МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

ОПТИМИЗАЦИИ ПОИСКА ПРИБЛИЖЕННЫХ РЕШЕНИЙ В ЭВОЛЮЦИОНИРУЮЩИХ КЛЕТОЧНЫХ АВТОМАТАХ

БАКАЛАВРСКАЯ РАБОТА

студента 4 курса 451 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Григорьева Алексея Александровича

Научный руководитель
доцент _____ М. С. Семенов

Заведующий кафедрой
к. ф.-м. н., доцент _____ А. С. Иванов

Саратов 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Клеточный автомат — дискретная модель, изучаемая теорией автоматов [1]. В общем случае, клеточный автомат состоит из регулярной решетки ячеек, множества состояний, множества входных сигналов и функции перехода. Клеточные автоматы применяются в компьютерном зрении [2], криптографии [3], компьютерной графике [4] и при моделировании различных процессов: физических [5], химических [6], биологических [7]. Одним из наиболее известных является клеточный автомат — игра «Жизнь» [8].

Существует огромное количество конфигураций клеточных автоматов. Меняя число возможных состояний, правила подсчета входных сигналов, возможные начальные состояния, получаются новые множества автоматов, перебор которых невозможен с текущими вычислительными мощностями. Поэтому для поиска клеточных автоматов, удовлетворяющих каким-либо условиям, используют алгоритмы нахождения приближенного результата. В недавних работах на схожую тему применялись генетические алгоритмы [9] и нейронные сети [10]. В настоящей работе перебор осуществляется с использованием генетических алгоритмов.

В основе работы лежит выполнение экспериментов по нахождению приближенного решения клеточными автоматами. Цель каждого эксперимента — стабильное выполнение заданных условий ячейками, полученными после работы клеточных автоматов из множества начальных состояний. В данной работе начальное состояние задано случайно в каждой ячейке автомата. Для наглядности в качестве цели, которая будет поставлена перед началом экспериментов, будет воспроизведение заданного наперед целевого изображения (паттерна). Благодаря этому можно будет не только полагаться на полученные числовые значения, но и сопоставить результат с визуализацией в приложении. После каждого эксперимента должна сохраняться исчерпывающая информация о поведении клеточного автомата. По ней должно быть возможно определить конфигурацию автомата, генетического алгоритма, был ли удачным эксперимент и сколько было затрачено времени.

Эксперименты по оптимизации проводились для двумерных клеточных автоматов первого порядка, с возможными состояниями 0 и 1. Следующее состояние ячейки клеточного автомата определяется 9 соседними ячейками на решетке включая данную или, другими словами, окрестностью Мура порядка

2 [11]. Тип автоматов выбран не случайно: количество всевозможных правил равно 2^{512} [12], что гарантирует невозможность перебора. Более того, не существует и общего аналитического решения для воспроизведения автоматами произвольного изображения, при условии что изначальная двумерная сетка задана случайными значениями. У выбранных в данной работе типов клеточных автоматов также есть преимущества, связанные с возможностью оптимизации алгоритмов с использованием быстрых побитовых операций [13]. Многие из описанных далее результатов возможно перенести на случай клеточных автоматов `N`-го порядка [14], а также для клеточных автоматов с увеличенным количеством состояний или иными правилами переходов.

Целью настоящей работы является исследование возможностей по оптимизации поиска приближенного решения на основе паттернов двумерными клеточными автоматами первого порядка при помощи генетического алгоритма. Для достижения этой цели необходимо:

1. разработать гибкое приложение, моделирующее работу клеточных автоматов с возможностью конфигурации генетического алгоритма;
2. добавить визуализацию клеточных автоматов и сбор статистики по экспериментам;
3. провести множество экспериментов с различными конфигурациями и целевыми изображениями;
4. сделать выводы, определить лучшие параметры генетического алгоритма, предложить возможные оптимизации.

1 Окружение для проведения экспериментов

В данном разделе приведено описание проекта и вспомогательных функций для моделирования и визуализации двумерных клеточных автоматов первого порядка.

1.1 Среда разработки

Программа для моделирования и визуализации клеточных автоматов выполнена в среде разработки `Unity 2019.2.21f`. Выбор среды разработки обоснован следующими удобствами:

— быстрая и многофункциональная работа с графикой посредством готовых функций и визуального проектирования;
— профессиональные инструменты для отладки и профилирования производительности программы;
— наглядность результата программы.

Программный код написан на языке `C#`. Основная логика экспериментов реализована в сцене MainScene. Весь проект подключен к системе контроля версий Git [15] на базе популярного ресурса для хранения версий Github. Ссылка на «репозиторий»: `https://github.com/Alexflames/cellular-automation`.

Со структурой проекта можно ознакомиться в приложении Г.

1.2 Реализация клеточного автомата

Каждый клеточный автомат представляется ячейками на двумерной сетке. Для прозрачности процесса и наглядности результатов, эксперимент проводится полностью в реальном времени: в каждую единицу времени все ячейки автоматов на всех сетках переходят в новое состояние, определенное входными сигналами и правилами перехода.

Опишем правила перехода для двумерного клеточного автомата первого порядка. При подсчете входного сигнала в ячейке автомата будут учитываться как и состояние данной ячейки, так и состояния восьми соседних ячеек в окрестности Мура порядка 2, для подсчета берутся значения с последней итерации. Возьмем самый простой случай: автомат может принимать значения 0 и 1. Таким образом, возможны $2^9 = 512$ вариантов входных сигналов. Отсюда и берется упомянутое ранее количество всевозможных правил для данных клеточных автоматов, равное 2^{512}.

В качестве визуализации клеточных автоматов в Unity можно использовать шейдеры. Плюс этого подхода заключается в том, что компьютер не будет тратить ресурсы оперативной памяти для визуализации автоматов, и вся нагрузка на это отдается видеокарте, находящейся большую часть времени в простое. Шейдеры позволяют создавать крупные визуализации огромной размерности благодаря эффективным вычислениям на видеокарте [16].

1.2.1 Правило перехода

Создадим простейшую реализацию подсчета входных сигналов для ячеек в клеточном автомате. Для избежания лишних затрат памяти, в основной части программы заранее созданы массивы для временного хранения следующих состояний ячеек автомата. Это необходимо, чтобы полученный результат не влиял на соседние ячейки в данный момент времени.

Создадим цикл, который будет выполняться на двумерной сетке клеточных автоматов размерностью `screenSizeInPixels` по X и Y.

```
var nextCAField = nextVirtualScreens[ind];
var screen2DSize = screenSizeInPixels * screenSizeInPixels;
for (short i = 0; i < screenSizeInPixels; i++)
{
   for (short j = 0; j < screenSizeInPixels; j++)
   {
       int signal = 0;
       for (short k = 0; k < 3; k++)
       {
           for (short m = 0; m < 3; m++)
           {
               signal += CAField[
                   (screen2DSize + screenSizeInPixels * i + j
                   + screenSizeInPixels * (k - 1) + (m - 1))
                   % (screen2DSize)] << (k * 3 + m);
           }
       }
       nextCAField[i * screenSizeInPixels + j] = (byte)allRules[ind][signal];
   }
}

for (short i = 0; i < screen2DSize; i++)
{
   CAField[i] = nextCAField[i];
}
```

1.2.2 Оптимизированное вычисление входных сигналов

Заметим, что в вычислении следующего состояния каждого автомата используется 9 обращений к сетке. Более того, при каждом последующем обращении к сетке после первого, повторно считываются 6 состояний. Представим квадрат 3×3, окрестность Мура порядка 2, в виде окна, которое можно сдвигать вправо по сетке. Разобьем окно на 3 горизонтальные буфер-линии и, при переходе в новое состояние (аналогично сдвигу окна вправо), будем считывать лишь 3 новых бита справа. Добавление каждого бита в строку-буфер сопровождается побитовым сдвигом строки на 1 влево с отсечением крайнего левого бита. Сигналом для текущего автомата будет сумма верхней строки, средней строки с побитовым сдвигом влево на 3 и нижней строки с побитовым сдвигом на 6. Для наглядности процесс изображен на рисунке 1.

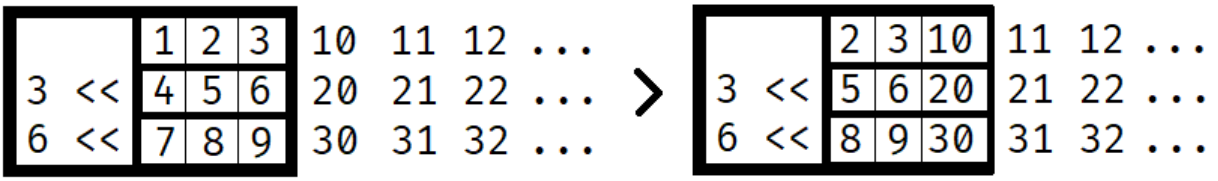

Рисунок 1 – Принцип оптимизированного вычисления входных сигналов с использованием буферов и побитовых сдвигов.

Результат аналогичен тому, что было получено в предыдущем разделе 1.2.1, но будет требоваться примерно в 3 раза меньше операций доступа. Полученная оптимизация особенно важна для нас, ведь обновление всех автоматов происходит каждый кадр. Полный код новой функции можно посмотреть в приложении А.

1.2.3 Правило перехода на языке шейдеров (HLSL)

Обновление клеточных автоматов возможно полностью перенести в шейдеры. Каждая сетка с клеточным автоматом будет использовать свою текстуру и шейдер.

Для написания правила перехода на языке шейдеров необходимо решить три проблемы:

1. получение информации о состояниях в предыдущий момент времени;
2. соотнесение представления в битах (пикселях) с текстурой в трехмерном пространстве;
3. обновление шейдера только при необходимости.

Возьмем Custom Render Texture из среды разработки Unity. Данная текстура создана для того, чтобы её можно было обновлять с помощью шейдеров. Первая проблема решается расширением языка HLSL, предоставляемым Custom Render Texture. В фрагментном шейдере можно обратиться к переменной localTexcoord, в которой хранится состояние текстуры на предыдущий момент отрисовки.

Вторая проблема также решается с использованием глобальных переменных: _CustomRenderTextureWidth, предоставляющая реальную ширину объекта в пространстве, и _CustomRenderTextureHeight, возвращающая высоту текстуры.

Определим настройки Custom Render Texture (рис. 2). Установка параметра Wrap Mode в значение Repeat задаст периодические граничные условия для сетки. Другими словами, при обращении к пикселям по координатам, превышающим размерность текстуры, будут взяты пиксели с противоположной стороны сетки. Создадим пустой шейдер, код для которого опишем дальше. Настройка Update Mode в значение On Demand позволяет обновлять состояние текстуры по вызову функции из основной программы, что позволит получить полный контроль над обновлением автоматов.

