDEF0 диаграммы, рисунки, изображающие ход работы клеточного автомата и игры «Жизнь», а также тот или иной выбор конфигурации клеточного автомата и последующие результаты спустя поколения.



## **IDEF0 диаграмма**

На рисунках 2.1 – 2.3 представлена IDEF0 диаграмма.

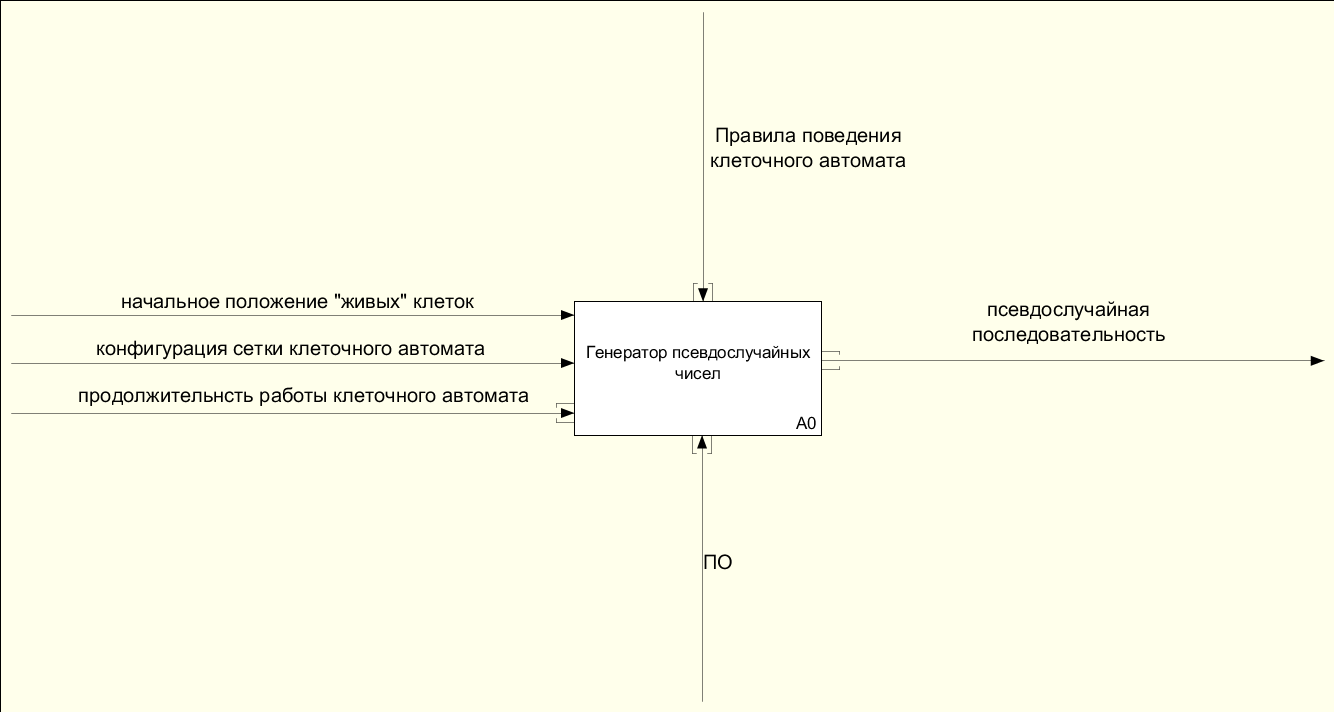


Рисунок 2.1 – Модель работы генератора псевдослучайных чисел, уровень 0

Генератор псевдослучайных чисел представляет собой клеточный автомат, правила поведения для которого заданы изначально и работают без вмешательства человека.

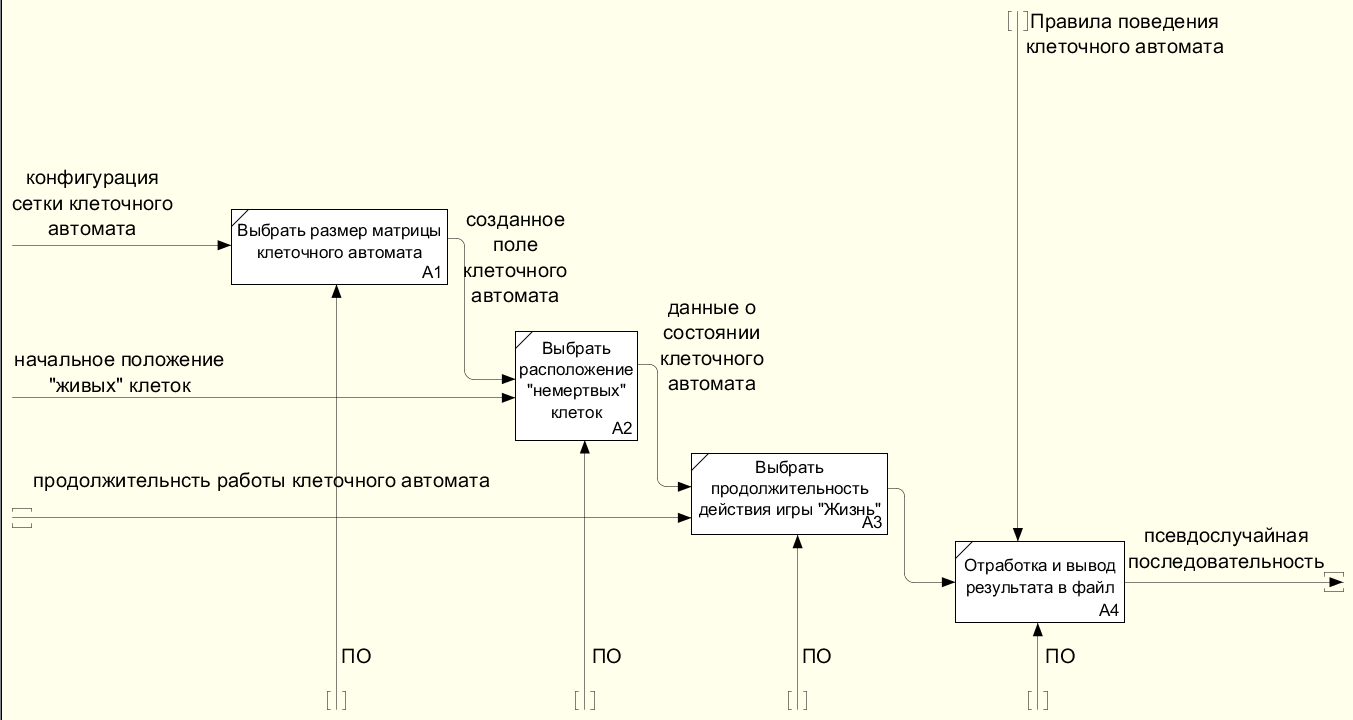


Рисунок 2.2 – Модель работы генератора псевдослучайных чисел, уровень 1

Для того, чтобы генератор выдал результат, клеточному автомату требуется задать ряд параметров, это:

* Размеры клеточного автомата;
* Положение ненулевых значений в клеточном автомате;
* Поведение клеточного автомата, представленного игрой «Жизнь»;
* Продолжительность игры

Клеточный автомат представляет собой матрицу, где каждый элемент представляет является отдельной клеткой. Правила клеточного автомата задаются условной конструкцией, рассмотренной в подразделе 2.3.2.

В результате работы программы выбирается одна строчка из матрицы, ее значение в виде тернарной последовательности записывается в текстовый файл формата .txt.

## **Сложности при проектировании**

При создании предложенного метода возник ряд проблем, требующих решения, а именно:

* Выбор начального положения не мертвых клеток. В рамках классической игры «Жизнь» эта проблема также актуальна, ибо начальное положение должно быть подобрано так, чтобы наблюдаемая «вселенная» прожила достаточное большое количество поколений. С точки зрения игры, это нужно, дабы добиться устойчивых форм жизни. Для предложенного же метода это нужно, чтобы клеточный автомат не был заполнен преимущественно нулями, тем самым улучшая результирующую случайность;
* Продолжительность жизни «вселенной». Добившись правильного начального состояния клеточного автомата, при котором система может существовать достаточно долго, нужно также и ограничить продолжительность существования «вселенной», так как для результата в виде псевдослучайного числа или последовательности из таких чисел не требуется ожидать завершения полного цикла ее «вселенной». Помимо этого, также это не позволяет получать результат быстро. Поэтому требуется находить такую конфигурацию клеток и времени работы генератора, при котором случайность может быть получена как можно быстрее;
* Видимая случайность. При всей, казалось бы, случайности результата игры «Жизнь», тем не менее, зачастую, конечная матрица клеточного автомата остается заполненной нулями. Не хватает «шума», который повысил бы случайность результирующей картины. Необходимо пересмотреть правила игры «Жизнь».

Несмотря на порядок выделенных проблем рассматриваться их решения будут в обратном порядке. Решение третьей проблемы будет приведено в разделе 2.3

Для решения второй и первой проблемы требуется ряд экспериментов, в ходе которых можно выделить нужную начальную конфигурацию всей системы, что и будет рассматриваться в разделе 2.4.

## **Модификация правил игры «Жизнь»**

В данном подразделе будут рассматриваться изменения, примененные к классической игре «Жизнь», сравнение модифицированной версии и классической. Благодаря приведенным ниже модификациям предложенный метод и добивается более «качественной» случайности, чем классическая игра «Жизнь». Продемонстрировано, как правила игры влияют на развитие «вселенной» игры.

### Классические правила игры «Жизнь»

Как уже отмечалось в аналитическом разделе, классическая игра имеет следующий ряд правил:

* «Вселенная» игры — это размеченная на клетки поверхность или плоскость. Она может быть безграничная, ограниченная или замкнутая.
* Каждая клетка на этой поверхности может находиться в двух состояниях: быть «мёртвой» или «живой». Клетка имеет восемь соседей, окружающих её.
* Распределение живых клеток в начале игры называется первым поколением. Каждое следующее поколение рассчитывается на основе предыдущего по следующим правилам:
  + Если рядом с мёртвой клеткой рядом находятся ровно три живые клетки, то в данной клетке зарождается жизнь;
  + если вокруг живой клетки два или три живых соседки, то эта клетка не умирает;
  + клетка умирает, если соседей меньше двух («от одиночества») или больше трёх («от перенаселённости»);
* Игра прекращается, если
  + Во «вселенной» не останется ни одной живой клетки;
  + складывается периодическая конфигурация;
  + складывается стабильная конфигурация;

Результат работы этого клеточного автомата, работающего по таким правилам, можно наблюдать в рисунках 2.3-2.4

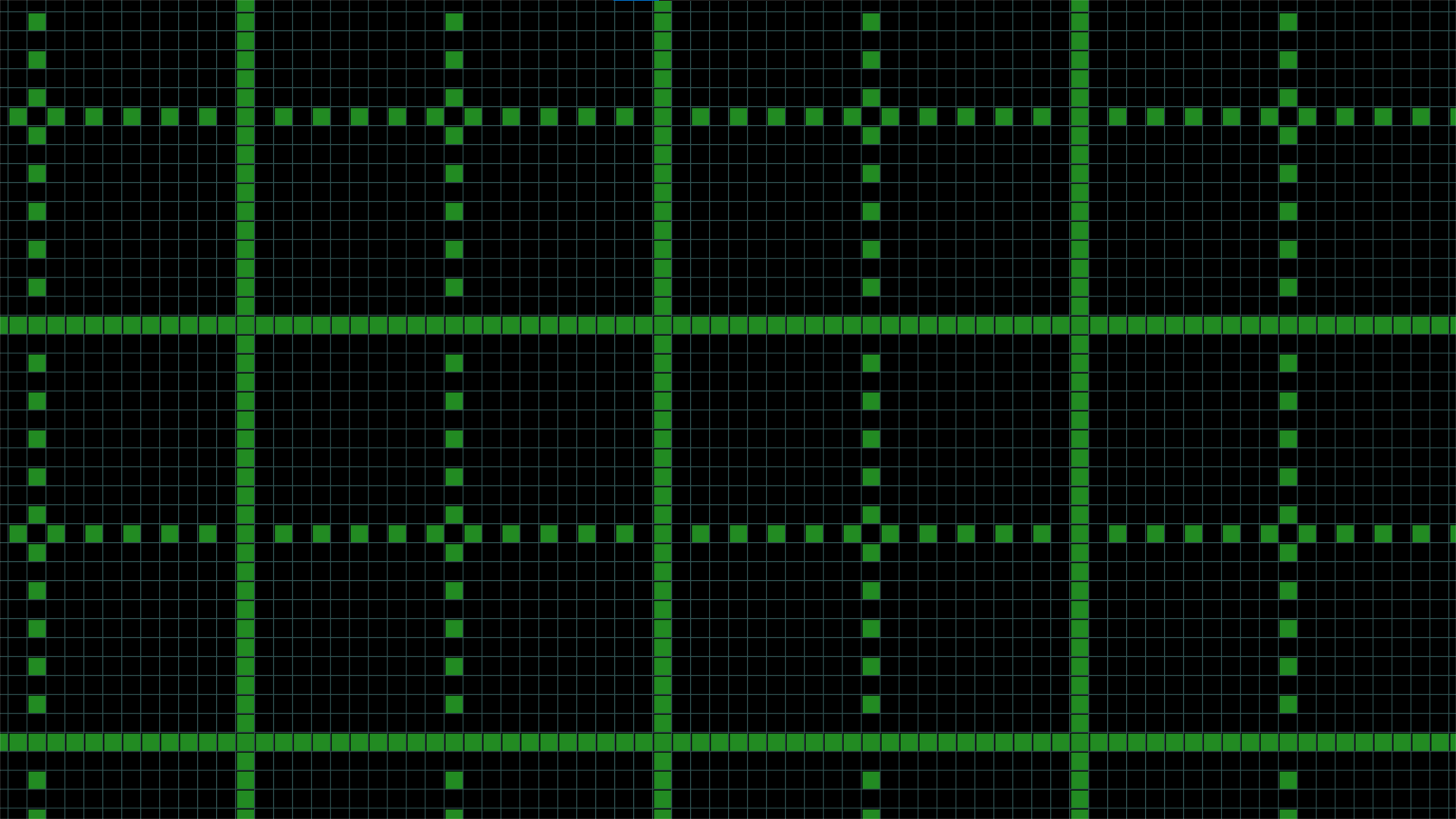


Рисунок 2.3 – Начальное состояние «вселенной»