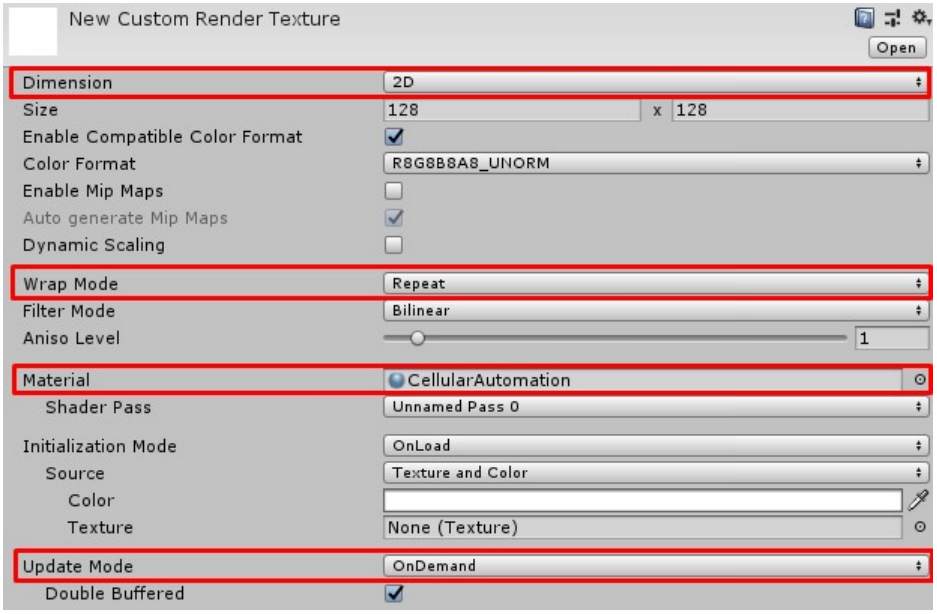

Рисунок 2 – Настройка Custom Render Texture.

В шейдерое определим uniform-переменную: таблицу переходов размерности 512 (все возможные состояния рассматриваемого клеточного автомата).

```
float _rule[512];
```

Опишем функцию, возвращающую значение пикселя с заданным отступом в сетке относительно текущего клеточного автомата.

```
float4 get(v2f_customrendertexture IN, int x, int y) : COLOR
{
   return tex2D(_SelfTexture2D, IN.localTexcoord.xy +
            fixed2(x / _CustomRenderTextureWidth, y / _CustomRenderTextureHeight));
}
```

Аналогично построению двоичного числа, последовательно, с левого верхнего угла, соберем биты входных сигналов для клеточного автомата в окрестности Мура порядка 2 и получим следующее состояние из таблицы переходов. Данный код необходимо выполнять в фрагментном шейдере [17]

```
float getRule9(v2f_customrendertexture IN) : float
{
   int accumulator = 0;
   for (int i = 2; i >= 0; i--)
   {
       for (int j = 0; j <= 2; j++)
       {
           int roundedAlpha = round(get(IN, i-1, j-1).a);
           accumulator = (accumulator << 1) + roundedAlpha;
       }
   }
   return _rule[accumulator];
}

float4 frag(v2f_customrendertexture IN) : COLOR
{
   return getRule9(IN);
}
```

Осталось объединить шейдер с основной программой. Пусть в массиве `allRules` находятся правила для всех клеточных автоматов. Тогда, опуская подробности, инициализацию текстуры можно произвести с помощью следующего кода

```
customRenderTexture.material.SetFloatArray("_rule", allRules[screenInd]);
customRenderTexture.Initialize();
```

Пусть в массиве `custormRenderTextures` содержатся все текстуры (сетки). Тогда, обновить каждый автомат можно, используя следующий фрагмент кода:

```
foreach(var texture in customRenderTextures)
{
   texture.Update();
}
```

Запустим программу и увидим клеточные автоматы в динамике, как изображено на рис. 3.

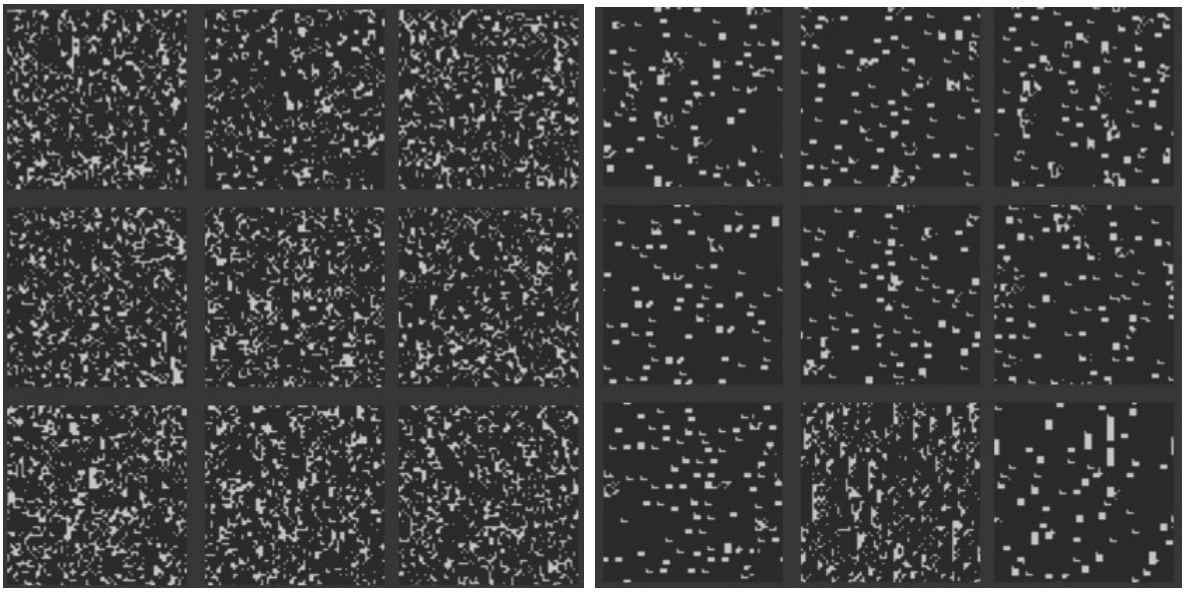

Рисунок 3 – Визуализация клеточных автоматов.

2 Генетический алгоритм для поиска клеточных автоматов

Для получения приближенных результатов, воспользуемся генетическим алгоритмом [18]. **Особи** — сетки с клеточными автоматами, **гены** — правила перехода для клеточных автоматов, **приспособленность** особи — суммарная по двумерной сетке степень совпадения каждой окрестности ячейки клеточного автомата некоторому целевому изображению (паттерну), воспроизведение которого и будет являться целью каждого эксперимента.

2.1 Подсчет приспособленности

Для правильной работы генетического алгоритма, нам нужно различать «полезные» особи от «ненужных». Для этого определим приспособленность особи. Приспособленностью особи будем считать процент совпадения окрестностей клеток на сетке заданному перед началом эксперимента паттерну.

При подсчете приспособленности учитывается количество допустимых ошибок при сравнении с паттерном. Если число пикселей, не совпадающих с целевым изображением, превышает порог, мгновенно переходим на следующую итерацию. В противном случае, если X — максимальное количество допустимых ошибок, а Y — число несовпадений в паттерне, тогда к ответу прибавим $(1+X-Y)/(X+1)$. Добавляя такой параметр, мы намеренно рискуем получить искаженный результат, не совпадающий полностью с паттерном, но зачастую такая мера необходима для значительного уменьшения времени сходимости к приемлемому результату. Однако выбранная нами формула стимулирует автомат находить результат без ошибок, что частично компенсирует недостаток, связанный с возможностью получения искаженного результата.

Также необходимо учесть, что мы можем искать совпадение не с единственным изображением, а с целым набором, как показано на изображении 4. В таком случае, на каждой итерации, при проверке совпадения окрестности с паттерном, обрабатывая каждую ячейку, к аккумулятору приспособленности добавляется максимальное значение приспособленности по всем паттернам.

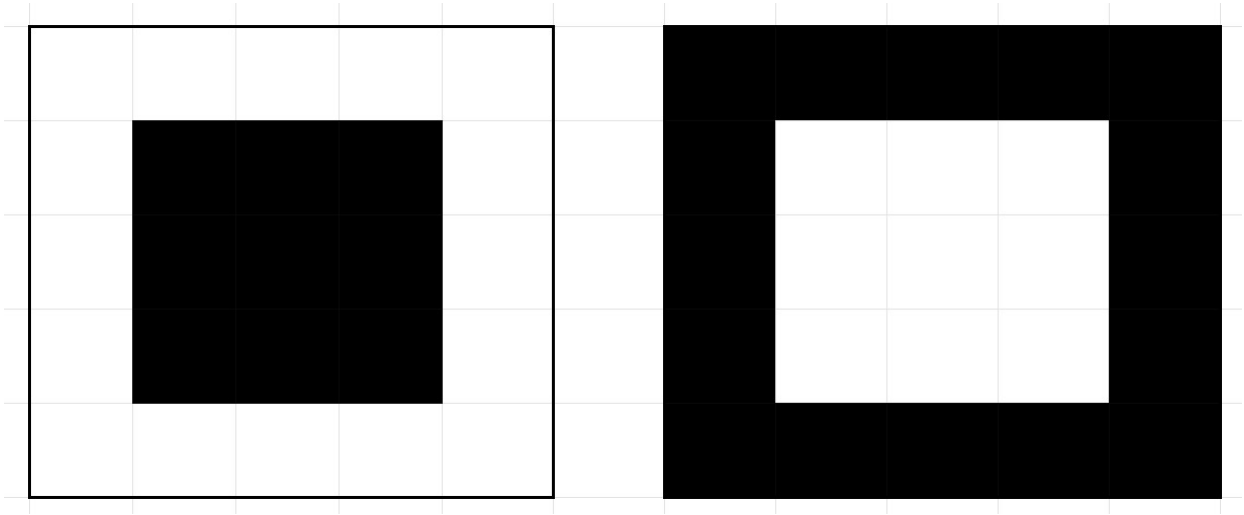

Рисунок 4 – Пример случая, когда проверяется совпадение с несколькими паттернами

Полный код функции приведен в приложении Б.

В идеальном случае, нам хотелось бы высчитать некоторую нормализованную функцию приспособленности, которая бы для сетки любой размерности и любого набора паттернов давала бы значение в диапазоне от 0 до 1 (или 100, для наглядности). Подобная функция учитывала бы максимально возможное количество изображений на паттерне. Таким образом мы могли бы поделить количество вхождений паттерна в изображении на максимально возможное и измерить процент совпадения. В реальности же это достижимо только с помощью полного перебора всей сетки, что в самом простом случае, без оптимизаций, при размере изображения 64×64, требовало бы произвести $2^{64 \times 64}$ подсчетов приспособленности. Поэтому для упрощения, будем считать значение приспособленности равным 100 если изображение полностью состоит из паттернов без пересечения. Например, если исходное изображение имеет размерность 64×64, а паттерн — 4×4, то на изображении возможно разместить $(64 \times 64)/(4 \times 4) = 256$ копий паттерна. В таком случае приспособленность будет равна 100.

Подведем подсчет приспособленности на последних итерациях и сделаем возможным устанавливать количество итераций перед процессом эволюции. Между двумя подсчетами приспособленности происходит ровно одно обновление клеточных автоматов на сетке.

Если запустить программу с профилировщиком, можно заметить, что самым затратным по времени местом в программе является именно подсчет приспособленности (см. рис. 5). Это не удивительно, ведь для каждого пикселя

сетки приходится проверять соответствие окрестности заданному паттерну, а с увеличением его размера в X раз, в столько же раз увеличивается и количество операций доступа при подсчете приспособленности. Далее попробуем оптимизировать этот процесс.

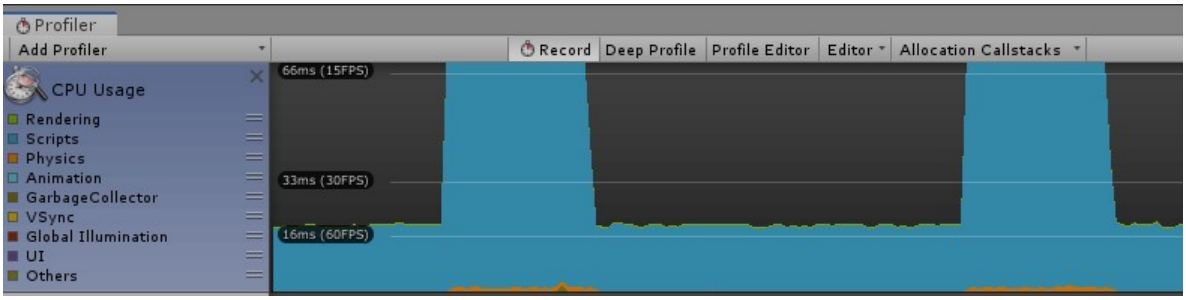

Рисунок 5 – Фрагмент окна профилировщика. Видимые подъемы связаны с процессом подсчета приспособленности

2.2 Подсчет приспособленности на основе данных с GPU

Попробуем снизить нагрузку во время подсчета приспособленности клеточных автоматов на сетке за счет переноса обновления сетки с процессора на видеокарту. Так как визуализация клеточных автоматов с помощью шейдеров уже готова, достаточно считывать изображение перед каждым подсчетом приспособленности.

После считывания, переведем пиксели цвета изображения в числа 0 и 1 и воспользуемся готовой функцией подсчета приспособленности.

```
private float CalculateFitness(Texture2D texture2D, Pattern[] patterns)
{
  texturePixels = texture2D.GetPixels32();
  return CalculateFitness(texturePixels, texture2D.width, texture2D.height, patterns);
}

byte[] texturePix = null;
private float CalculateFitness(Color32[] texturePixels, int texW,
        int texH, Pattern[] patterns)
{
  var texSize = texW * texH;
  texturePix = new byte[texSize];
  for (int i = 0; i < texSize; i++)
  {
      texturePix[i] = texturePixels[i].a;
  }
  return CalculateFitness(texturePix, texW, texH, patterns);
}
```

Вопреки ожиданиям, это не ускорило программу. Это можно объяснить тем, что скорость передачи данных с видеопамяти в оперативную память крайне медленная [19].

2.3 Оптимизация подсчета приспособленности

Воспользуемся оптимизацией, описанной в разделе 1.2.2, с отличием в том, что паттерн может быть произвольного размера. Для начала переведем двоичные числа, образованные последовательным считыванием слева направа каждой строки паттерна и переводом в десятичное число и сохраним эту информацию во вспомогательном массиве. Данные числа останутся неизменными. Далее, проведем сравнение битов сетки клеточного автомата и паттерна.

Повторим действия, описанные в разделе 1.2.2 для сетки клеточного автомата. Аналогично, переведем двоичные числа, образованные битами, в десятичные и запишем в массив. Таким образом, размерность массива по обоим измерениям будет совпадать с полученным для паттерна.

Имея два двоичных числа в десятичном представлении, мы можем получить новое двоичное число, которое будет состоять исключительно из различающихся битов в двух исходных числах. Делается это с помощью операции XOR (сложение по модулю 2). В языке программирования `C#` она обозначается как `^`.

В результате, мы получаем новое десятичное число, количество единиц в двоичном представлении которого равно количеству битов в окрестности ячейки, **не** совпадающих с паттерном. Для быстрого подсчета единиц, воспользуемся параллельным SWAR-алгоритмом, подробную информацию про который можно посмотреть в источнике [20].

После подсчета разницы, для каждого числа-буфера, полученного для сетки, используем побитовый сдвиг влево с выбрасыванием крайнего левого бита (используя бинарное «И») и добавим новое число справа. Повторим алгоритм, пока не дойдем до конца сетки.

Далее вставлен основной фрагмент описанного кода. Полный код доступен в приложении B.

```
for (short j = 0; j < textureWidth; j++)
 {
   cornerPixel = j + i * textureWidth;
   for (byte p = 0; p < patterns.Length; p++)
```

```
    {
        curErr[p] = 0;
    }
    int minErrors = patternErrors + 1;

    for (byte k = 0; k < patternHeight; k++)
    {
      int ind = (qOffset + k) % patternHeight;
      screenLine = screenLines[ind];
      newErr = patternLines[p][k] ^ screenLine;

      for (byte p = 0; p < patterns.Length; p++)
      {
        newErr = newErr - ((newErr >> 1) & 0x55555555);
        newErr = (newErr & 0x33333333) + ((newErr >> 2) & 0x33333333);
        curErr[p] += (((newErr + (newErr >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
      }

      screenLine <<= 1;
      if (screenLine >= patternCycle) screenLine -= patternCycle;
      nextPixInd = (((i + k) % textureHeight) * textureWidth)
              + (j + patternWidth) % textureWidth;
      screenLine += texturePixels[nextPixInd];

      screenLines[(qOffset + k) % patternHeight] = (screenLine);
    }

    for (byte p = 0; p < patterns.Length; p++)
    {
        if (curErr[p] < minErrors) minErrors = curErr[p];
    }
    fitness += minErrors <= patternErrors ?
     (1 + patternErrors - minErrors) * 1f / (patternErrors + 1) : 0;
}
```

Даже с учетом подобных и ранее описанных оптимизаций в разделе 1.2.2, данный этап остается самым высоко нагруженным. Поэтому для пользователя программы сделана возможность определить количество подсчетов приспособленности на каждом цикле эволюции перед этапом скрещивания.

2.4 Эволюция

Этап эволюции начинается после завершения подсчетов приспособленности клеточных автоматов на последних кадрах. На основании полученных данных будет отбираться половина особей с наивысшей приспособленностью, из них случайным образом будут созданы пары, каждая из которых создаст два

потомка. Правила переходов для потомков формируются из генов родителей, которые также сохраняются до следующего этапа эволюции. В работе будут рассмотрены два вида скрещивания.

Для первого вида скрещивания выберем случайный номер бита в генах (правила перехода длиной 512), будем называть его «разделителем». Все биты первого родителя до разделителя станут основой для правила перехода первого дочернего клеточного автомата, все биты с номером $\geq$ разделитель возьмутся из второго родителя. Для второго дочернего клеточного автомата, до разделителя берутся биты из второго родителя, а после — из первого.

Второй вид скрещивания реализован с помощью случайного выбора родителя для **каждого** бита.

После скрещивания случайным образом мутируем гены автоматов, полученных после этапа эволюции. В случае клеточных автоматов с возможными состояниями 0 и 1, 0 меняется на 1 и наоборот. Разрешим до X мутаций, а вероятность того что выбранный клеточный автомат мутирует — Y. Оба параметра настраиваются пользователем перед экспериментом.

2.5 Сбор данных и визуализация эксперимента

Перед началом эксперимента по поиску клеточного автомата, воспроизводящего паттерн, случайным образом создадим идентификационный номер (ID) эксперимента. Название сопутствующих файлов со статистикой будет содержать в суффиксе ID эксперимента. Во время исполнения эксперимента, можно наблюдать за графиком приспособленности (рис. 6) и генофондом популяции (рис. 7-8).

После каждого эксперимента статистика записывается в файлы. Она содержит полную информацию о настройках и результатах (средняя приспособленность, время, количество итераций эволюции) эксперимента, максимальные и средние значения приспособленности на каждой итерации, опорные биты (см. раздел 3.2). Данные файлы расположены в папке:

`{Расположение_проекта}/Assets/SimulationData/`

После завершения эксперимента, в файл также записываются и правила переходов в виде генов. Файл расположенный в пути:

`{Расположение_проекта}/Assets/SimulationData/Genes/`

Каждый файл имеет название `G-{Имя_Паттерна}-{IDэксперимента}.txt`.

В нем ген каждого клеточного автомата записывается в новой строке,

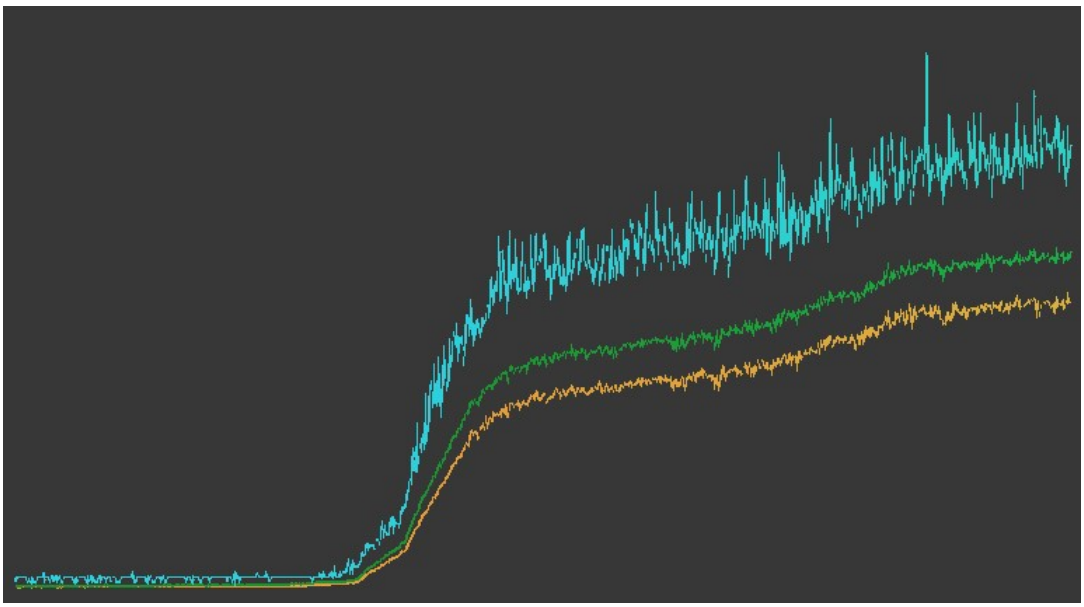

Рисунок 6 – Динамически построенный график приспособленности на эксперименте. Подробные числовые значения доступны в файлах с префиксом «FH-».

и, соответственно, количество строк в файле определяется числом клеточных автоматов в проведенном эксперименте.

Для просмотра клеточных автоматов после эксперимента создана дополнительная сцена в Unity с названием GeneVisualisation. В основном, она отличается от главной сцены тем, что в ней имеется одна большая сетка размерности 1024×1024 и отключена возможность эволюции автоматов. Для считывания файла с генами создан дополнительный скрипт. В редакторе необходимо выбрать автоматически созданный файл с данными генов. Запустив сцену, можно увидеть, как сетка меняет свое изначальное изображение 9, используя правила перехода из файла. Для переключения между всеми правилами в наборе, необходимо нажать стрелку влево или стрелку вправо на клавиатуре.

Нажатием клавиши «P» можно приостановить обновление клеточного автомата, повторное нажатие возобновляет его работу.

Нажатием клавиши «R» можно обновить состояние ячеек клеточного автомата в изначальное, то есть, значение каждой ячейки будет случайным.