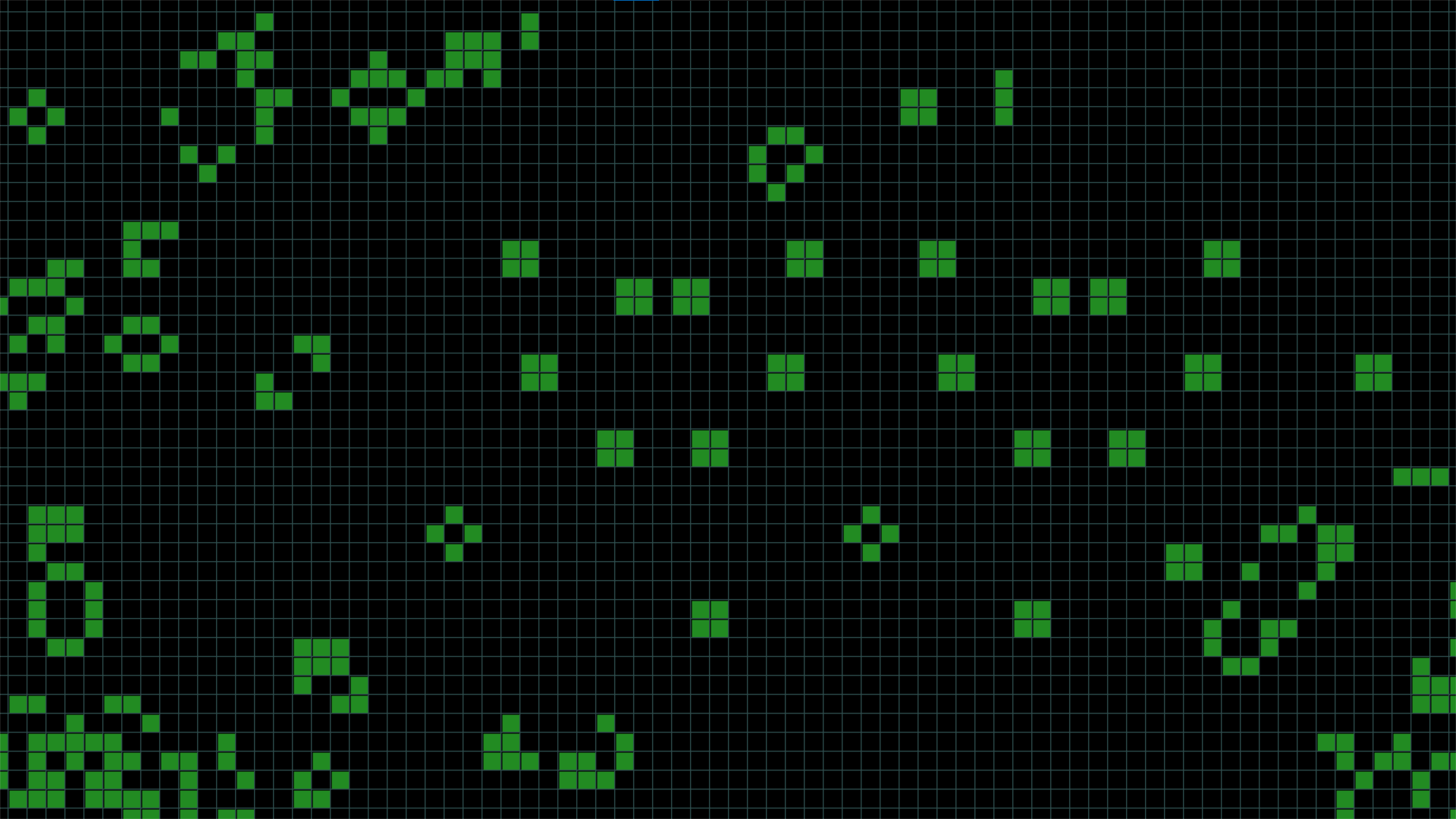


Рисунок 2.4 – Состояние «вселенной» спустя 100 поколений

Как можно заметить, в ходе развития популяции «живых» клеток, возникают устойчивые формы, такие как блок, ящик, каравай. Их расположение, равно как и положение просто «живых» клеток видится случайным, однако вряд ли с таким большим количеством нулевых клеток можно добиться хороших показателей ГПСЧ.

### Модифицированная игра «Жизнь»

В предложенном методе генерации псевдослучайных чисел произведен следующий набор модификаций для игры «Жизнь»:

* Введены новые персонажи игры – «Зомби»;
* Введены новые правила игры с учетом новых персонажей

Начнем с первого – с «зомби» клеток. «Зомби» клетки имеют состояние равное 2. В сути своей они являются противниками «живых» клеток и между ними начинается «война». В ходе этой «войны» противники не только захватывают новые «территории» в виде клеток, но и убивают или вербуют клетки соперников.

Благодаря такому нехитрому способу можно наблюдать интересную картину:

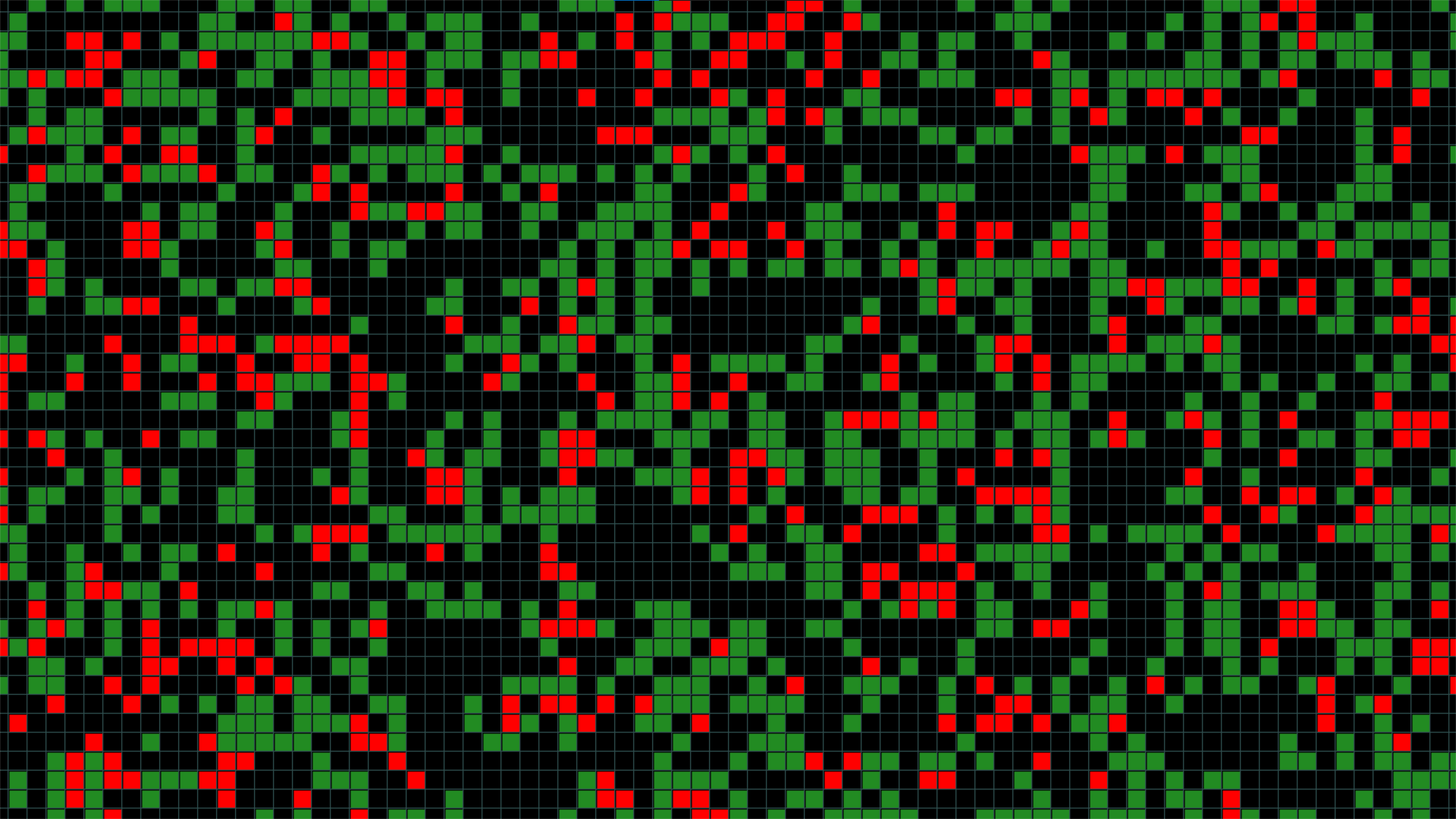


Рисунок 2.5 – Состояние «вселенной» спустя 100 поколений согласно новым условиям игры

При прочих равных, включая и правила поведения «живых» клеток, которые также остались неизменными, изменилось общее состояние клеточного автомата: уменьшилось количество «мертвых», нулевых клеток и увеличилось количество клеток, занятых каким-либо значением (1 или 2), даже визуально полученный рисунок кажется более случайным и определить какую-либо закономерность становится куда сложнее.

По итогу, такими простыми манипуляциями из такого клеточного автомата можно получать последовательности не бинарные, но тернарные.

## **Выбор начального положения клеток**

От выбора начально положения немертвых клеток зависит то, как долго будет жить «вселенная», как быстро будет получена достаточно случайный результирующий рисунок и насколько случайным он окажется. К сожалению, данная проблема не решается теоретически, а потому выбор будет производиться исходя из результатов эмпирических исследований.

Далее будут показаны некоторые из вариантов начальной конфигурации клеточного автомата и ход их «жизни» от зарождения вселенной до ее «развития».

1. Расположение «живых» клеток крестом:

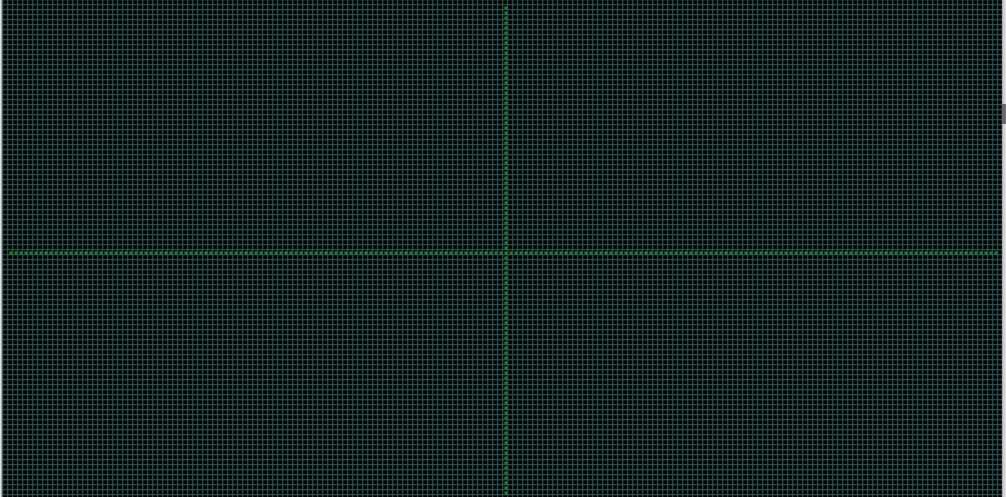


Рисунок 2.5 – Вариант 1, поколение 0

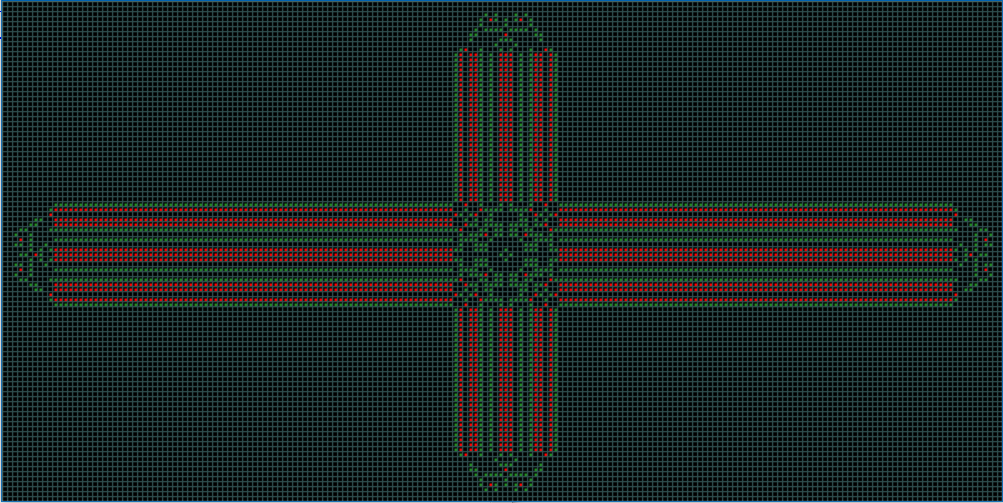


Рисунок 2.6 – Вариант 1, поколение 10

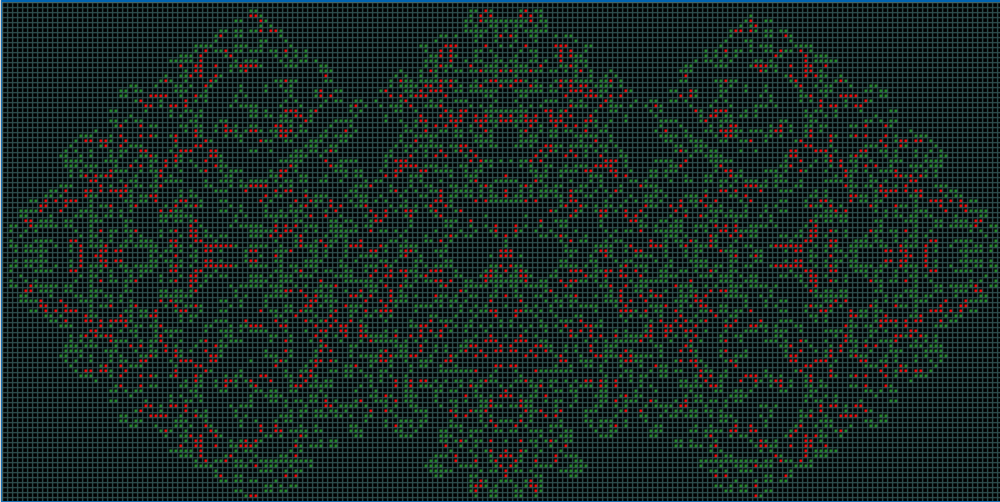


Рисунок 2.7 – Вариант 1, поколение 50

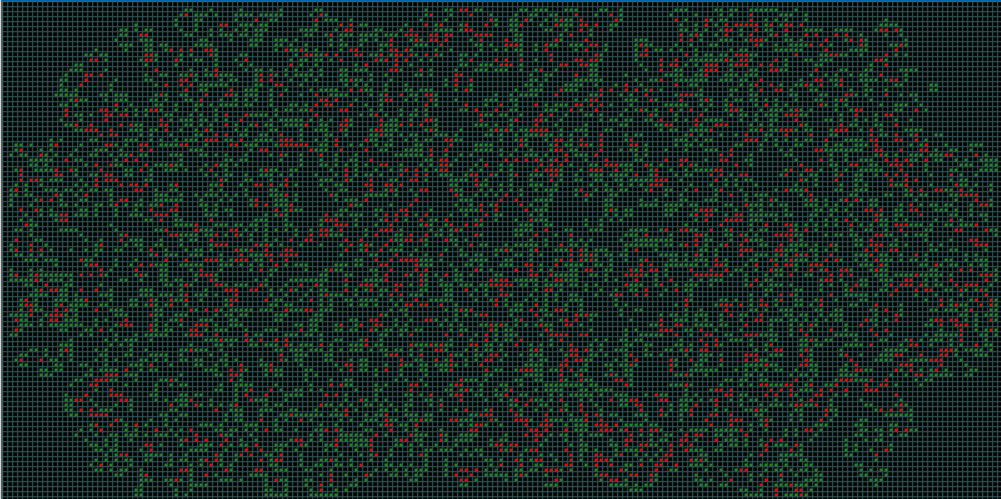


Рисунок 2.8 – Вариант 1, поколение 100

В рисунках 2.5 – 2.8 видно, как по мере развития «вселенная» становится все менее симметричной и все более случайной. Однако для того, чтобы дождаться несимметричной картины, понадобилось около 100 поколений, что является достаточно затратным процессом.