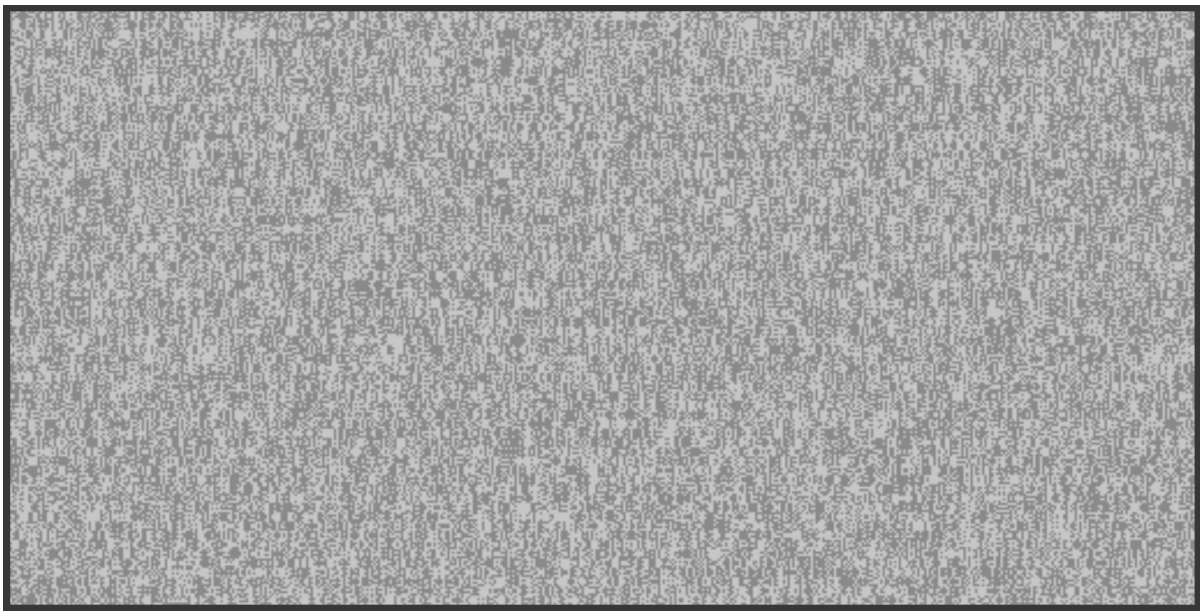

Рисунок 7 – Генофонд популяции на старте. По оси X — биты генов от 0 до 512, по оси Y — клеточные автоматы в популяции (256)

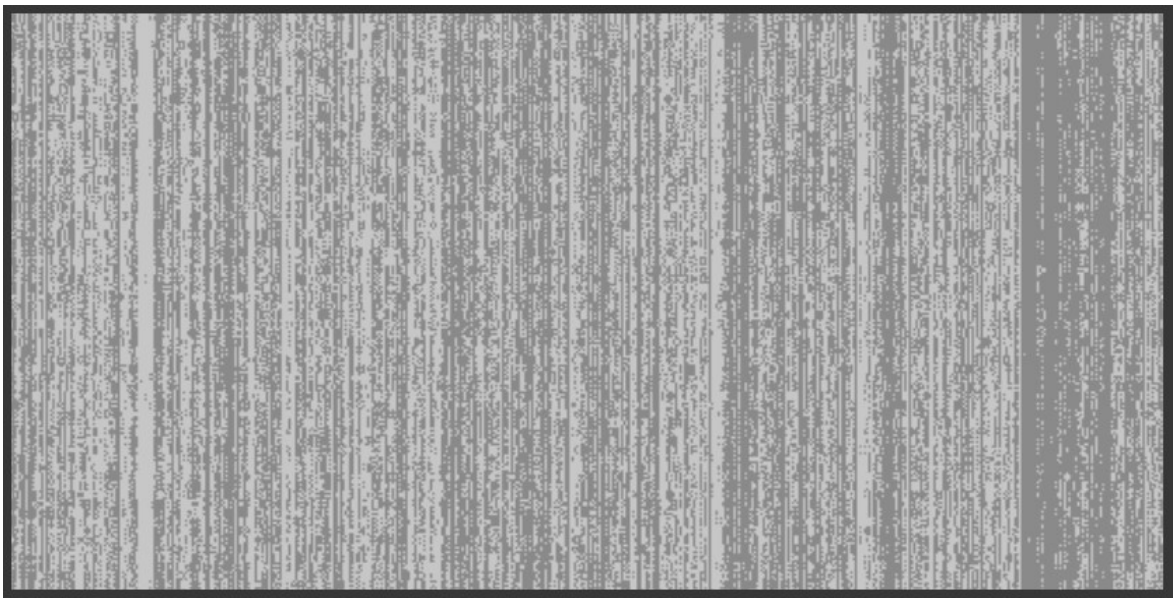

Рисунок 8 – Генофонд популяции в автомате с высокой приспособленностью. Можно заметить, что некоторые биты гена остаются неизменными у всех особей

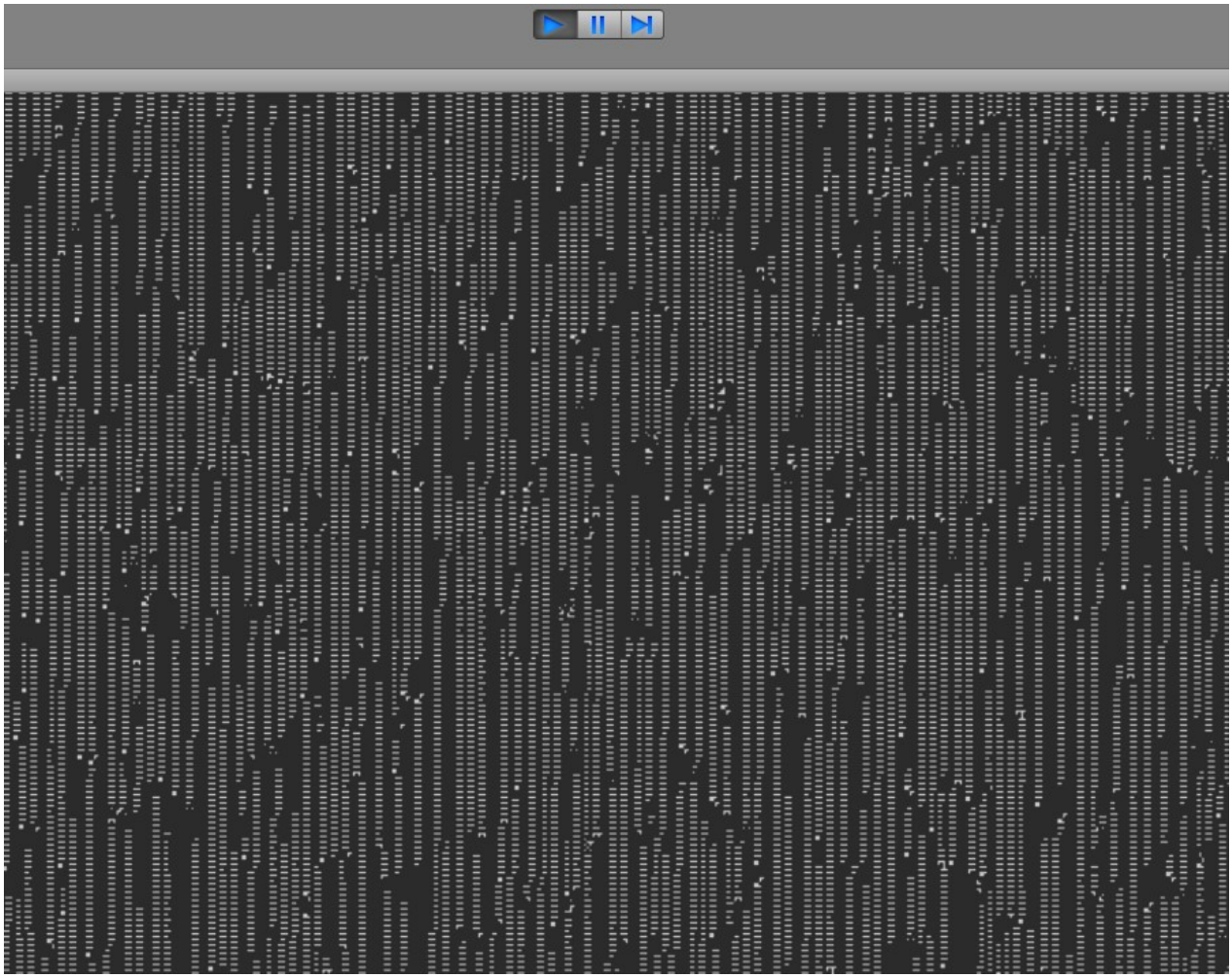

Рисунок 9 – Сцена для визуализации клеточных автоматов после эксперимента. Данный результат получен при попытке воссоздать паттерн «квадрат 3×3 с рамкой в 1 пиксель противоположного цвета».

3 Эксперименты

Данный раздел будет посвящен проведению экспериментов для сбора статистики, с помощью которой в дальнейшем можно выявить зависимость между параметрами эксперимента и скоростью нахождения клеточных автоматов, удовлетворяющих заданным условиям

Перед выполнением экспериментов были поставлены следующие вопросы. Возможно ли найти такие параметры модели, которые для любого паттерна будут приводить к более быстрому нахождению клеточного автомата, моделирующего его?

3.1 Проведение эксперимента

Перед проведением эксперимента необходимо определить его конфигурацию: количество клеточных автоматов, параметры генетического алгоритма и предельное значение количества итераций или приспособленности, при котором эксперимент будет завершен. Также должен быть указан файл, содержащий целевые изображения, которые клеточный автомат попытается воспроизвести.

3.1.1 Конфигурация параметров эксперимента

Перед началом эксперимента в окне редактора (см. рис. `Unity` 10) можно определить следующие значения параметров эксперимента:

- `Update Period` — минимальное время между кадрами перед следующим обновлением автомата;
- `Screen Size In Pixels` — размер квадратной сетки (в количестве ячеек) по одному измерению;
- `Virtual Screens In Simulation` — (кратное 4) количество клеточных автоматов в эксперименте;
- `Screens In Simulation` — количество автоматов, для которых будет создана визуализация;
- `Time To Evolution` — время до первого подсчета приспособленности автоматов;
- `Mutation Percent` — вероятность, определенная для каждого автомата, того, что он мутирует;
- `Mutate Bits Up To` — X, количество битов, которые могут мутировать: от 1 до X;

— `Pivot Bit Fitness Threshold` — пороговое значение приспособленности, начиная с которого будут отслеживаться опорные биты (подробнее в разделе 3.2);
— `Write To Global Pivot Bits` — записывать ли опорные биты в данном эксперименте в файл (подробнее в разделе 3.2);
— `Fitness Calculations Needed` — количество подсчетов приспособленности перед этапом эволюции;
— `MS Fitness Threshold` — пороговое значение приспособленности для завершения эксперимента и перехода к следующему;
— `MS Calculations After Threshold` — дополнительное количество этапов эволюции после достижения порога (проверка, может ли клеточный автомат развиваться дальше, даже после достижения указанного порога);
— `Evolution Step Limit` — пороговое значение количества этапов эволюции для завершения эксперимента и перехода к следующему;
— `Cross Separation` — проводить ли скрещивание на основе разделителя (подробнее в разделе 2.4).

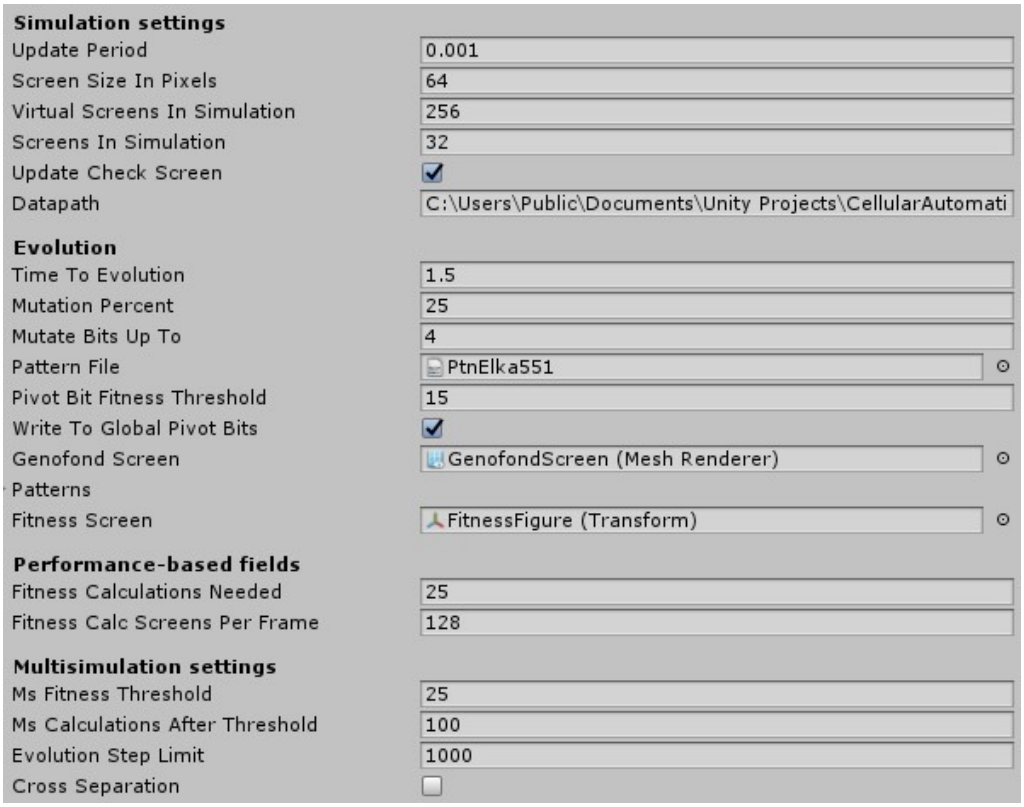

Рисунок 10 – Конфигурация эксперимента.

3.1.2 Создание файла с паттернами

Файлы с паттерном определяют целевое изображение, воспроизведение которого клеточным автоматом будет выполнено после применения генетического алгоритма. В первой строке указывается ширина и высота целевого изображения в пикселях. Третьим числом идет количество допустимых несовпадений в окрестности, при которых по-прежнему будет считаться, что паттерн нашелся.

Следующая строка может содержать любые символы, она оставлена на случай необходимости дополнительных конфигураций генетического алгоритма.

Затем необходимо указать количество паттернов и, используя 0 и 1, построчно записать пиксели паттерна. 0 — черный цвет, 1 — белый. Пример заполнения файла и соответствующее представление целевого изображения показано на рисунке 11.

```
5 5 2
ADDITIONAL_TAG ADDITIONAL_TAG_2
2
1 1 1 1 1
1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
1 1 1 1 1

0 0 0 0 0
0 1 1 1 0
0 1 1 1 0
0 1 1 1 0
0 0 0 0 0
```

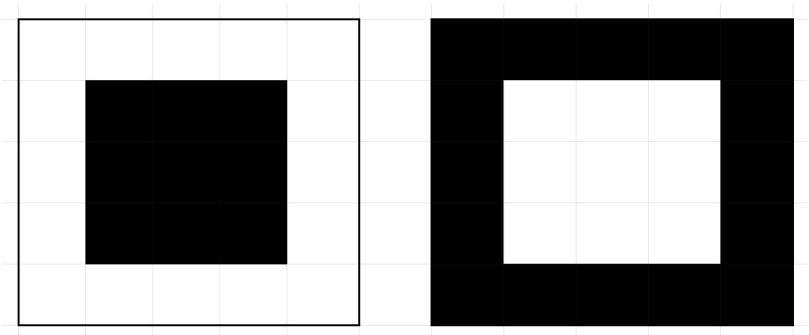

Рисунок 11 – Создание файла с информацией о паттерне. На втором изображении показаны описанные в файле целевые изображения.

3.1.3 Запуск программы

Перед запуском экспериментов необходимо удостовериться, что путь, указанный в настройках эксперимента, соответствует реальному в файловой системе. Для начала экспериментов необходимо либо нажать кнопку запуска внутри среды разработки `Unity`, либо собрать приложение, перейдя во вкладку `File->Build Settings`.

Приложение будет работать до тех пор, пока не будет отключено. В файл с результатами будет последовательно записываться информация о каждом проведенном эксперименте.

3.1.4 Результаты эксперимента

После каждого эксперимента остается блок сообщений 12. Далее следует описание основных строк блока:

1. номер эксперимента и время проведения;
2. `OK` — эксперимент завершился успешно, `ABORT` — эксперимент прерван, что может произойти если его выполнение остановлено вручную или достигнуто предельное количество шагов эволюции, и дополнительно указывается прошедшее количество этапов эволюции с начала эксперимента (каждый этап: обновление автоматов, подсчет приспособленности, скрещивание);
3. количество времени в секундах, потраченного на эксперимент и итоговая средняя приспособленность по всем клеточным автоматам;
4. в последующих строках следует информация о конфигурации эксперимента, подробнее в разделе 3.1.1.

```
Simulation [9185,813] -- 09.06.2020 10:28:01
ABORT|Simulation aborted for PtnFive755 after 1001 evolutions.
Time spent: 4583,473. Average good fitness: 0.
Virtual screens: 128. Update period: 0,001. Time to evolution: 1,5.
Fitness calculations between evolution: 25.
Fitness threshold: 25. Additional steps after threshold: 100.
Simple cross separation: False.
Mutation percent: 25. Mutate up to 8 bits.


Simulation [7321,073] -- 09.06.2020 11:30:24
OK|Simulation finished for PtnFive755 after 966 evolutions.
Time spent: 4074,686. Average good fitness: 25,2676.
Virtual screens: 128. Update period: 0,001. Time to evolution: 1,5.
Fitness calculations between evolution: 25.
Fitness threshold: 25. Additional steps after threshold: 100.
Simple cross separation: False.
Mutation percent: 8. Mutate up to 8 bits.
```

Рисунок 12 – Вывод в результате двух экспериментов: первый закончился неудачно после 1000 шагов эволюции, второй завершился успешно с приспособленностью 25,2676 после 966 шагов эволюции. В первом эксперименте вероятность мутации каждого гена 0.25, во втором — 0.08 (последняя строка в сообщении).

Информация по каждому проведенному эксперименту содержатся в ре-

позитории на Github в папке Statistics по ссылке:

https://github.com/Alexflames/diploma-cellular-automata

3.2 Поиск закономерностей в таблице переходов

Предположим, что для любого правила клеточного автомата существуют биты, являющиеся наиболее важными при формировании определенного паттерна. Далее, для удобства, будем называть эти биты «опорными». Тогда, для их нахождения создадим дополнительный массив, в котором будем хранить «ценность» бита. Определим её следующим образом:

1. К значению ценности бита прибавим 1, если бит гена равен единице, и -1 в противном случае.
2. Если приспособленность автомата меньше средней, то взять результат из п.1 и домножить на -1.
3. Итоговая ценность бита берется как значение по модулю от ценности бита.

Будем считать **опорными** те биты, значение ценности по модулю которых превосходит среднее значение (по модулю) ценности среди всех битов правила.

Введенный нами алгоритм объясняется предположением, что в правиле найдутся биты, которые будут присутствовать только в автоматах с высокой приспособленностью. Добавим дополнительное ограничение на то, что подсчет опорных битов будет вестись лишь начиная с заданного процента приспособленности. Для наглядности будем отображать опорные биты на генофонде клеточных автоматов отдельным (оранжевым) цветом, как на изображении 13.

Рисунок 13 – Генофонд автоматов. Оранжевым цветом отображаются опорные биты.

Однако, само по себе нахождение опорных битов в одном эксперименте не позволит сделать выводы о закономерностях в правилах. Для того чтобы узнать, сохраняются ли опорные биты для повторных попыток найти правила, воспроизводящих заданный паттерн, необходимо найти «глобальные опорные биты». После каждого эксперимента запишем все опорные биты в новую строку в файл. Перед началом следующего эксперимента считаем строки с опорными битами и прибавим $+1$ к каждому глобальному значению ценности бита. Все биты, глобальное значение ценности которых превышает половину числа строк опорных битов, будут считаться глобальными опорными битами. Отобразим их на генофонде отдельным (бирюзовым) цветом, как на изображении 14.

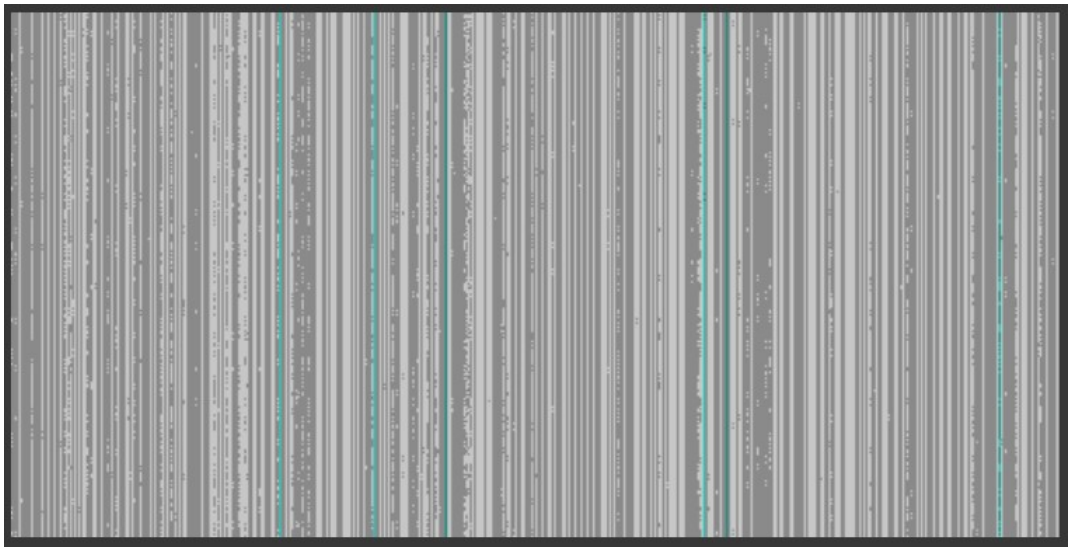

Рисунок 14 – Генофонд автоматов. Бирюзовым цветом отображаются глобальные опорные биты.

Полученные результаты противоречат предположению. С каждым последующим экспериментом количество глобальных опорных битов стремится к 0. С одной стороны, это можно объяснить тем, что существует множество изображений, удовлетворяющих цели эксперимента. Более того, для каждого целевого изображения существует множество клеточных автоматов, воспроизводящих его. Созданные клеточные автоматы могут быть периодичными и статичными.