1. Расположение «сеткой»

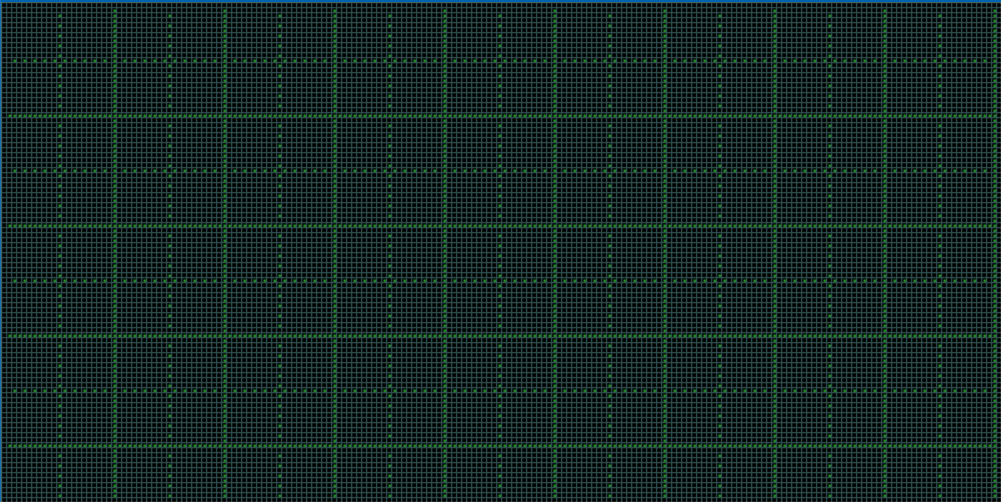


Рисунок 2.9 – Вариант 2, поколение 0



Рисунок 2.10 – Вариант 2, поколение 5

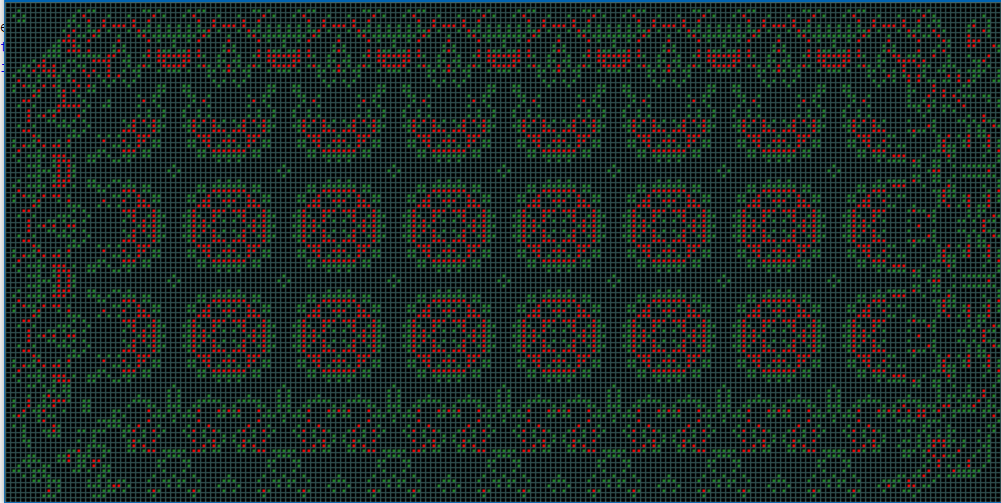


Рисунок 2.11 – Вариант 2, поколение 25

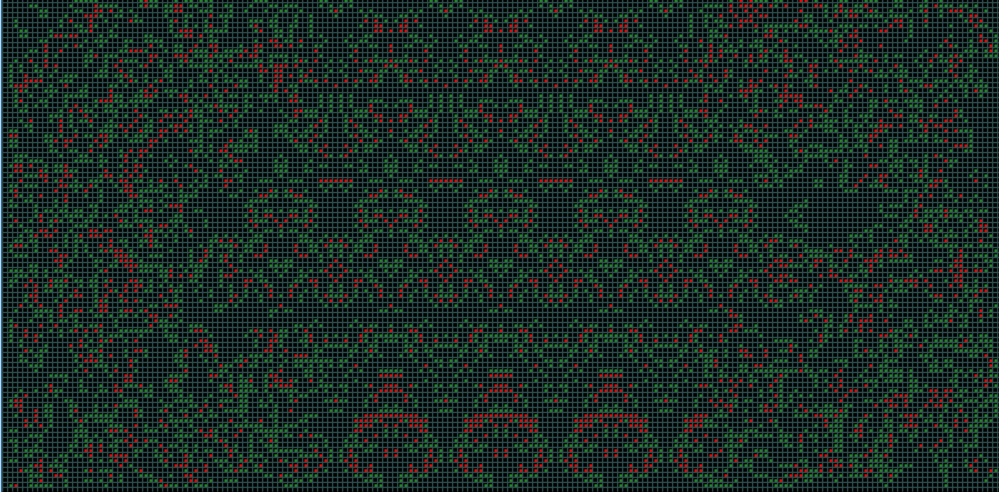


Рисунок 2.12 – Вариант 2, поколение 65

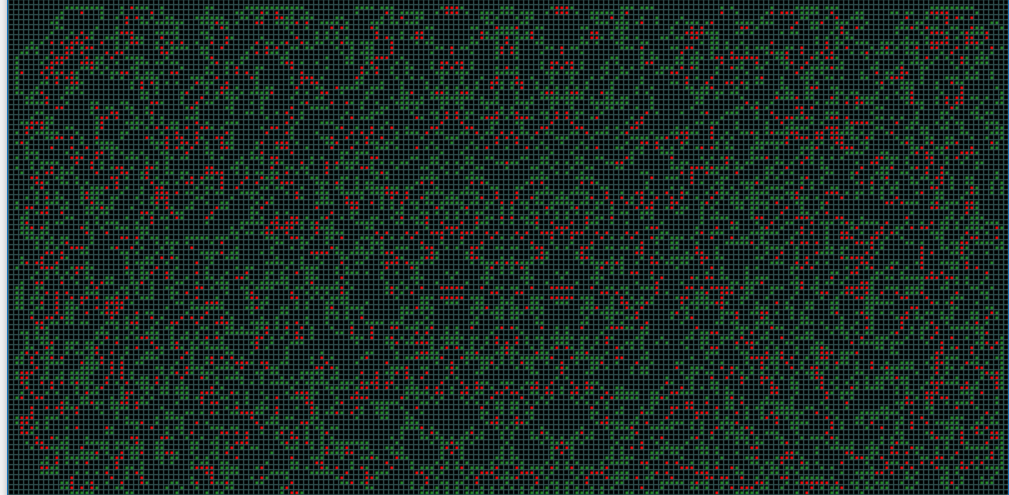


Рисунок 2.13 – Вариант 2, поколение 85

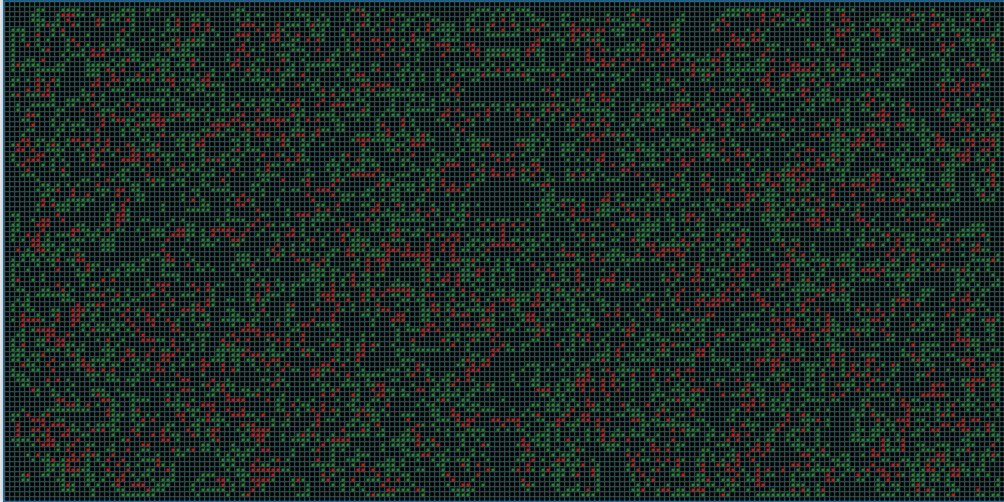


Рисунок 2.13 – Вариант 2, поколение 100

Второй вариант из симметричного рисунка, напоминающего персидский ковер, с каждым поколением все больше и больше превращался в полотно случайно расставленных «живых» и «зомби» персонажей, причем случайность возникает от краев поля к ее центру. Тем не менее если приглядеться, то вплоть до 85 поколения можно наблюдать в некоторых участках симметричное изображение, и даже в сотом поколении в некоторых малых участках также симметричность присутствует. Тем не менее, в отличии от первого варианта, данная конфигурация покрывает все поле, в то время как первый вариант, включая такое же сотое поколение, оставляет пустыми все четыре угла «вселенной».

1. «Живые» клетки на четных позициях

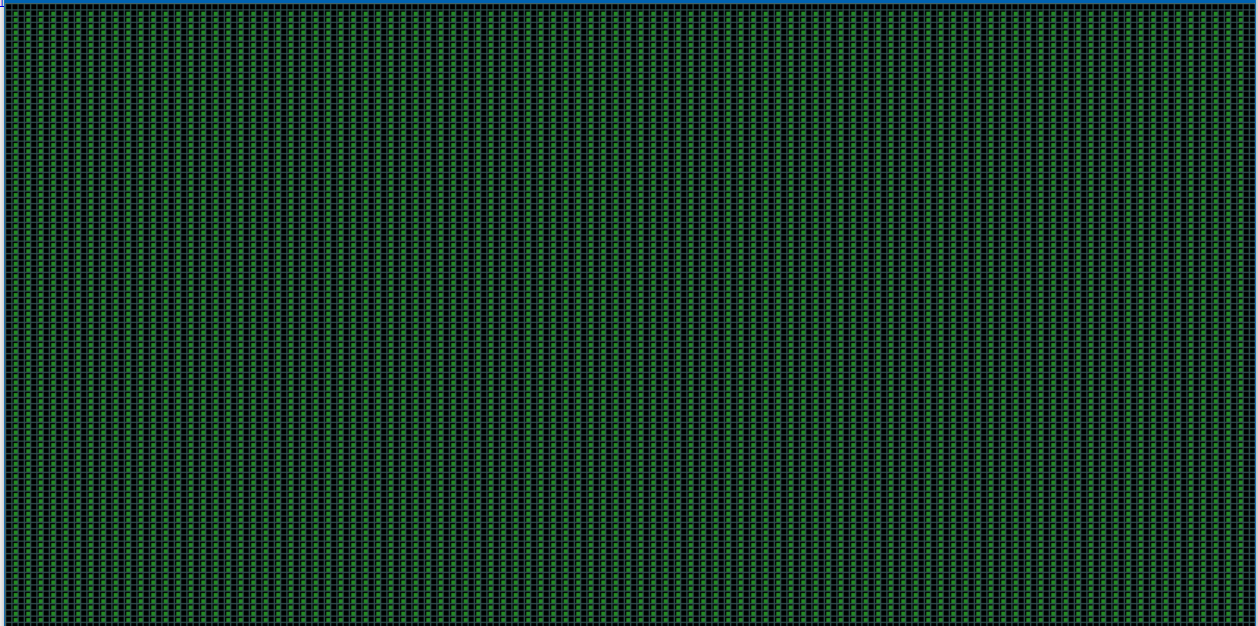


Рисунок 2.14 – Вариант 3, поколение 0

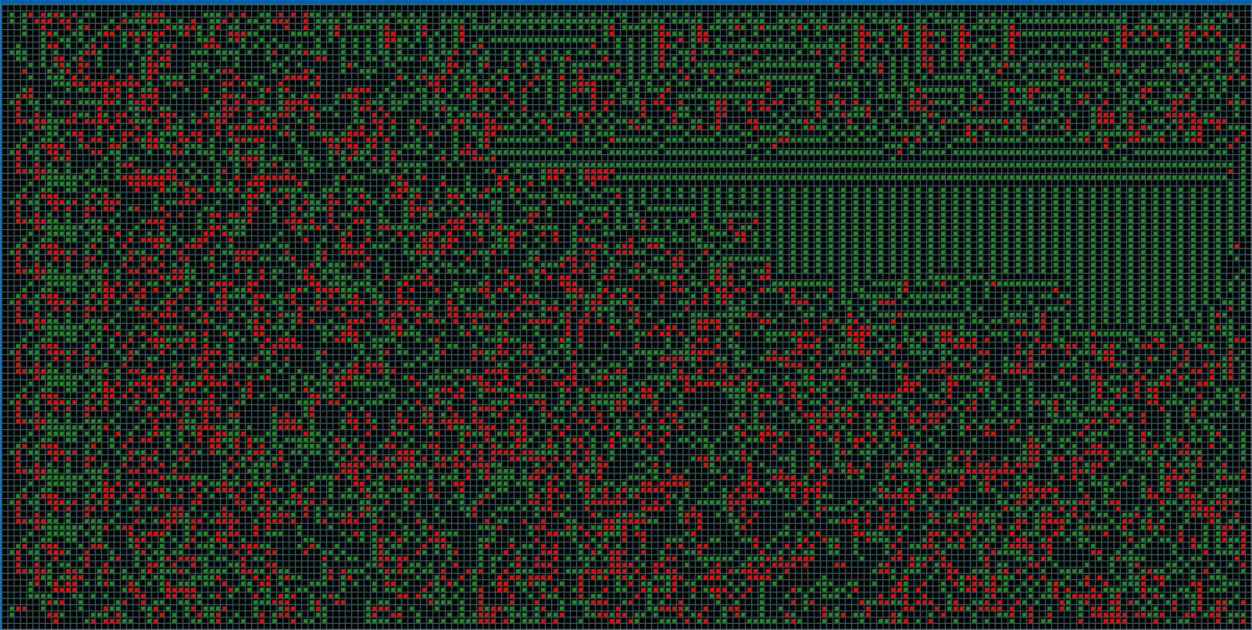


Рисунок 2.15 – Вариант 3, поколение 5

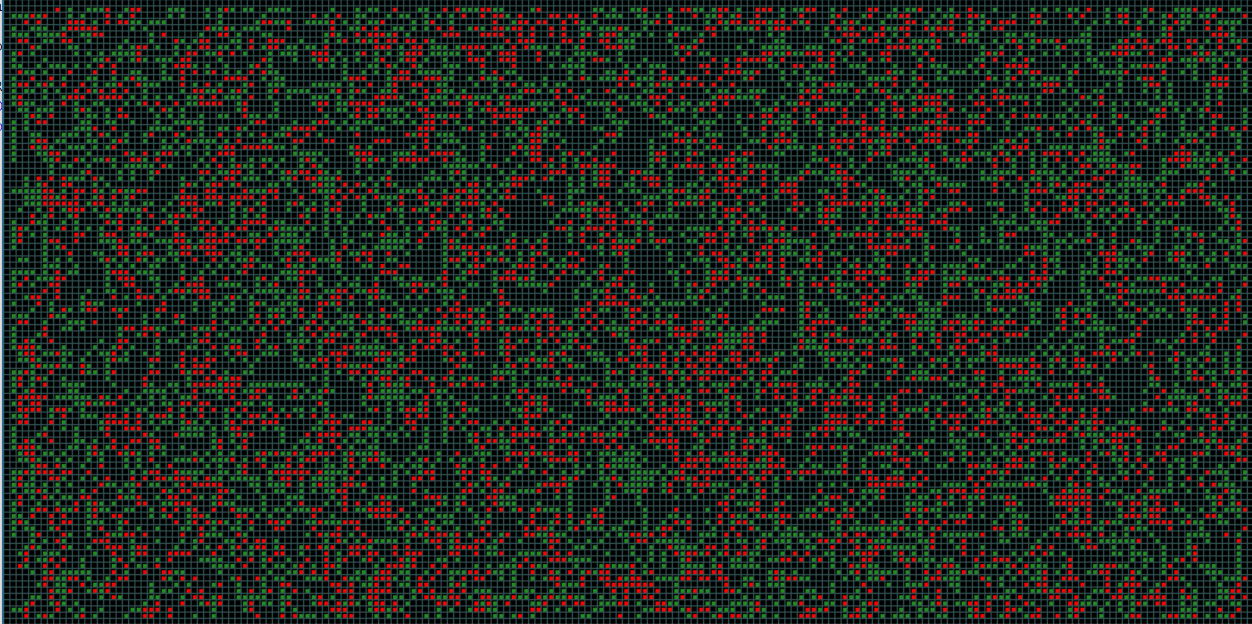


Рисунок 2.16 – Вариант 3, поколение 15

Здесь можно увидеть значительное сокращение времени, при котором можно получить случайную картину. Уже к 5 поколению большая часть полотна клеточного автомата заполнена случайно расположенными значениями, а к 15 поколению и вовсе все полотно. Однако можно еще немного ускорить данный процесс.

1. «Зомби» клетки на четных позициях

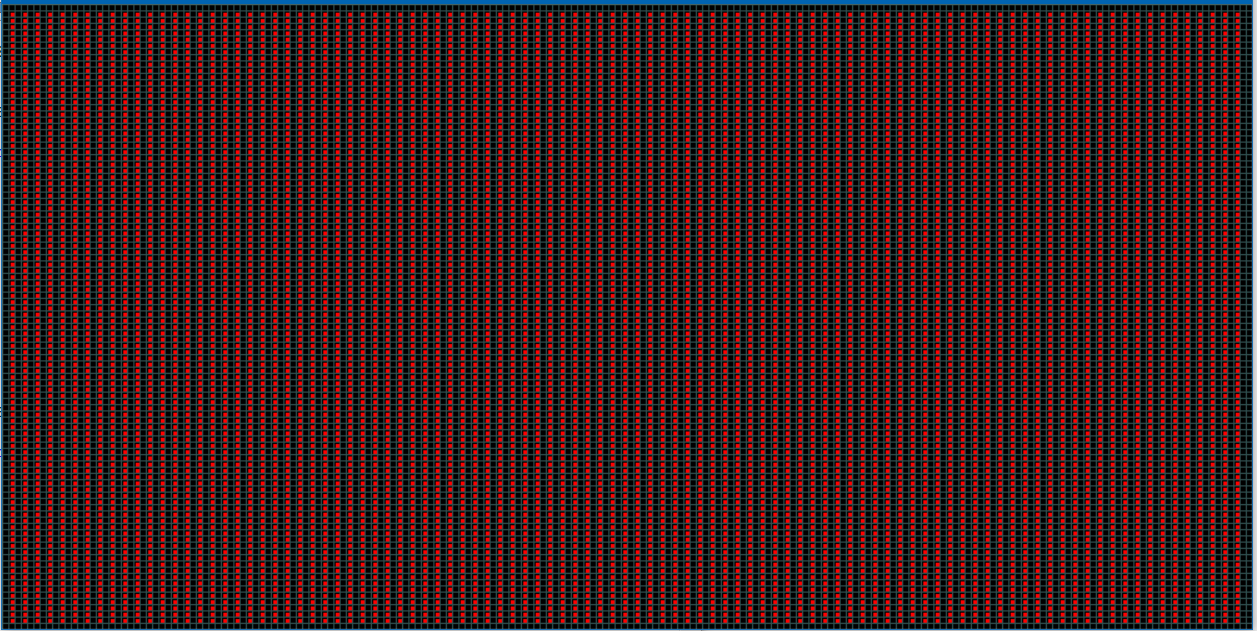


Рисунок 2.17 – Вариант 4, поколение 0

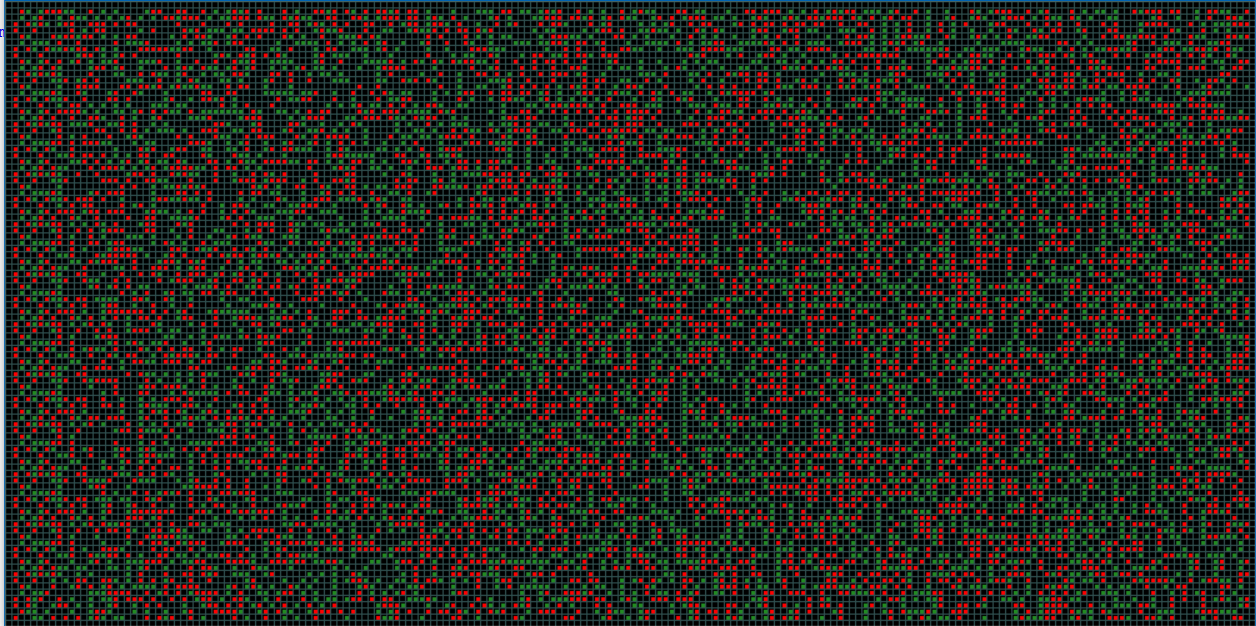


Рисунок 2.18 – Вариант 4, поколение 3

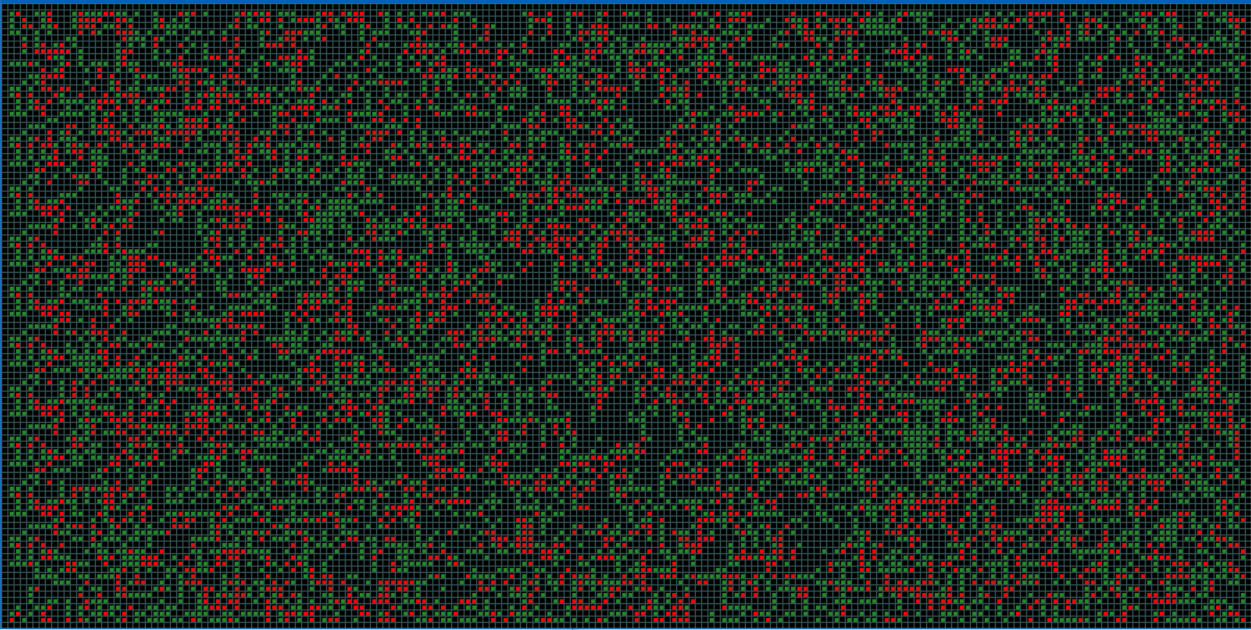


Рисунок 2.16 – Вариант 3, поколение 5

Заменив, «живые» клетки «мертвыми», удалось еще больше ускорить время, за которое достигается видимая случайность. Уже к 3 поколению получается достаточно хороший с визуальной точки результат.

Данную конфигурацию начального состояния клеточного автомата и будем рассматривать в качестве исследуемой.

## **Выбор продолжительности «Жизни»**

Исходя из предыдущего пункта, можно понять, что при разных конфигурациях, а именно при разных размерах клеточного автомата и разных начальных состояниях клеточного автомата, требуется разная продолжительность «жизни». Поэтому при выборе стоит опираться на то, как быстро может быть достигнута хотя бы видимая случайность. Так как был выбран 4 вариант конфигурации, то в дальнейшем в исследовании будет рассматриваться результат после 5 поколений.

## **Вывод**

В разделе проведена формализация задачи. Были представлены диаграммы, уточняющие суть представляемого метода. Были приведены основные элементы, лежащие в основе предлагаемого метода генерации псевдослучайных чисел, их модификация. Дано сравнение модифицированной и классической игр «Жизнь» и объяснение причин модификации. Также приведено описание проблем, которые возникли и возникают появляются при работе с данным генераторами псевдослучайных чисел, как например выбор начального заполнения клеточного автомата, выбор числа поколений, предложены способы их решения.

# Технологический раздел

В технологическом разделе описан выбор способа реализации метода генерации псевдослучайных чисел на основе клеточного автомата, а также выбор средств реализации. Описаны форматы входных и выходных данных, используемые библиотеки и реализация подсистем. Приведены примеры реализации структур данных и описаны правила описания модулей для загрузки в систему. Описан предоставляемый функционал и интерфейс. Описаны пути масштабирования – добавление в систему модулей и обработчиков сообщений для модулей.



## **Обоснование выбора языка, среды программирования**

Для реализации метода генерации псевдослучайных чисел на основе клеточного автомата «Жизнь» был выбран язык программирования Python 3.8, так как для него существует огромное количество различных библиотек, от математических до графических, облегчающих работу программиста и расширяющих его возможности. Данный язык имеет ряд преимуществ, которые и повлияли на выбор:

* Широкое распространение. Python используется во многих областях, включая научные и инженерные вычисления, веб-приложения и анализ данных.
* Простота и удобство в использовании. Python предлагает чистый и понятный синтаксис, что делает код легким для чтения и изменения.
* Большое количество библиотек. Python имеет широкий выбор библиотек для научных вычислений, включая numpy и scipy, которые предоставляют инструменты для работы с матрицами, числами, массивами и другими типами данных.
* Открытый исходный код. Python – это программное обеспечение с открытым исходным кодом, что означает, что сообщество разработчиков может вносить свой вклад в улучшение и расширение языка.
* Динамическая типизация данных ускоряет разработку.
* Python содержит встроенный сборщик мусора, благодаря чему отсутствует утечка памяти. Также он используется в качестве источника энтропии.

Python является интерпретируемым языком, поэтому ПО запустится везде, где есть этот интерпретатор и необходимые библиотеки. В качестве среды разработки в данной работе использовалась стандартная IDLE Python. Эта среда предоставляется вместе с интерпретатором и в этой связи использование указанного инструмента является достаточным для эффективной разработки систем с невысоким уровнем сложности, при этом обеспечивая возможность независимого отслеживания разработчиком.

## **Описание используемых модулей**

### Используемые библиотеки

Как уже говорилось, язык программирования Python богат на различные библиотеки, расширяющие возможности программиста. Так и в реализации предложенного генератора псевдослучайных чисел используется ряд библиотек, описание которых и будет приведено в этой главе.