Например, при попытке воспроизвести квадрат 3×3, клеточные автоматы могут создавать множество разнообразных фигур. Внешне это может выглядеть как если бы квадраты двигались в одну из 8 сторон или как одни квадраты появляются на месте других. Учитывая, что во многих паттернах мы

для снижения времени экспериментов позволяем допускать ошибки, то полученные изображения будут выглядеть как исходные, и при этом иметь незначительные различия. Однако, эти различия играют важную роль в правилах клеточных автоматов. Некоторые клеточные автоматы для паттерна «квадрат 3×3» изображены на рисунке 15

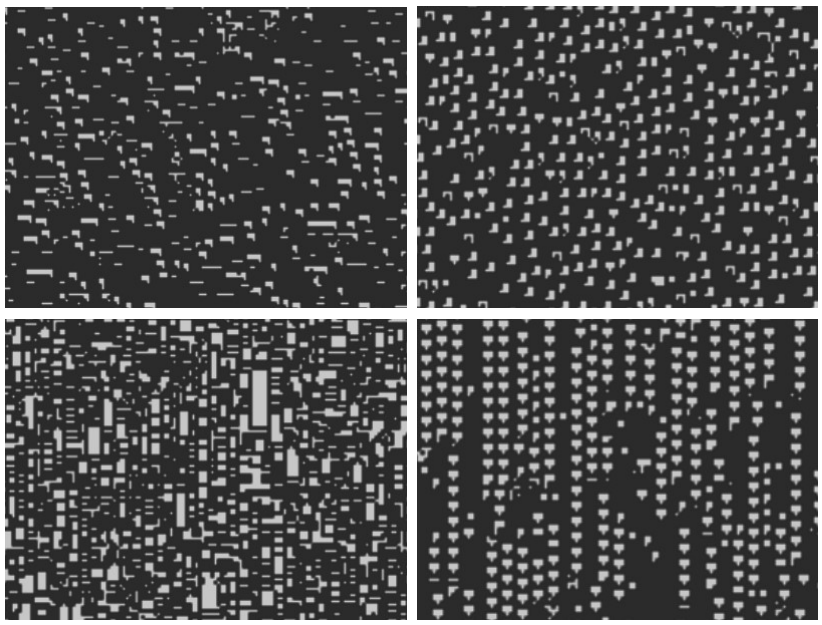

Рисунок 15 – Некоторые клеточные автоматы, полученные для одного паттерна.

Более того, если рассмотреть гены клеточных автоматов из разных экспериментов (№1935 и №2021), но дающих почти неразличимые внешне результаты, как изображено на рис. 16, то будет понятно, что и гены, и опорные биты в этих автоматах тоже не будут совпадать. Из этого можно сделать вывод о том, что данным методом невозможно ни экспериментально, ни аналитически определить диапазон поиска клеточных автоматов.

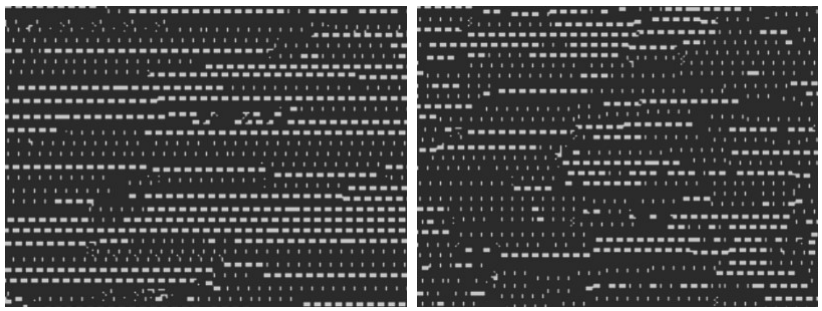

Рисунок 16 – Внешне схожие клеточные автоматы с разных экспериментов.

Все дальнейшие оптимизации нахождения паттерна будут проводиться за счет выбора «правильной» конфигурации эксперимента, чему и посвящены следующие разделы.

3.3 Описание проведенных экспериментов

В настоящей работе было проведено 1306 экспериментов над различными паттернами. Каждый паттерн дополняется похожими версиями себя, которые строятся отражением по вертикали или горизонтали исходного целевого изображения или сменой цветов на противоположные (черный на белый и наоборот).

Далее следует описание количества экспериментов, проведенных для некоторых целевых изображений. Визуальное представление паттернов изображено на рисунке 17.

- 222 над «крестом 5×5 с одной ошибкой»;
- 43 над «елкой 5×5 с одной ошибкой»;
- 346 над «елкой 5×5 с двумя ошибками»;
- 215 над «квадратом 3×3 и рамкой 1 пиксель противоположного цвета с двумя ошибками»;
- 22 над «повернутой в 4 стороны елкой 5×5 с одной ошибкой»;
- 70 над «цифрой ’5’ в рамке, 7×5 с пятью ошибками»;
- 84 над «треугольник 7×5 в двух поворотах с шестью ошибками».

Можно заметить, что большая часть паттернов имеет размер 5×5. Такой выбор объясняется тем, что клеточные автоматы для паттернов такого размера зачастую ищутся примерно за 300 этапов эволюции, что позволяет провести значительное количество экспериментов за относительно короткий промежуток времени и одновременно заметить разницу между различными параметрами генетического алгоритма. Так как в дальнейшем выводы о наилучшей конфигурации будут браться из статистических данных, необходимо провести большое количество запусков программы.

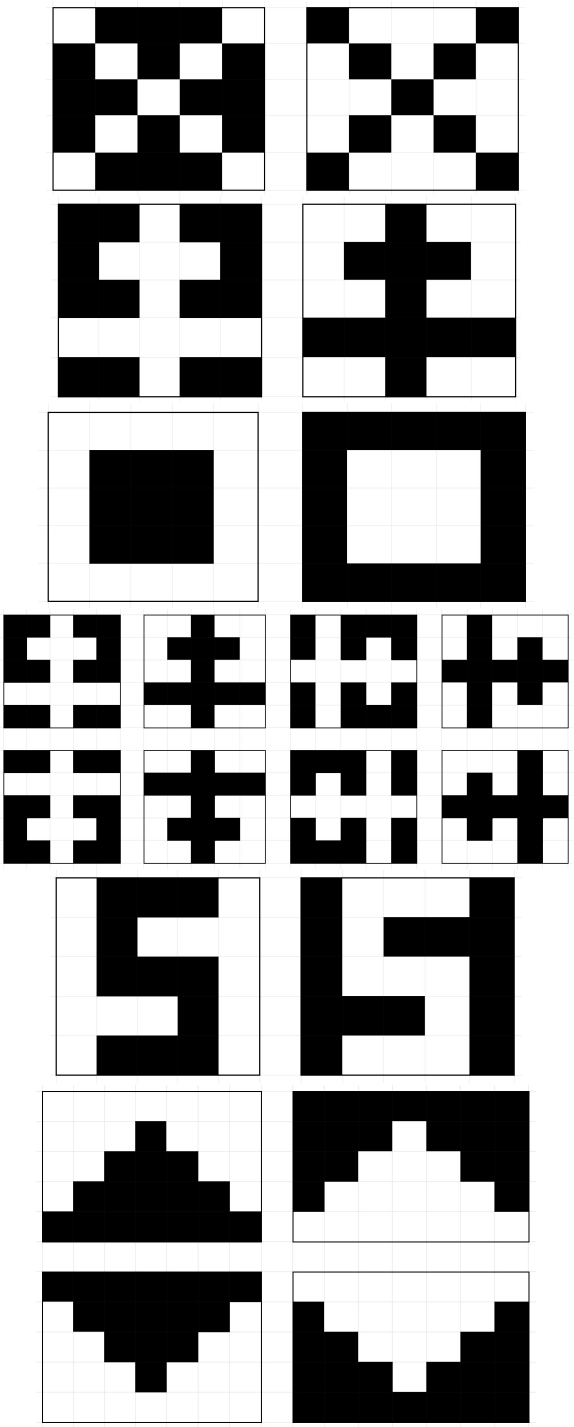

Рисунок 17 – Паттерны: «крест 5×5», «елка 5×5», «квадрат 3×3 и рамка 1 пиксель противоположного цвета», «крест 5×5 и рамка 1 пиксель противоположного цвета», «повернутая в 4 стороны елка 5×5», «цифра ’5’ размера 5×5», «треугольник 7×5 в двух поворотах».

4 Анализ экспериментов

Данный раздел посвящен анализу данных, полученных после проведения экспериментов. Для каждого целевого изображения проводились эксперименты с различными настройками генетического алгоритма: количество клеточных автоматов, вероятность мутации, число мутирующих битов. Основываясь на полученных результатах можно сделать вывод о наилучших параметрах генетического алгоритма, при которых происходит более быстрое воспроизведение целевого изображения.

4.1 Обработка и визуализация данных

Для обработки и визуализации полученных данных создана программа, написанная на языке `Python` с использованием библиотеки `Matplotlib`. Программный код с комментариями написан в среде `Jupyter Notebook`, позволяющей создать наглядный и задокументированный программный код с сохраненным результатом выполнения. Файл в формате `.ipynb` хранится в репозитории. Он расположен в корне папки `Statistics`.

Данный «ноутбук» собирает и обрабатывает все файлы со статистикой по выбранному набору файлов. Интерфейсом между данными и программистом выступает функция `analyze`, которая принимает на вход список файлов и любое количество пар `(функция,номер строки)`. К каждой выбранной строке в блоке вывода по эксперименту будет применена заданная функция.

```
def analyze(statistics_files, *funcs_indices):
    for file in statistics_files:
        f = open(file, 'r')
        cur_index = 0
        for line in f:
            for f_ind in funcs_indices:
                if (cur_index == f_ind[1]):
                    f_ind[0](line)
            cur_index += 1
            cur_index = cur_index % 10
        f.close()
```

В качестве примера приведем функцию, которая совершит подсчет времени, затраченного на эксперименты. Для этого подготовлены две вспомога-

тельные функции. Первая, по входной строке определяет, для какого паттерна проведен эксперимент. Вторая функция получает затраченное на него время в секундах и добавляет к соответствующему аккумулятору (в словарь, ключом к которому является название паттерна). Вызовем функцию `analyze`, передав в параметры список файлов с данными по экспериментам и вспомогательные функции вместе с номерами строк. Полученный фрагмент кода выглядит следующим образом:

```
def calculate_time(statistics_files, patterns):
    time_accum = dict()
    for pattern in patterns:
        time_accum[pattern.split('Stats-')[1].split('.txt')[0]] = 0

    current_pattern = ""

    def get_current_pattern(line):
        nonlocal current_pattern
        current_pattern = line.split('for')[1]
                .split('after')[0].strip()

    def success_time(line):
        nonlocal time_accum
        time_acc = int(line.split('.')[0].split(':')[1]
                .split(',')[0])
        if (time_acc > 0):
            time_accum[current_pattern] += int(line.split('.')[0]
                    .split(':')[1].split(',')[0])

    analyze(statistics_files,
            (get_current_pattern, 1), (success_time, 2))

    for entry in time_accum.keys():
        time_accum[entry] /= 3600

    return time_accum
```

Наконец, сделаем визуализацию полученных результатов в графиках 18.