* tkinter - стандартный интерфейс Python для инструментария Tk GUI. Как Tk, так и tkinter доступны на большинстве платформ Unix, а также в системах Windows. Tkinter обеспечивает создание и работу графического интерфейса.
* os – модуль, обеспечивающий портативный способ использования функций, зависящих от операционной системы. Используется для открытия файла, содержащего результат работы генератора
* random – реализует генераторы псевдослучайных чисел для различных дистрибутивов. Данная библиотека не используется для в работе предложенного метода генерации случайных чисел, однако она используется в исследовании, в частности используется подкласс randint, который выдает псевдослучайное число в заданном диапазоне.
* pygame – набор модулей (библиотек), предназначенный для написания компьютерных игр и мультимедиа-приложений. Pygame базируется на мультимедийной библиотеке SDL. Имеет большие графические возможности и используется для реализации графического представления работы генератора псевдослучайных чисел
* math – стандартный модуль, обеспечивающий доступ к математическим функциям, определенным стандартом C. Не поддерживает комплексные числа.

### Используемые модули

В этой реализации программного обеспечения имеется два модуля с исходным кодом: один обеспечивает интерфейс пользователя, в то время как другой реализует работу генератора псевдослучайных чисел. Поэтому далее мы будем сосредотачиваться на функциях и блоках кода, отвечающих за работу генератора псевдослучайных чисел.

Для работы метода генерации псевдослучайных чисел в разработанном ПО реализованы следующие модули:

* Заполнение карты клеточного автомата;
* Воспроизведение игры «Жизнь»;
* Отбор строки;
* Вывод результатов.

### Реализация модулей.

На листингах 3.1 – 3.3 представлена программная реализация клеточного автомата «Жизнь» с введенными модификациями и новыми правилами игры.

#### **Заполнение карты клеточного автомата**

Листинг 3.1 – Обход клеток мира клеточного автомата.

|  |
| --- |
| def life(W, H, next\_field, current\_field, tries):  for k in range(tries):  for x in range(1, W - 1):  for y in range(1, H - 1):  next\_field[y][x] = check\_cell(current\_field, x, y,W,H)    current\_field = [\*next\_field]    return current\_field |

Листинг 3.2 – Заполнение карты клеточного автомата. Проверка каждой ячейки согласно модифицированным правилам игры «Жизнь»

|  |
| --- |
| def fill (W, H):  current\_field = next\_field = [[0 for i in range(W)] for j in range(H)]  entrp = 0  while entrp == 0:  entrp = round((time.process\_time\_ns() \* timeit.timeit()))#% (W\*H)/2  entrp = entrp % 10000  print(entrp)  current\_field = [[2 if i == W // entrp or j == H // entrp else 0 for i in range(W)] for j in range(H)]  for j in range (W):  for i in range(H):  if i != 0 and j!= 0:  if entrp % i == 0 or entrp % j== 0:  current\_field[i][j] = i % 3  elif i > j and not (2 \* i + j) % 4:  current\_field[i][j] = 2  else:  current\_field[i][j-i] = 1  return next\_field, current\_field |

#### **Воспроизведение игры «Жизнь»**

Листинг 3.3 – Проверка каждой ячейки согласно модифицированным правилам игры «Жизнь»

|  |
| --- |
| def check\_cell(current\_field, x, y,W, H):  count = 0    for j in range(y - 1, y + 2):  for i in range(x - 1, x + 2):  if current\_field[j % H][i % W] != 0:  count += 1  # Zombie  if current\_field[y][x] == 2:  count -= 1  if count == 2 or count == 4:  return 2  return 0  else:  if count == 6:  return 2  # Alive  if current\_field[y][x] == 0:  count -= 1  if count == 2 or count == 3:  return 1  return 0  else:  if count == 3:  return 1  return 0 |

Данные модули представлены на листингах приложения А.

#### **Отбор строки**

Ввиду того, что клеточный автомат представлен в виде матрицы, то в ходе работы реализованного программного обеспечения матрица заполняется цифрами 0, 1 и 2.

В зависимости от требований к применяемым задачам, а также в связи с тем, что клеточный автомат можно масштабировать и рассматривать разными способами, как например собирать все строки в одну или работать со столбцами. Кроме того, мир игры «Жизнь» можно представить и как тор, и как шар. Последнее и реализовано в разработанном программном обеспечении.

Результатом работы ГПСЧ является двоичная последовательность, сформированная из отдельно взятой строки матрицы. Выбор данной строки основан на сравнении сериальной корреляции каждой из строк матрицы «Жизни», после чего выбирается та, у которой значение наиболее приближенное к нулю.

Так как критерий сериальной корреляции работает с бинарными значениями, а каждая из строк матрицы представлена в виде последовательности тритов, то для разрешения этой проблемы используется функция конвертации последовательности тритов в последовательность битов.

Листинг 3.4 – Модуль выбора строки

|  |
| --- |
| def choise\_line(current\_field):  summ = 0  summax = 0  masmax = []  cmin = 1  x = []  masK = []  masnum = []  KK = 0  for k in range(1,len(current\_field)-1):  x = current\_field[k]  masK.append(k)  num = ''  num3 = ''  for j in x:  num += str(j)  num3 += str(j)    if sum(x) == 0:  continue  num = convert\_base(num, 2, 3)  num = list(num)    c = Knut\_test(num, 0)  if c == '!!!':  continue  if abs(c) < cmin:  masmax = num  masnum = num3  cmin = c  KK = k  return KK, masmax, masnum |

Листинг 3.5 – Конвертер систем счислений

|  |
| --- |
| def convert\_base(num, to\_base, from\_base):  # first convert to decimal number  n = int(num, from\_base) if isinstance(num, str) else num  # now convert decimal to 'to\_base' base  alphabet = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"  res = ""  while n > 0:  n,m = divmod(n, to\_base)  res += alphabet[m]  return res[::-1] |

Листинг 3.6 – Тест сериальной корреляции

|  |
| --- |
| def Knut\_test(U, flag):  n = len(U)  u2 = 0  u\_sum2 = 0  # print(U)  if isinstance(U[0], str):  for i in range(len(U)):  U[i] = int(U[i])  u\_sum2 = sum(U)\*\*2  U1U2 = 0  for i in range(n-1):  U1U2 += U[i]\*U[i+1]  u2 += U[i]\*\*2  C = (n\*U1U2 - u\_sum2)/(n\*u2 - u\_sum2)  un = (-1)/(n-1)  sigma = sqrt((n\*\*2)/(((n-1)\*\*2)\*(n-2)))  if flag == 1:  if un - 2\*sigma < C < un + 2\*sigma:  print('Хорошее значение - ', C)    else:  print('Нехорошее значение - ', C)    return C |

Данные модули представлены на листингах приложения Б.

#### **Вывод результатов**

Выходными данными данной реализации метода генерации псевдослучайных чисел является документ формата .txt, в котором находится бинарная последовательность псевдослучайных битов.

Листинг 3.6 – Тест сериальной корреляции

|  |
| --- |
| def print\_field(field, W, H, pref):  if H!= 0:  f = open('nums'+pref+'.txt','w')  for i in range(len(field)):  buf = ''  for j in range(len(field[i])):  buf += str(field[i][j])  f.write(buf+'\n')  f.close()  else:  f = open('nums'+pref+'.txt','w')  buf = ''  for i in range(W):  buf += str(field[i])+''  f.write(buf)  f.close() |

Данные модули представлены на листингах приложения Б.

## **Реализация критериев для исследования**

Подробно о критериях для исследования качественных характеристик будет сказано в исследовательском разделе. Здесь же листинг модулей, связующих реализации критериев для тестов NIST и DIEHARD.

Листинг 3.7 – модуль sp800\_22\_tests для пакета тестов NIST.

|  |
| --- |
| from \_\_future\_\_ import print\_function  import argparse  import sys  import pickle  def read\_bits\_from\_file(filename,bigendian):  bitlist = list()  if filename == None:  f = sys.stdin  else:  f = open(filename, "rb")  while True:  bytes = f.read(16384)  if bytes:  for bytech in bytes:  if sys.version\_info > (3,0):  byte = bytech  else:  byte = ord(bytech)  for i in range(8):  if bigendian:  bit = (byte & 0x80) >> 7  byte = byte << 1  else:  bit = (byte >> i) & 1  bitlist.append(bit)  else:  break  f.close()  return bitlist  import argparse  import sys  parser = argparse.ArgumentParser(description='Test data for distinguishability form random, using NIST SP800-22Rev1a algorithms.')  parser.add\_argument('filename', type=str, nargs='?', help='Filename of binary file to test')  parser.add\_argument('--be', action='store\_false',help='Treat data as big endian bits within bytes. Defaults to little endian')  parser.add\_argument('-t', '--testname', default=None,help='Select the test to run. Defaults to running all tests. Use --list\_tests to see the list')  parser.add\_argument('--list\_tests', action='store\_true',help='Display the list of tests')  args = parser.parse\_args()  bigendian = args.be  filename = args.filename  testlist = [  'monobit\_test',  'frequency\_within\_block\_test',  'runs\_test',  'longest\_run\_ones\_in\_a\_block\_test',  'binary\_matrix\_rank\_test',  'dft\_test',  'non\_overlapping\_template\_matching\_test',  'overlapping\_template\_matching\_test',  'maurers\_universal\_test',  'linear\_complexity\_test',  'serial\_test',  'approximate\_entropy\_test',  'cumulative\_sums\_test',  'random\_excursion\_test',  'random\_excursion\_variant\_test']    print("Tests of Distinguishability from Random")  if args.list\_tests:  for i,testname in zip(range(len(testlist)),testlist):  print(str(i+1).ljust(4)+": "+testname)  exit()  f = open('../nums.txt','r')  bits = ''  for i in f:  bits += i  bits = list(bits)  for i in range(len(bits)):  bits[i] = int(bits[i])  gotresult=False  if args.testname:  if args.testname in testlist:  m = \_\_import\_\_ ("sp800\_22\_"+args.testname)  func = getattr(m,args.testname)  print("TEST: %s" % args.testname)  success,p,plist = func(bits)  gotresult = True  if success:  print("PASS")  else:  print("FAIL")    if p:  print("P="+str(p))  if plist:  for pval in plist:  print("P="+str(pval))  else:  print("Test name (%s) not known" % args.ttestname)  exit()  else:  results = list()    for testname in testlist:  print("TEST: %s" % testname)  m = \_\_import\_\_ ("sp800\_22\_"+testname)  func = getattr(m,testname)    (success,p,plist) = func(bits)  summary\_name = testname  if success:  print(" PASS")  summary\_result = "PASS"  else:  print(" FAIL")  summary\_result = "FAIL"    if p != None:  print(" P="+str(p))  summary\_p = str(p)    if plist != None:  for pval in plist:  print("P="+str(pval))  summary\_p = str(min(plist))    results.append((summary\_name,summary\_p, summary\_result))    print()  print("SUMMARY")  print("-------")    for result in results:  (summary\_name,summary\_p, summary\_result) = result  print(summary\_name.ljust(40),summary\_p.ljust(18),summary\_result) |

Данный модуль представлены на листингах приложения В.

## **Вывод**

В разделе были приведены элементы конкретной реализации генератора псевдослучайных чисел. Было приведено описание работы функций и логических единиц приведенной реализации. Также приведено техническое описание разработанных подсистем, причины их использования, предоставлены примеры из исходного кода разработанного программного обеспечения.

# Исследовательский раздел

В исследовательском разделе приведено исследование предложенного метода генерации псевдослучайных чисел. Представлены результаты исследования качества работы генератора согласно ряду выбранных критериев качества. Также представлен сравнительный анализ предложенного метода генерации с линейным конгруэнтным методом и методом, основанном на клеточном автомате.

В текущем разделе также приведено описание проведенных экспериментов, тестовых результатов и пояснения к ним.



## **Описание среды и устройства для исследования**

В данном разделе рассмотрены характеристики устройства и характеристики среды, в которых будет проходить дальнейшее исследование. Стоит отметить, что в действительности, ввиду просто ты реализации предложенного метода генерации псевдослучайных чисел, само программное обеспечение генератора не требует каких-либо серьезных характеристик запускающего его ЭВМ.

Характеристики устройства:

* операционная система Windows 10 Pro (сборка 19042.985);
* Процессор Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1801 МГц, ядер: 4, логических процессоров: 8;
* ОЗУ 8Gb DDR3.

Характеристики среды:

* интерпретатор Python 3.8.2 x64;

Библиотеки, необходимые для запуска:

* pygame;

## **Пример работы**

По своей сути, генератору псевдослучайных чисел не требуется графический интерфейс. Однако в рамках демонстрации работы предложенного метода, был создан простой графический интерфейс, предоставляющий возможность ввода входных параметров, описанных в IDEF0 диаграмме конструкторского раздела.

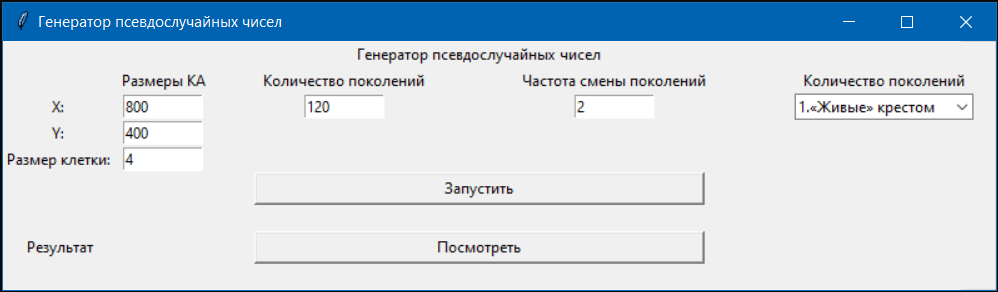


Рисунок 4.1 – Начальное окно демонстрационного приложения

Помимо интерфейса также в качестве демонстрации работы модифицированной игры «Жизнь» создано графическое представление данного клеточного автомата. Данное представление изображено на рисунке 4.2. Здесь можно видеть сетчатое поле, заполненное клетками 3 цветов: зеленые клетки отвечают за «живых», черные за «мертвых» и красные за «зомби».