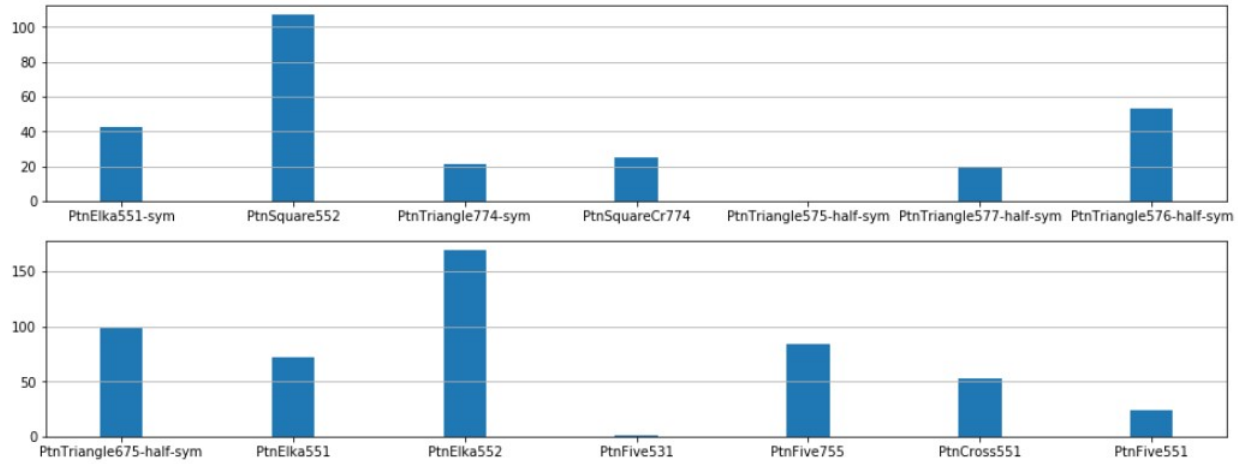

Рисунок 18 – Визуализация времени работы. По оси X — названия паттернов, по Y — время в часах.

Аналогичным образом посмотрим на количество проведенных экспериментов 19.

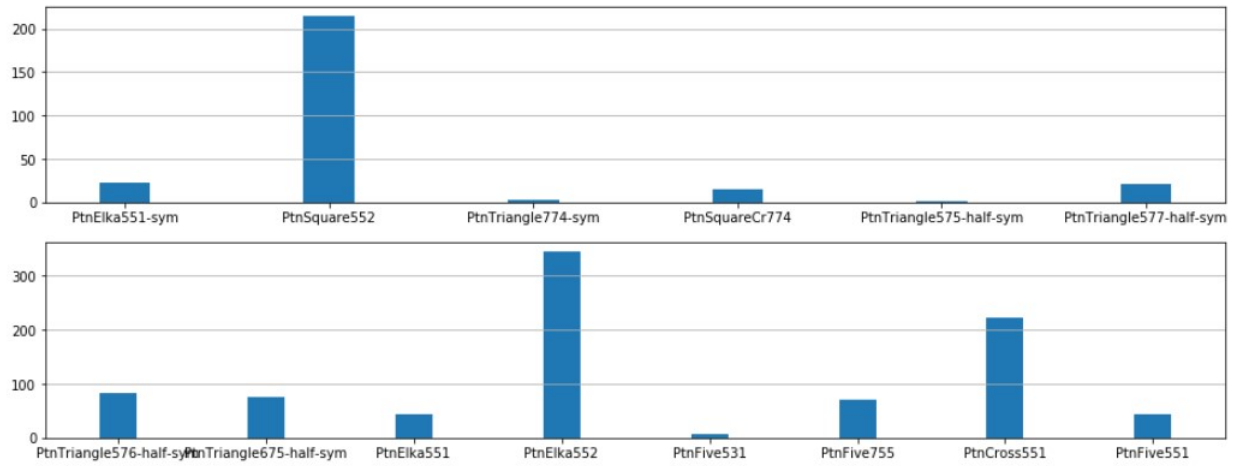

Рисунок 19 – Визуализация количества проведенных экспериментов. По оси X — названия паттернов, по Y — количество.

4.2 Паттерн-зависимые результаты эксперимента

Прежде чем начать сравнивать различные конфигурации генетического алгоритма, необходимо понять, как отличаются попытки найти различные целевые изображения. Основное внимание будет уделяться следующим характеристикам паттернов: размерность (первые 2 числа в названии), количество допустимых ошибок (последнее число в названии), симметричность. Посмотрим на среднее время выполнения экспериментов 20.

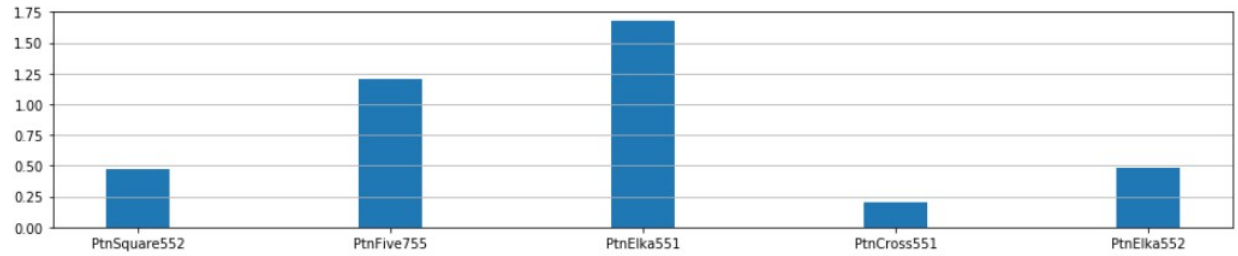

Рисунок 20 – Визуализация среднего количества времени на 1 эксперимент для некоторых паттернов. По оси X — названия паттернов, по Y — время в часах.

Не удивительно, что чем больше ошибок допускается при сравнении с целевыми изображениями, тем больше диапазон возможных клеточных автоматов, воспроизводящих заданные паттерны. Тем самым, время поиска любой подходящей комбинации битов в генах снижается. В случае с паттерном «елка 5×5» разница между средним временем работы в случае двух и одной ошибки — 3.5 раза.

Интересными оказались наблюдения, связанные с паттернами одинакового размера. Можно заметить, что среднее время работы паттерна «крест 5×5» и «елка 5×5» сильно различаются. Более того, в первом допускается всего 1 ошибка, и даже с учетом этого он ищется примерно в два раза быстрее чем «елка 5×5» с двумя допустимыми ошибками. На основании полученных данных по всем экспериментам можно сделать предположение, что на скорость нахождения определенного паттерна влияет форма изображения. Паттерны, внешне похожие на линии и простые фигуры, ищутся быстрее чем сложные изображения.

В связи с высокой долей непредсказуемости результатов по разным паттернам, будем сравнивать друг с другом лишь эксперименты, проведенные над одинаковыми целевыми изображениями. Для минимизации вероятности того, что с определенным паттерном «повезло», проведем эксперименты с разными параметрами генетического алгоритма над несколькими целевыми изображениями.

4.3 Оптимальные параметры эксперимента

Важной составляющей для достижения наиболее быстрого нахождения клеточного автомата, воспроизводящего заданные целевые изображения, является подбор параметров для генетического алгоритма. В настоящей работе рассматриваются следующие основные параметры генетического алгоритма:

процент мутации и максимальное количество бит, которые могут быть мутированы (выбирается случайное число от 1 до максимального).

Для проведения экспериментов были выбраны паттерны, приближенный результат для которых находится меньше чем за 500 итераций (полчаса реального времени). Это позволит провести большое количество экспериментов, тем самым проверив множество комбинаций процента мутации и количества бит. Полученные результаты для трех целевых изображений: «квадрат 3×3 с рамкой 1 пиксель», «елка 5×5», «крест 5×5» и «треугольник 7×5 в двух поворотах» 3.3.

Полученные результаты отражены на изображениях 21, 22, 23, 24. Данные эксперименты проводились для различных комбинаций процента мутаций и количества бит в мутации. Из файла с результатами эксперимента (в первом ряду на изображениях) извлечена информации по среднему коэффициенту приспособленности, что является главным критерием соответствия целевому изображению. Также выведем количество проведенных экспериментов, чтобы убедиться, что для всех конфигураций проведено достаточно экспериментов. Во втором ряду изображений расположен график процента успешных/неудачных экспериментов при установленном пороговом значении 25 и среднее количество этапов эволюции от начала до конца эксперимента. Для всех экспериментов в данном разделе выбиралось пороговое значение приспособленности равное 25, а количества экспериментов — 500. Таким образом, эксперимент может закончиться раньше, если пороговое значение было достигнуто.

По результатам эксперимента можно сделать выводы о параметрах генетического алгоритма, дающих большую приспособленность. Вполне ожидаемо, что низкий процент мутации и мутация одного бита не дает достаточного разнообразия особей. Рассматривая высокие вероятности мутации и количество бит, можно заметить, что конфигурация генетического алгоритма, при которой процент мутации равен 25, а количество бит может достигать 8 или 16, дает наилучшую приспособленность среди всех других рассмотренных конфигураций генетических алгоритмов. Дальнейшее увеличение процента мутаций и количества бит не даст увеличения коэффициента приспособленности, потому что дочерние особи будут слишком сильно отличаться от родительских. Снижение процента мутации, наоборот, не создает достаточного разнообразия особей.

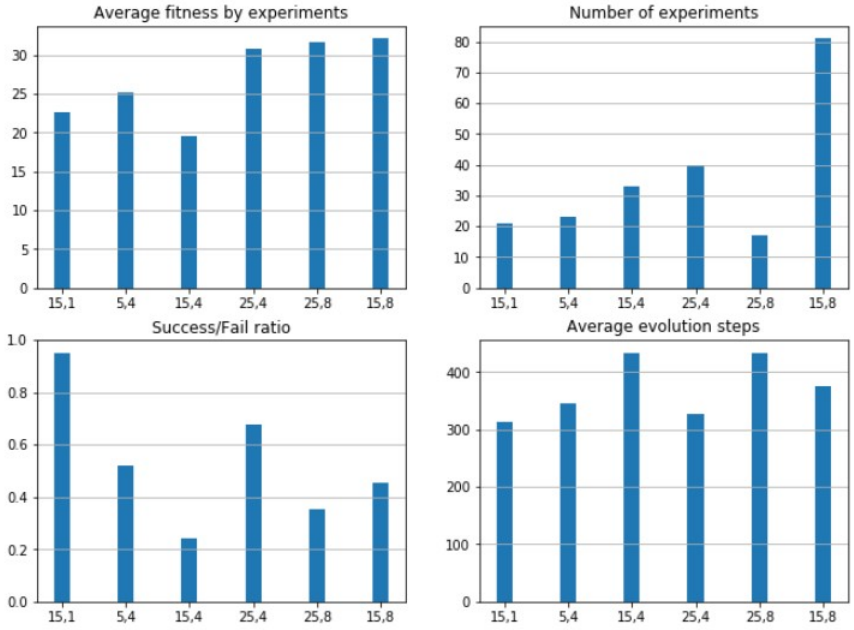

Рисунок 21 – Результаты экспериментов для паттерна «квадрат 3×3 с рамкой 1 пиксель противоположного цвета» с 2 допустимыми ошибками. По оси X — конфигурация генетического алгоритма в виде пары чисел через запятую: вероятность мутации в процентах и максимальное количество бит для мутации.