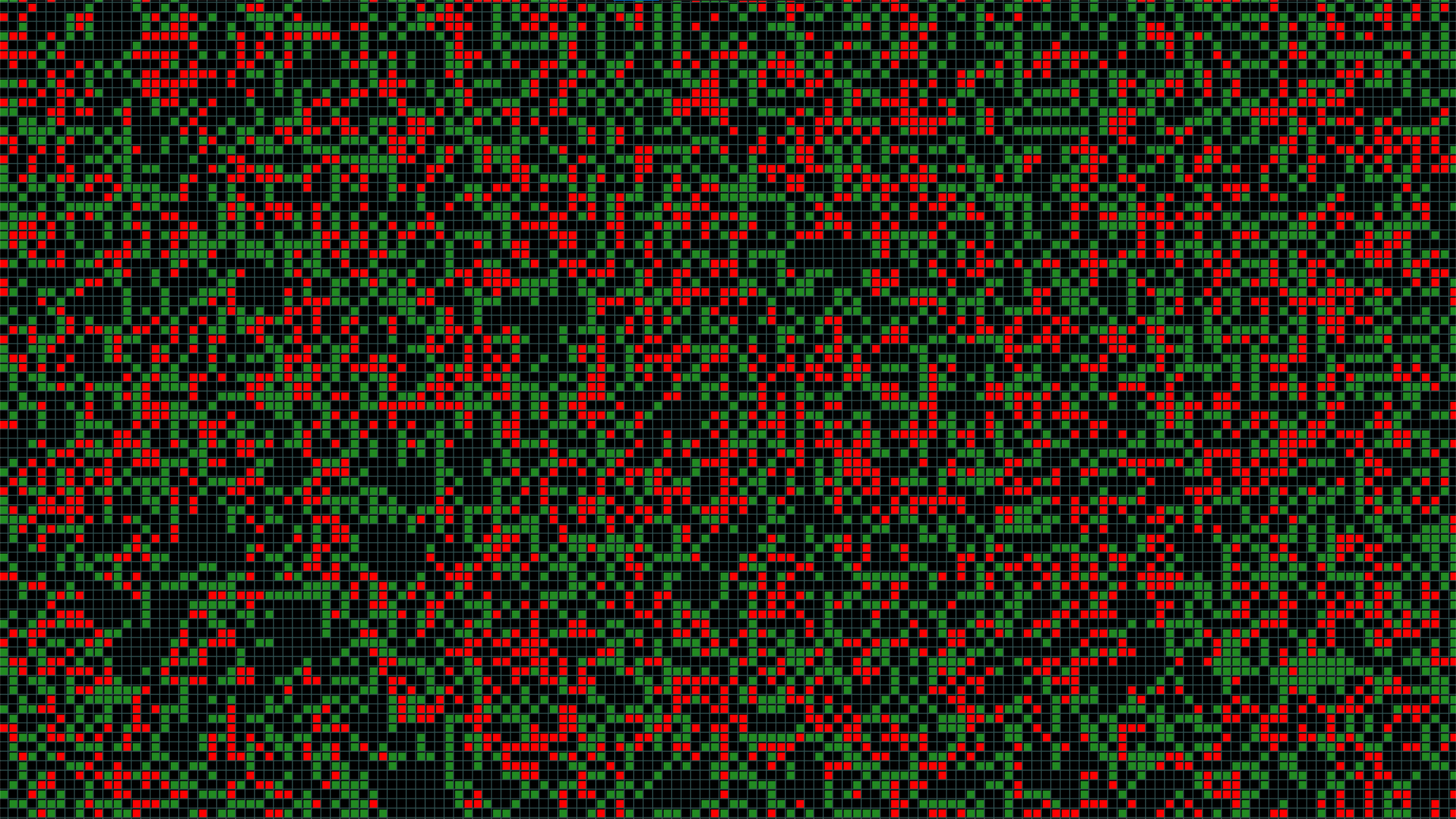


Рисунок 4.2 – Модифицированная игра «Жизнь»

Результат работы программного обеспечения, а именно значения одной из строк клеточного автомата, записывается в текстовый файл с именем nums.txt. Для его просмотра предусмотрена кнопка «посмотреть» в начальном окне приложения.

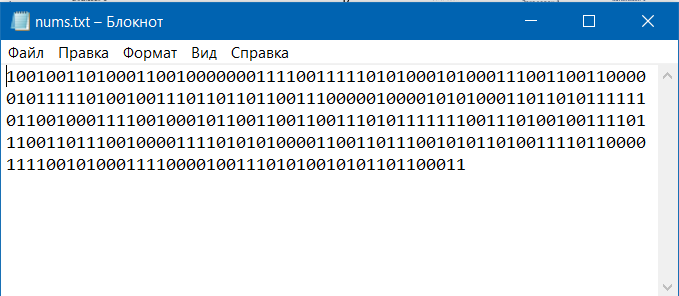


Рисунок 4.3 – Результат работы клеточного автомата

Как видно из рисунков 4.3, результат представлен в виде бинарной последовательности, а не тернарной, как должно быть исходя из особенностей спроектированного клеточного автомата. Это сделано с целью дальнейшего исследования на оценку качества. О критериях качества будет рассказано в следующих подразделах.

## **Постановка экспериментов**

Для оценки качества разработанного метода генерации псевдослучайных чисел был выбран ряд критериев, входящих в набор NIST-тестов. Тесты NIST предназначены для оценки случайности последовательностей, тем самым позволяют оценить, насколько хорошо или насколько плохо работает тот или иной генератор случайных чисел или псевдослучайных чисел и работает ли он вообще.

### Критерии оценки качества

В данном подразделе дается описание критериев оценки качества, а также их реализации. Каждый из данных критериев отдельно дает оценку случайности последовательности значений. Суммарно же, успешное прохождение всех и каждого из приведенных критериев дает право говорить о том, имеет ли предложенный метод генерации псевдослучайных чисел право на существование.

#### **Частотный побитовый тест**

Целью теста является выявление несбалансированности количества нулей и единиц в бинарной последовательности. Тест считается проваленным, если нулей или единиц слишком много.

Рекомендуемыми параметрами для тестирования являются бинарная последовательность из 1.000.000 бит и 100 последовательностей для принятия гипотез.

Листинг 4.1 – модуль monobit\_test

*from \_\_future\_\_ import print\_function*

*import math*

*def count\_ones\_zeroes(bits):*

*ones = 0*

*zeroes = 0*

*for bit in bits:*

*if (bit == 1):*

*ones += 1*

*else:*

*zeroes += 1*

*return (zeroes,ones)*

*def monobit\_test(bits):*

*n = len(bits)*

*zeroes,ones = count\_ones\_zeroes(bits)*

*s = abs(ones-zeroes)*

*print(" Ones count = %d" % ones)*

*print(" Zeroes count = %d" % zeroes)*

*p = math.erfc(float(s)/(math.sqrt(float(n)) \* math.sqrt(2.0)))*

*success = (p >= 0.01)*

*return (success,p,None)*

#### **Частотный блочный тест**

Данный тест схож с предыдущим. Последовательность делится на M блоков, в каждом из блоков считается частота появления единиц и насколько эта частота близка к эталонному значению M/2. При M=1 длина блока составляет всего 1 бит что превращает данный тест в предыдущий.

Рекомендуемыми параметрами для тестирования являются последовательность не менее 100 бит и блок равный 20 бит.

Листинг 4.2 – модуль frequency\_within\_block\_test

*from \_\_future\_\_ import print\_function*

*import math*

*from fractions import Fraction*

*#from scipy.special import gamma, gammainc, gammaincc*

*from gamma\_functions import \**

*#ones\_table = [bin(i)[2:].count('1') for i in range(256)]*

*def count\_ones\_zeroes(bits):*

*ones = 0*

*zeroes = 0*

*for bit in bits:*

*if (bit == 1):*

*ones += 1*

*else:*

*zeroes += 1*

*return (zeroes,ones)*

*def frequency\_within\_block\_test(bits):*

*# Compute number of blocks M = block size. N=num of blocks*

*# N = floor(n/M)*

*# miniumum block size 20 bits, most blocks 100*

*n = len(bits)*

*M = 20*

*N = int(math.floor(n/M))*

*if N > 99:*

*N=99*

*M = int(math.floor(n/N))*

*if len(bits) < 100:*

*print("Too little data for test. Supply at least 100 bits")*

*return False,1.0,None*

*print(" n = %d" % len(bits))*

*print(" N = %d" % N)*

*print(" M = %d" % M)*

*num\_of\_blocks = N*

*block\_size = M #int(math.floor(len(bits)/num\_of\_blocks))*

*#n = int(block\_size \* num\_of\_blocks)*

*proportions = list()*

*for i in range(num\_of\_blocks):*

*block = bits[i\*(block\_size):((i+1)\*(block\_size))]*

*zeroes,ones = count\_ones\_zeroes(block)*

*proportions.append(Fraction(ones,block\_size))*

*chisq = 0.0*

*for prop in proportions:*

*chisq += 4.0\*block\_size\*((prop - Fraction(1,2))\*\*2)*

*p = gammaincc((num\_of\_blocks/2.0),float(chisq)/2.0)*

*success = (p >= 0.01)*

*return (success,p,None)*

#### **Частотный блочный тест**

В тесте находятся все серии битов – непрерывные последовательности одинаковых битов – и их распределение сравнивается с ожидаемым распределением таких серий для случайной последовательности. Длина последовательности 100 и более бит. [4]

Листинг 4.3 – модуль runs\_test

*from \_\_future\_\_ import print\_function*

*import math*

*from fractions import Fraction*

*#from scipy.special import gamma, gammainc, gammaincc*

*from gamma\_functions import \**

*import numpy*

*import cmath*

*import random*

*#ones\_table = [bin(i)[2:].count('1') for i in range(256)]*

*def count\_ones\_zeroes(bits):*

*ones = 0*

*zeroes = 0*

*for bit in bits:*

*if (bit == 1):*

*ones += 1*

*else:*

*zeroes += 1*

*return (zeroes,ones)*

*def runs\_test(bits):*

*n = len(bits)*

*zeroes,ones = count\_ones\_zeroes(bits)*

*prop = float(ones)/float(n)*

*print(" prop ",prop)*

*tau = 2.0/math.sqrt(n)*

*print(" tau ",tau)*

*if abs(prop-0.5) > tau:*

*return (False,0.0,None)*

*vobs = 1.0*

*for i in range(n-1):*

*if bits[i] != bits[i+1]:*

*vobs += 1.0*

*print(" vobs ",vobs)*

*p = math.erfc(abs(vobs - (2.0\*n\*prop\*(1.0-prop)))/(2.0\*math.sqrt(2.0\*n)\*prop\*(1-prop) ))*

*success = (p >= 0.01)*

*return (success,p,None)*

#### **Спектральный тест**

Целью теста является обнаружение повторяющегося блока или последовательности. Для этого последовательность преобразуется по формуле преобразования Фурье, после чего оцениваются высоты пиков дискретного преобразования Фурье исходной. [4]

Листинг 4.4 – модуль dft\_test

*from \_\_future\_\_ import print\_function*

*import math*

*import numpy*

*import sys*

*def dft\_test(bits):*

*n = len(bits)*

*if (n % 2) == 1: # Make it an even number*

*bits = bits[:-1]*

*ts = list() # Convert to +1,-1*

*for bit in bits:*

*ts.append((bit\*2)-1)*

*ts\_np = numpy.array(ts)*

*fs = numpy.fft.fft(ts\_np) # Compute DFT*

*if sys.version\_info > (3,0):*

*mags = abs(fs)[:n//2] # Compute magnitudes of first half of sequence*

*else:*

*mags = abs(fs)[:n/2] # Compute magnitudes of first half of sequence*

*T = math.sqrt(math.log(1.0/0.05)\*n) # Compute upper threshold*

*N0 = 0.95\*n/2.0*

*print(" N0 = %f" % N0)*

*N1 = 0.0 # Count the peaks above the upper theshold*

*for mag in mags:*

*if mag < T:*

*N1 += 1.0*

*print(" N1 = %f" % N1)*

*d = (N1 - N0)/math.sqrt((n\*0.95\*0.05)/4) # Compute the P value*

*p = math.erfc(abs(d)/math.sqrt(2))*

*success = (p >= 0.01)*

*return (success,p,None)*

#### **Тест с неперекрывающимися непериодическими шаблонами**

Показывает число заранее заданных битовых строк (шаблонов) в последовательности. Поиск шаблонов осуществляется методом бегущего окна, с подсчетом найденных совпадений. В этом тесте, при найденном совпадении окно. [4]

Листинг 4.5 – модуль non\_overlapping\_template\_matching\_test\_test

*from \_\_future\_\_ import print\_function*

*import math*

*#from scipy.special import gamma, gammainc, gammaincc*

*from gamma\_functions import \**

*import random*

*def non\_overlapping\_template\_matching\_test(bits):*

*# The templates provdided in SP800-22rev1a*

*templates = [None for x in range(7)]*

*templates[0] = [[0,1],[1,0]]*

*templates[1] = [[0,0,1],[0,1,1],[1,0,0],[1,1,0]]*

*templates[2] = [[0,0,0,1],[0,0,1,1],[0,1,1,1],[1,0,0,0],[1,1,0,0],[1,1,1,0]]*

*templates[3] = [[0,0,0,0,1],[0,0,0,1,1],[0,0,1,0,1],[0,1,0,1,1],[0,0,1,1,1],[0,1,1,1,1],*

*[1,1,1,0,0],[1,1,0,1,0],[1,0,1,0,0],[1,1,0,0,0],[1,0,0,0,0],[1,1,1,1,0]]*

*templates[4] = [[0,0,0,0,0,1],[0,0,0,0,1,1],[0,0,0,1,0,1],[0,0,0,1,1,1],[0,0,1,0,1,1],*

*[0,0,1,1,0,1],[0,0,1,1,1,1],[0,1,0,0,1,1],*

*[0,1,0,1,1,1],[0,1,1,1,1,1],[1,0,0,0,0,0],*

*[1,0,1,0,0,0],[1,0,1,1,0,0],[1,1,0,0,0,0],*

*[1,1,0,0,1,0],[1,1,0,1,0,0],[1,1,1,0,0,0],*

*[1,1,1,0,1,0],[1,1,1,1,0,0],[1,1,1,1,1,0]]*

*templates[5] = [[0,0,0,0,0,0,1],[0,0,0,0,0,1,1],[0,0,0,0,1,0,1],[0,0,0,0,1,1,1],*

*[0,0,0,1,0,0,1],[0,0,0,1,0,1,1],[0,0,0,1,1,0,1],[0,0,0,1,1,1,1],*

*[0,0,1,0,0,1,1],[0,0,1,0,1,0,1],[0,0,1,0,1,1,1],[0,0,1,1,0,1,1],*

*[0,0,1,1,1,0,1],[0,0,1,1,1,1,1],[0,1,0,0,0,1,1],[0,1,0,0,1,1,1],*

*[0,1,0,1,0,1,1],[0,1,0,1,1,1,1],[0,1,1,0,1,1,1],[0,1,1,1,1,1,1],*

*[1,0,0,0,0,0,0],[1,0,0,1,0,0,0],[1,0,1,0,0,0,0],[1,0,1,0,1,0,0],*

*[1,0,1,1,0,0,0],[1,0,1,1,1,0,0],[1,1,0,0,0,0,0],[1,1,0,0,0,1,0],*

*[1,1,0,0,1,0,0],[1,1,0,1,0,0,0],[1,1,0,1,0,1,0],[1,1,0,1,1,0,0],*

*[1,1,1,0,0,0,0],[1,1,1,0,0,1,0],[1,1,1,0,1,0,0],[1,1,1,0,1,1,0],*

*[1,1,1,1,0,0,0],[1,1,1,1,0,1,0],[1,1,1,1,1,0,0],[1,1,1,1,1,1,0]]*

*templates[6] = [[0,0,0,0,0,0,0,1],[0,0,0,0,0,0,1,1],[0,0,0,0,0,1,0,1],[0,0,0,0,0,1,1,1],*

*[0,0,0,0,1,0,0,1],[0,0,0,0,1,0,1,1],[0,0,0,0,1,1,0,1],[0,0,0,0,1,1,1,1],*

*[0,0,0,1,0,0,1,1],[0,0,0,1,0,1,0,1],[0,0,0,1,0,1,1,1],[0,0,0,1,1,0,0,1],*

*[0,0,0,1,1,0,1,1],[0,0,0,1,1,1,0,1],[0,0,0,1,1,1,1,1],[0,0,1,0,0,0,1,1],*

*[0,0,1,0,0,1,0,1],[0,0,1,0,0,1,1,1],[0,0,1,0,1,0,1,1],[0,0,1,0,1,1,0,1],*

*[0,0,1,0,1,1,1,1],[0,0,1,1,0,1,0,1],[0,0,1,1,0,1,1,1],[0,0,1,1,1,0,1,1],*

*[0,0,1,1,1,1,0,1],[0,0,1,1,1,1,1,1],[0,1,0,0,0,0,1,1],[0,1,0,0,0,1,1,1],*

*[0,1,0,0,1,0,1,1],[0,1,0,0,1,1,1,1],[0,1,0,1,0,0,1,1],[0,1,0,1,0,1,1,1],*

*[0,1,0,1,1,0,1,1],[0,1,0,1,1,1,1,1],[0,1,1,0,0,1,1,1],[0,1,1,0,1,1,1,1],*

*[0,1,1,1,1,1,1,1],[1,0,0,0,0,0,0,0],[1,0,0,1,0,0,0,0],[1,0,0,1,1,0,0,0],*

*[1,0,1,0,0,0,0,0],[1,0,1,0,0,1,0,0],[1,0,1,0,1,0,0,0],[1,0,1,0,1,1,0,0],*

*[1,0,1,1,0,0,0,0],[1,0,1,1,0,1,0,0],[1,0,1,1,1,0,0,0],[1,0,1,1,1,1,0,0],*

*[1,1,0,0,0,0,0,0],[1,1,0,0,0,0,1,0],[1,1,0,0,0,1,0,0],[1,1,0,0,1,0,0,0],*

*[1,1,0,0,1,0,1,0],[1,1,0,1,0,0,0,0],[1,1,0,1,0,0,1,0],[1,1,0,1,0,1,0,0],*

*[1,1,0,1,1,0,0,0],[1,1,0,1,1,0,1,0],[1,1,0,1,1,1,0,0],[1,1,1,0,0,0,0,0],*

*[1,1,1,0,0,0,1,0],[1,1,1,0,0,1,0,0],[1,1,1,0,0,1,1,0],[1,1,1,0,1,0,0,0],*

*[1,1,1,0,1,0,1,0],[1,1,1,0,1,1,0,0],[1,1,1,1,0,0,0,0],[1,1,1,1,0,0,1,0],*

*[1,1,1,1,0,1,0,0],[1,1,1,1,0,1,1,0],[1,1,1,1,1,0,0,0],[1,1,1,1,1,0,1,0],*

*[1,1,1,1,1,1,0,0],[1,1,1,1,1,1,1,0]]*

*n = len(bits)*

*# Choose the template B*

*r = random.SystemRandom()*

*template\_list = r.choice(templates)*

*B = r.choice(template\_list)*

*m = len(B)*

*N = 8*

*M = int(math.floor(len(bits)/8))*

*n = M\*N*

*blocks = list() # Split into N blocks of M bits*

*for i in range(N):*

*blocks.append(bits[i\*M:(i+1)\*M])*

*W=list() # Count the number of matches of the template in each block Wj*

*for block in blocks:*

*position = 0*

*count = 0*

*while position < (M-m):*

*if block[position:position+m] == B:*

*position += m*

*count += 1*

*else:*

*position += 1*

*W.append(count)*

*mu = float(M-m+1)/float(2\*\*m) # Compute mu and sigma*

*sigma = M \* ((1.0/float(2\*\*m))-(float((2\*m)-1)/float(2\*\*(2\*m))))*

*chisq = 0.0 # Compute Chi-Square*

*for j in range(N):*

*chisq += ((W[j] - mu)\*\*2)/(sigma\*\*2)*

*p = gammaincc(N/2.0, chisq/2.0) # Compute P value*

*success = ( p >= 0.01)*

*return (success,p,None)*

#### **Тест на периодичность**

Определяет, действительно ли количество появлений перекрывающихся шаблонов длиной m бит приблизительно такое же, как в случае абсолютно случайной входной последовательности бит. Тест заключается в подсчете частоты всех возможных перекрываний шаблонов длины m бит на протяжении исходной последовательности битов. Предполагается, что в абсолютно случайной последовательности каждый шаблон длиной m бит появляется в последовательности с одинаковой вероятностью. [4]

Листинг 4.6 – модуль serial\_test

*from \_\_future\_\_ import print\_function*

*import math*

*#from scipy.special import gamma, gammainc, gammaincc*

*from gamma\_functions import \**

*def int2patt(n,m):*

*pattern = list()*

*for i in range(m):*

*pattern.append((n >> i) & 1)*

*return pattern*

*def countpattern(patt,bits,n):*

*thecount = 0*

*for i in range(n):*

*match = True*

*for j in range(len(patt)):*

*if patt[j] != bits[i+j]:*

*match = False*

*if match:*

*thecount += 1*

*return thecount*

*def psi\_sq\_mv1(m, n, padded\_bits):*

*counts = [0 for i in range(2\*\*m)]*

*for i in range(2\*\*m):*

*pattern = int2patt(i,m)*

*count = countpattern(pattern,padded\_bits,n)*

*counts.append(count)*

*psi\_sq\_m = 0.0*

*for count in counts:*

*psi\_sq\_m += (count\*\*2)*

*psi\_sq\_m = psi\_sq\_m \* (2\*\*m)/n*

*psi\_sq\_m -= n*

*return psi\_sq\_m*

*def serial\_test(bits,patternlen=None):*

*n = len(bits)*

*if patternlen != None:*

*m = patternlen*

*else:*

*m = int(math.floor(math.log(n,2)))-2*

*if m < 4:*

*print("Error. Not enough data for m to be 4")*

*return False,0,None*

*m = 4*

*# Step 1*

*padded\_bits=bits+bits[0:m-1]*

*# Step 2*

*psi\_sq\_m = psi\_sq\_mv1(m, n, padded\_bits)*

*psi\_sq\_mm1 = psi\_sq\_mv1(m-1, n, padded\_bits)*

*psi\_sq\_mm2 = psi\_sq\_mv1(m-2, n, padded\_bits)*

*delta1 = psi\_sq\_m - psi\_sq\_mm1*

*delta2 = psi\_sq\_m - (2\*psi\_sq\_mm1) + psi\_sq\_mm2*

*P1 = gammaincc(2\*\*(m-2),delta1/2.0)*

*P2 = gammaincc(2\*\*(m-3),delta2/2.0)*

*print(" psi\_sq\_m = ",psi\_sq\_m)*

*print(" psi\_sq\_mm1 = ",psi\_sq\_mm1)*

*print(" psi\_sq\_mm2 = ",psi\_sq\_mm2)*

*print(" delta1 = ",delta1)*

*print(" delta2 = ",delta2)*

*print(" P1 = ",P1)*

*print(" P2 = ",P2)*

*success = (P1 >= 0.01) and (P2 >= 0.01)*

*return (success, None, [P1,P2])*

*if \_\_name\_\_ == "\_\_main\_\_":*

*bits = [0,0,1,1,0,1,1,1,0,1]*

*success, \_, plist = serial\_test(bits, patternlen=3)*

*print("success =",success)*

*print("plist = ",plist)*

#### **Тест приблизительной энтропии**

Как и в тесте на периодичность в данном тесте акцент делается на подсчете частоты всех возможных перекрытий шаблонов длины m бит на протяжении исходной последовательности битов. Цель теста — сравнить частоты перекрывания двух последовательных блоков исходной последовательности с длинами m и m+1 с частотами перекрывания аналогичных блоков в абсолютно случайной последовательности. [4]

Листинг 4.7 – модуль approximate\_entropy\_test

*from \_\_future\_\_ import print\_function*

*import math*

*#from scipy.special import gamma, gammainc, gammaincc*

*from gamma\_functions import \**

*def bits\_to\_int(bits):*

*theint = 0*

*for i in range(len(bits)):*

*theint = (theint << 1) + bits[i]*

*return theint*

*def approximate\_entropy\_test(bits):*

*n = len(bits)*

*m = int(math.floor(math.log(n,2)))-6*

*if m < 2:*

*m = 2*

*if m >3 :*

*m = 3*

*print(" n = ",n)*

*print(" m = ",m)*

*Cmi = list()*

*phi\_m = list()*

*for iterm in range(m,m+2):*

*# Step 1*

*padded\_bits=bits+bits[0:iterm-1]*

*# Step 2*

*counts = list()*

*for i in range(2\*\*iterm):*

*#print " Pattern #%d of %d" % (i+1,2\*\*iterm)*

*count = 0*

*for j in range(n):*

*if bits\_to\_int(padded\_bits[j:j+iterm]) == i:*

*count += 1*

*counts.append(count)*

*print(" Pattern %d of %d, count = %d" % (i+1,2\*\*iterm, count))*

*# step 3*

*Ci = list()*

*for i in range(2\*\*iterm):*

*Ci.append(float(counts[i])/float(n))*

*Cmi.append(Ci)*

*# Step 4*

*sum = 0.0*

*for i in range(2\*\*iterm):*

*if (Ci[i] > 0.0):*

*sum += Ci[i]\*math.log((Ci[i]/10.0))*

*phi\_m.append(sum)*

*print(" phi(%d) = %f" % (m,sum))*

*# Step 5 - let the loop steps 1-4 complete*

*# Step 6*

*appen\_m = phi\_m[0] - phi\_m[1]*

*print(" AppEn(%d) = %f" % (m,appen\_m))*

*chisq = 2\*n\*(math.log(2) - appen\_m)*

*print(" ChiSquare = ",chisq)*

*# Step 7*

*p = gammaincc(2\*\*(m-1),(chisq/2.0))*

*success = (p >= 0.01)*

*return (success, p, None)*

*if \_\_name\_\_ == "\_\_main\_\_":*

*bits = [1,1,0,0,1,0,0,1,0,0,0,0,1,1,1,1,*

*1,1,0,1,1,0,1,0,1,0,1,0,0,0,1,0,*

*0,0,1,0,0,0,0,1,0,1,1,0,1,0,0,0,*

*1,1,0,0,0,0,1,0,0,0,1,1,0,1,0,0,*

*1,1,0,0,0,1,0,0,1,1,0,0,0,1,1,0,*

*0,1,1,0,0,0,1,0,1,0,0,0,1,0,1,1,*

*1,0,0,0]*

*success, p, \_ = approximate\_entropy\_test(bits)*

*print("success =",success)*

*print("p = ",p)*

#### **Тест кумулятивных сумм**

Тест заключается в максимальном отклонении (от нуля) при произвольном обходе, определяемым кумулятивной суммой заданных (-1, +1) цифр в последовательности. Цель данного теста — определить является ли кумулятивная сумма частичных последовательностей, возникающих во входной последовательности, слишком большой или слишком маленькой по сравнению с ожидаемым поведением такой суммы для абсолютно произвольной входной последовательности. Таким образом, кумулятивная сумма может рассматриваться как произвольный обход. Для случайной последовательности отклонение от произвольного обхода должно быть вблизи нуля. [4]

Листинг 4.8 – модуль cumulative\_sums\_test

*from \_\_future\_\_ import print\_function*

*import math*

*#from scipy.special import gamma, gammainc, gammaincc*

*from gamma\_functions import \**

*#import scipy.stats*

*def normcdf(n):*

*return 0.5 \* math.erfc(-n \* math.sqrt(0.5))*

*def p\_value(n,z):*

*sum\_a = 0.0*

*startk = int(math.floor((((float(-n)/z)+1.0)/4.0)))*

*endk = int(math.floor((((float(n)/z)-1.0)/4.0)))*

*for k in range(startk,endk+1):*

*c = (((4.0\*k)+1.0)\*z)/math.sqrt(n)*

*#d = scipy.stats.norm.cdf(c)*

*d = normcdf(c)*

*c = (((4.0\*k)-1.0)\*z)/math.sqrt(n)*

*#e = scipy.stats.norm.cdf(c)*

*e = normcdf(c)*

*sum\_a = sum\_a + d - e*

*sum\_b = 0.0*

*startk = int(math.floor((((float(-n)/z)-3.0)/4.0)))*

*endk = int(math.floor((((float(n)/z)-1.0)/4.0)))*

*for k in range(startk,endk+1):*

*c = (((4.0\*k)+3.0)\*z)/math.sqrt(n)*

*#d = scipy.stats.norm.cdf(c)*

*d = normcdf(c)*

*c = (((4.0\*k)+1.0)\*z)/math.sqrt(n)*

*#e = scipy.stats.norm.cdf(c)*

*e = normcdf(c)*

*sum\_b = sum\_b + d - e*

*p = 1.0 - sum\_a + sum\_b*

*return p*

*def cumulative\_sums\_test(bits):*

*n = len(bits)*

*# Step 1*

*x = list() # Convert to +1,-1*

*for bit in bits:*

*#if bit == 0:*

*x.append((bit\*2)-1)*

*# Steps 2 and 3 Combined*

*# Compute the partial sum and records the largest excursion.*

*pos = 0*

*forward\_max = 0*

*for e in x:*

*pos = pos+e*

*if abs(pos) > forward\_max:*

*forward\_max = abs(pos)*

*pos = 0*

*backward\_max = 0*

*for e in reversed(x):*

*pos = pos+e*

*if abs(pos) > backward\_max:*

*backward\_max = abs(pos)*

*# Step 4*

*p\_forward = p\_value(n, forward\_max)*

*p\_backward = p\_value(n,backward\_max)*

*success = ((p\_forward >= 0.01) and (p\_backward >= 0.01))*

*plist = [p\_forward, p\_backward]*

*if success:*

*print("PASS")*

*else:*

*print("FAIL: Data not random")*

*return (success, None, plist)*

*if \_\_name\_\_ == "\_\_main\_\_":*

*bits = [1,1,0,0,1,0,0,1,0,0,0,0,1,1,1,1,1,1,0,1,*

*1,0,1,0,1,0,1,0,0,0,1,0,0,0,1,0,0,0,0,1,*

*0,1,1,0,1,0,0,0,1,1,0,0,0,0,1,0,0,0,1,1,*

*0,1,0,0,1,1,0,0,0,1,0,0,1,1,0,0,0,1,1,0,*

*0,1,1,0,0,0,1,0,1,0,0,0,1,0,1,1,1,0,0,0]*

*success, \_, plist = cumulative\_sums\_test(bits)*

*print("success =",success)*

*print("plist = ",plist)*

#### **Тест на произвольные отклонения**

Суть данного теста заключается в подсчете числа циклов, имеющих строго k посещений при произвольном обходе кумулятивной суммы. Произвольный обход кумулятивной суммы начинается с частичных сумм после последовательности (0,1), переведенной в соответствующую последовательность (-1, +1). Цикл произвольного обхода состоит из серии шагов единичной длины, совершаемых в произвольном порядке. Кроме того, такой обход начинается и заканчивается на одном и том же элементе. Цель данного теста — определить отличается ли число посещений определенного состояния внутри цикла от аналогичного числа в случае абсолютно случайной входной последовательности. Фактически данный тест есть набор, состоящий из восьми тестов, проводимых для каждого из восьми состояний цикла: −4, −3, −2, −1 и +1, +2, +3, +4. [4]

Листинг 4.9 – модуль random\_excursion\_test

*from \_\_future\_\_ import print\_function*

*import math*

*#from scipy.special import gamma, gammainc, gammaincc*

*from gamma\_functions import \**

*# RANDOM EXCURSION TEST*

*def random\_excursion\_test(bits):*

*n = len(bits)*

*x = list() # Convert to +1,-1*

*for bit in bits:*

*#if bit == 0:*

*x.append((bit\*2)-1)*

*#print "x=",x*

*# Build the partial sums*

*pos = 0*

*s = list()*

*for e in x:*

*pos = pos+e*

*s.append(pos)*

*sprime = [0]+s+[0] # Add 0 on each end*

*#print "sprime=",sprime*

*# Build the list of cycles*

*pos = 1*

*cycles = list()*

*while (pos < len(sprime)):*

*cycle = list()*

*cycle.append(0)*

*while sprime[pos]!=0:*

*cycle.append(sprime[pos])*

*pos += 1*

*cycle.append(0)*

*cycles.append(cycle)*

*pos = pos + 1*

*J = len(cycles)*

*print("J="+str(J))*

*vxk = [['a','b','c','d','e','f'] for y in [-4,-3,-2,-1,1,2,3,4] ]*

*# Count Occurances*

*for k in range(6):*

*for index in range(8):*

*mapping = [-4,-3,-2,-1,1,2,3,4]*

*x = mapping[index]*

*cyclecount = 0*

*#count how many cycles in which x occurs k times*

*for cycle in cycles:*

*oc = 0*

*#Count how many times x occurs in the current cycle*

*for pos in cycle:*

*if (pos == x):*

*oc += 1*

*# If x occurs k times, increment the cycle count*

*if (k < 5):*

*if oc == k:*

*cyclecount += 1*

*else:*

*if k == 5:*

*if oc >=5:*

*cyclecount += 1*

*vxk[index][k] = cyclecount*

*# Table for reference random probabilities*

*pixk=[[0.5 ,0.25 ,0.125 ,0.0625 ,0.0312 ,0.0312],*

*[0.75 ,0.0625 ,0.0469 ,0.0352 ,0.0264 ,0.0791],*

*[0.8333 ,0.0278 ,0.0231 ,0.0193 ,0.0161 ,0.0804],*

*[0.875 ,0.0156 ,0.0137 ,0.012 ,0.0105 ,0.0733],*

*[0.9 ,0.01 ,0.009 ,0.0081 ,0.0073 ,0.0656],*

*[0.9167 ,0.0069 ,0.0064 ,0.0058 ,0.0053 ,0.0588],*

*[0.9286 ,0.0051 ,0.0047 ,0.0044 ,0.0041 ,0.0531]]*

*success = True*

*plist = list()*

*for index in range(8):*

*mapping = [-4,-3,-2,-1,1,2,3,4]*

*x = mapping[index]*

*chisq = 0.0*

*for k in range(6):*

*top = float(vxk[index][k]) - (float(J) \* (pixk[abs(x)-1][k]))*

*top = top\*top*

*bottom = J \* pixk[abs(x)-1][k]*

*chisq += top/bottom*

*p = gammaincc(5.0/2.0,chisq/2.0)*

*plist.append(p)*

*if p < 0.01:*

*err = " Not Random"*

*success = False*

*else:*

*err = ""*

*print("x = %1.0f\tchisq = %f\tp = %f %s" % (x,chisq,p,err))*

*if (J < 500):*

*print("J too small (J < 500) for result to be reliable")*

*elif success:*

*print("PASS")*

*else:*

*print("FAIL: Data not random")*

*return (success, None, plist)*

*if \_\_name\_\_ == "\_\_main\_\_":*

*bits = [0,1,1,0,1,1,0,1,0,1]*

*success, \_, plist = random\_excursion\_test(bits)*

*print("success =",success)*

*print("plist = ",plist)*

#### **Другой тест на произвольные отклонения**

В этом тесте подсчитывается общее число посещений определенного состояния при произвольном обходе кумулятивной суммы. Целью является определение отклонений от ожидаемого числа посещений различных состояний при произвольном обходе. В действительности этот тест состоит из 18 тестов, проводимых для каждого состояния: −9, −8, …, −1 и +1, +2, …, +9. На каждом этапе делается вывод о случайности входной последовательности. [4]

Листинг 4.10 – модуль random\_excursion\_variant\_test

*from \_\_future\_\_ import print\_function*

*import math*

*# RANDOM EXCURSION VARIANT TEST*

*def random\_excursion\_variant\_test(bits):*

*n = len(bits)*

*x = list() # Convert to +1,-1*

*for bit in bits:*

*x.append((bit \* 2)-1)*

*# Build the partial sums*

*pos = 0*

*s = list()*

*for e in x:*

*pos = pos+e*

*s.append(pos)*

*sprime = [0]+s+[0] # Add 0 on each end*

*# Count the number of cycles J*

*J = 0*

*for value in sprime[1:]:*

*if value == 0:*

*J += 1*

*print("J=",J)*

*# Build the counts of offsets*

*count = [0 for x in range(-9,10)]*

*for value in sprime:*

*if (abs(value) < 10):*

*count[value] += 1*

*# Compute P values*

*success = True*

*plist = list()*

*for x in range(-9,10):*

*if x != 0:*

*top = abs(count[x]-J)*

*bottom = math.sqrt(2.0 \* J \*((4.0\*abs(x))-2.0))*

*p = math.erfc(top/bottom)*

*plist.append(p)*

*if p < 0.01:*

*err = " Not Random"*

*success = False*

*else:*

*err = ""*

*print("x = %1.0f\t count=%d\tp = %f %s" % (x,count[x],p,err))*

*if (J < 500):*

*print("J too small (J=%d < 500) for result to be reliable" % J)*

*elif success:*

*print("PASS")*

*else:*

*print("FAIL: Data not random")*

*return (success,None,plist)*

Все приведенные выше модули представлены на листингах приложения Г.

## **Анализ результатов исследования**

В данном исследовании приводится результат оценки качества предложенного генератора псевдослучайных чисел. В таблицу 4.1 приведены результаты исследования.

Таблица 4.1. Результаты качественной оценки

|  |  |  |
| --- | --- | --- |
| Тест | Числовая оценка | Результат оценки |
| Частотный побитовый тест | 0.7324398999038725 | PASS |
| Частотный блочный тест | 0.3954669433496319 | PASS |
| Тест на серийность | 0.9145182387620326 | PASS |
| Спектральный тест | 0. 3740525925192202 | PASS |
| Тест с неперекрывающимися непериодическими шаблона-  ми | 0.6993018720649098 | PASS |
| Тест на периодичность | 0.8022450936184753 | PASS |
| Тест приблизительной энтропии | 0.46960616243661973 | PASS |
| Тест кумулятивных сумм | 0.6059354934001464 | PASS |
| Тест на произвольные отклонения | 0.3302969563320579 | PASS |
| Другой тест на произвольные отклонения | 0.21231716077296492 | PASS |

Как можно видеть, каждый из критериев оценки качества показал положительный результат. Исходя из этого можно утверждать с достаточной степенью уверенности в том, что предложенный метод генерации псевдослучайных чисел действительно имеет право на существование.

## **Сравнительный анализ ГПСЧ с аналогами**

Для сравнительного анализа были выбраны два генератора псевдослучайных чисел. Первый – это линейный конгруэнтный генератор, пожалуй, самый популярный из ГПСЧ. Второй – это генератор, основанный также на технологии клеточных автоматов, предложенный Д. Д. Мухамеджановым и А. Б. Левиной из университета ИТМО.

В таблице 4.2 приведены результаты оценки качества теми же критериями, что рассматривались ранее.

Таблица 4.2. Сравнение генераторов псевдослучайных чисел

|  |  |  |  |
| --- | --- | --- | --- |
| **Тест** | На основе игры «Жизнь» | На основе алгоритма NESW  (клеточный автомат) | Линейный конгруэнтный |
| Частотный побитовый тест | 0.7324398999038725 | 0,744146 | 0,739918 |
| Частотный блочный тест | 0.3954669433496319 | 0,380537 | 0,122325 |
| Тест на серийность | 0.9145182387620326 | 0,428244 | 0,213309 |
| Спектральный тест | 0. 3740525925192202 | 0,650637 | 0,639918 |
| Тест с неперекрывающимися непериодическими шаблона-  ми | 0.6993018720649098 | 0,578346 | 0,578346 |
| Тест на периодичность | 0.8022450936184753 | 0,651956 | 0,615983 |
| Тест приблизительной энтропии | 0.46960616243661973 | 0,778903 | 0,791468 |
| Тест кумулятивных сумм | 0.6059354934001464 | 0,734146 | 0,689918 |
| Тест на произвольные отклонения | 0.3302969563320579 | 0,610977 | 0,534146 |
| Другой тест на произвольные отклонения | 0.21231716077296492 | 0,633388 | 0,468312 |

Исходя из данных таблицы можно заметить, что предложенный генератор не только не уступает классическому методу генерации псевдослучайных чисел (линейно-конгруэнтный метод), но и превосходит его. Также можно наблюдать, что и второй генератор, также основанный на технологии клеточного автомата, превосходит классический метод генерации псевдослучайных чисел.

## **Вывод**

В исследовательском разделе проведена оценка качества разработанного метода генерации псевдослучайных чисел. Также проведен сравнительный анализ с существующими генераторами. Результаты исследований показали, что генератор действительно способен выдавать последовательности достаточно случайных чисел и не уступает классическому методу генерации, но и превосходит его. Также, такие же характеристики были замечены и за другим генератором псевдослучайных чисел, основанным на технологии клеточного автомата, что говорит о том, что данная технология несет в себе значительный потенциал для развития генераторов псевдослучайных чисел в целом.

# ЗАКЛЮЧЕНИЕ

В данной работе был представлен генератор псевдослучайных чисел, в основе которого лежит технология клеточных автоматов, а в качестве самого автомата использовалась игра «Жизнь» с модифицированными правилами. Выполнение данной работы сопровождалось рядом проблем, которые были либо решены, либо доведены до приемлемых положений.

Было разработано демонстрационное приложение для визуализации работы предложенного генератора. Оно имеет ряд входных параметров, среди которых заранее определенные начальные заполнения клеточного автомата, выбор которых являлся одной из самых острых вопросов, решаемых в ходе этой работы.

Также проведена оценка качества предложенного генератора, где последний продемонстрировал отличные показатели, в том числе в сравнительном анализе, где оказался лучше классического метода генерации.

В ходе сравнительного анализа также было замечено, что технология клеточных автоматов таит большой потенциал развития клеточных автоматов. Клеточные автоматы не только дают статистически лучшие результаты, но также и просты в разработке, удобно масштабируются, к ним можно применять многопоточность и параллельное программирование, что может значительно ускорить их работу.

Безусловно данная работа имеет потенциал развития, ибо не решены многие задачи. Так, существуют проблемы распределения, используемой памяти. Помимо прочего, данная реализация может быть перенесена на язык C, что позволит увеличить производительность, а также позволит лучше контролировать работу памяти, применить более продвинутые технологии, такие как многопоточность, например.

Еще одной задачей для решения является вопрос применимости данного генератора. Сейчас он может использоваться для узкого круга задач, как например для задач криптографии.

Кроме всего прочего, предложенный генератор имеет потенциал для комбинирования его с другими генераторами, что возможно позволит решить ряд задач, описанных выше.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Дональд Кнут. Искусство программирования, том 2. Получисленные алгоритмы = The Art of Computer Programming, vol.2. Seminumerical Algorithms.— 3-е изд. — М.: «Вильямс», 2007.
2. Т. Тоффоли, Н. Марголус, Машины клеточных автоматов// Мир 1991, стр. 8.
3. Aspray W. John von Neumann and the Origins of Modern Computing. MIT Press, 1990, стр. 376
4. Слеповичев И.И., ГЕНЕРАТОРЫ ПСЕВДОСЛУЧАЙНЫХ ЧИСЕЛ, май 21, 2017.
5. Документация Python 3.8 [Электронный ресурс]. URL: https://docs.python.org/3.8 (дата обращения 04.06.2021).
6. ГОСТ Р ИСО 28640-2012. [Электронный ресурс]. – Режим доступа: <http://files.stroyinf.ru/cgi-bin/ecat/ecat.fcgi?b=0&i=53898&pr=1>. Проверено 01.06.2021.
7. S. Wolfram, “Random Sequence generation by Cellular Automata”, Advances in Applied Mathematics, v. 7, 1986, pp.123-164.
8. W. Meier and O. Staffelbach, “Fast Correlation Attack on Stream Ciphers”, Journal of Cryptology v I n. 3, 1989, pp.159-176.
9. P.H. Bardell, “Analisis of Cellular Automata Used as Pseudorandom Pattern generators”, Proceedings of 1990 International Test Conference, pp. 762-768.
10. S. Wolfram, “Statistical mechanics of cellular automata”, Reviews of Modern Physics, Vol. 55, No. 3, July 1983, pp.8-13.
11. Мухамеджанов Д.Д., Левина А.Б. Генератор псевдослучайных чисел на основе клеточных автоматов // Научно-технический вестник информационных технологий, механики и оптики. 2018. Т. 18. № 5. С. 894–900. doi: 10.17586/22261494-2018-18-5-894-90
12. Деон А.Ф., Меняев Ю.А. Генератор равномерных случайных величин по технологии полного вихревого массива // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2017. № 2. C. 86–110. DOI: 10.18698/0236-3933-2017-2-86-110
13. Mirzoyan S.A. PSEUDORANDOM NUMBER GENERATOR BASED ON CELLULAR AUTOMATA AND THE GAME "LIFE" // International Scientific – Practical Conference «INFORMATION INNOVATIVE TECHNOLOGIES», 2021, стр. 280-285