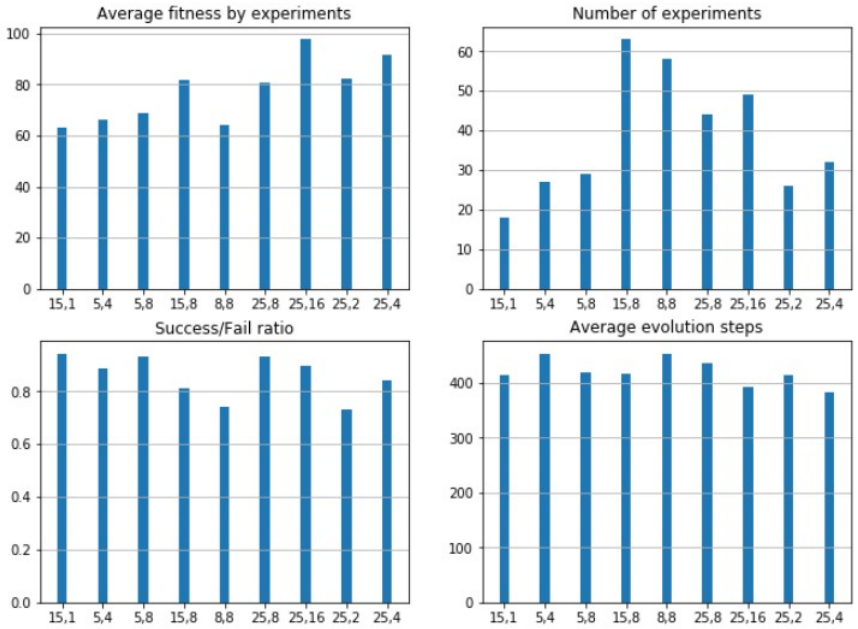

Рисунок 22 – Результаты экспериментов для паттерна «елка 5×5» с 2 допустимыми ошибками. По оси X — конфигурация генетического алгоритма в виде пары чисел через запятую: вероятность мутации в процентах и максимальное количество бит для мутации.

Данные закономерности прослеживаются для различных паттернов: разной размерности, сложности фигуры, количества допустимых ошибок. Поэтому можно быть уверенным, что данные результаты применимы к любому паттерну, и скорее всего к некоторым другим способам подсчета приспособ-

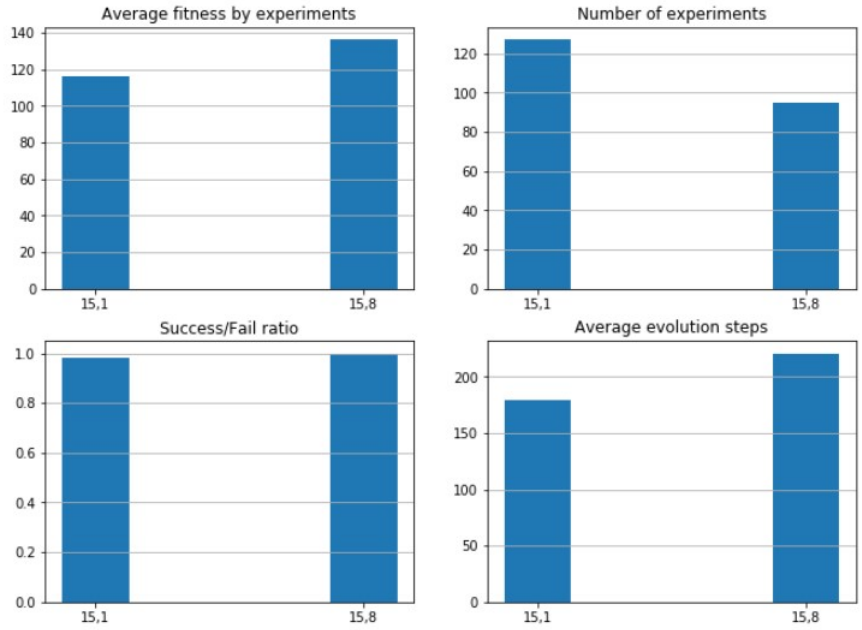

Рисунок 23 – Результаты экспериментов для паттерна «крест 5×5» с 1 допустимой ошибкой. По оси X — конфигурация генетического алгоритма в виде пары чисел через запятую: вероятность мутации в процентах и максимальное количество бит для мутации.

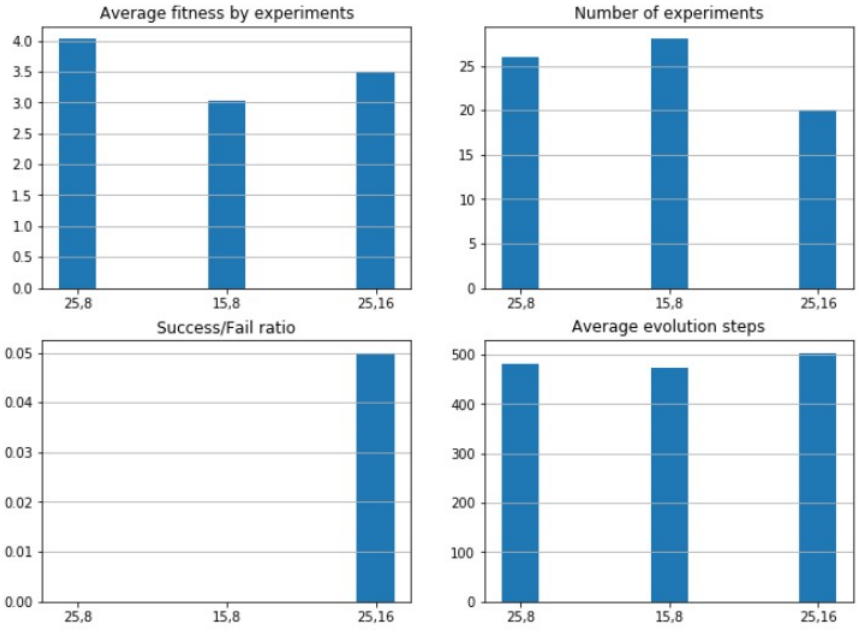

Рисунок 24 – Результаты экспериментов для паттерна «треугольник 7×5 в двух поворотах» с 6 допустимыми ошибками. По оси X — конфигурация генетического алгоритма в виде пары чисел через запятую: вероятность мутации в процентах и максимальное количество бит для мутации.

ленности, так как выбор данных параметров генетического алгоритма контролирует только разнообразие популяции, отвечая за то чтобы производные автоматы, созданные в процессе скрещивания, были одновременно и похожими на родителей, и совмещали себе новые черты.

Результаты для некоторых других паттернов показаны на рисунках 25, 26.

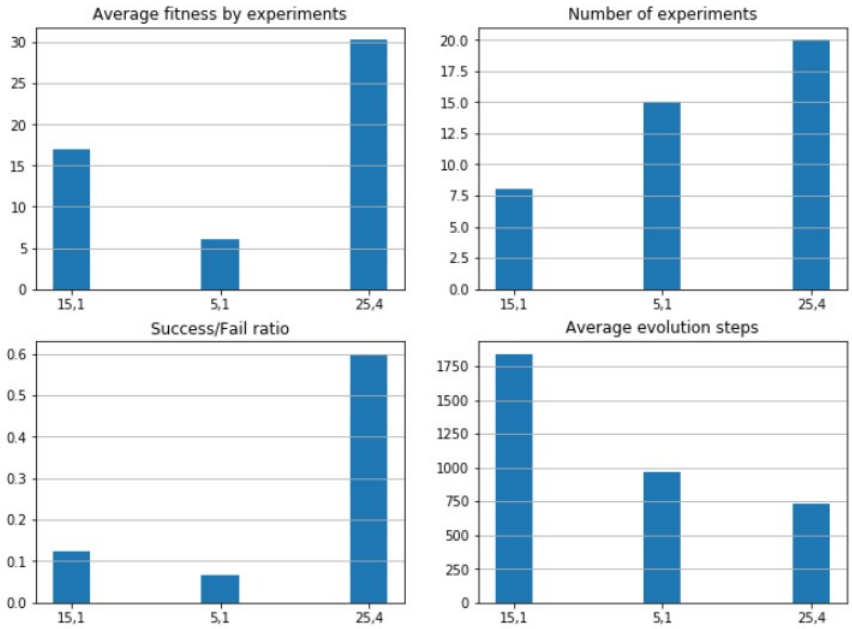

Рисунок 25 – Результаты экспериментов для паттерна «елка 5×5» с 1 допустимой ошибкой. По оси X — конфигурация генетического алгоритма в виде пары чисел через запятую: вероятность мутации в процентах и максимальное количество бит для мутации.

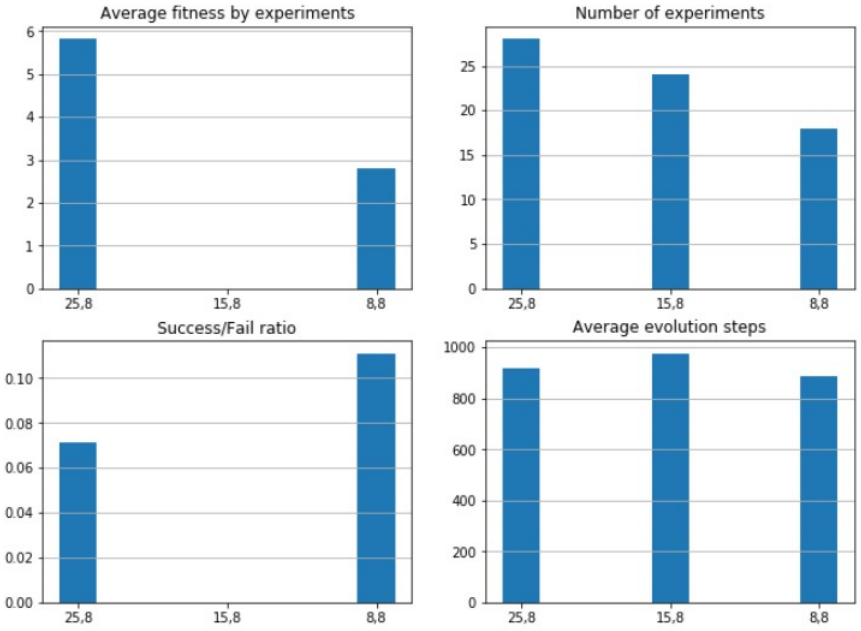

Рисунок 26 – Результаты экспериментов для паттерна «цифра ’5’ размера 7×5» с 5 допустимыми ошибками. По оси X — конфигурация генетического алгоритма в виде пары чисел через запятую: вероятность мутации в процентах и максимальное количество бит для мутации.

4.4 Дальнейшие исследования

В предыдущих разделах рассматривается повышение скорости нахождения приближенного паттерна с помощью конфигурации генетического алгоритма, но есть две нерешенные проблемы.

1. Некоторые паттерны даже относительно небольших размеров (5×5, например) требуют больших вычислительных ресурсов, чтобы их найти. Особенно сложными являются паттерны, в которых отсутствует симметричность или имеется сильный дисбаланс в распределении черных и белых ячеек (особенно если размер паттерна достаточно высокий).
2. Не все полученные клеточные автоматы воспроизводят изображение, внешне похожее на исходное.

В качестве иллюстрации второй проблемы, возьмем паттерн «цифра ’пять’ размера 5×5» (изображение 27). Если позволить клеточному автомату допускать две ошибки при сравнении с целевым изображением, то с высокой вероятностью найдется более простая фигура, представляющая собой параллельные линии. Данная закономерность была выявлена эмпирическим путем: паттерн, состоящий из параллельных линий, находится намного быстрее чем более сложные изображения.

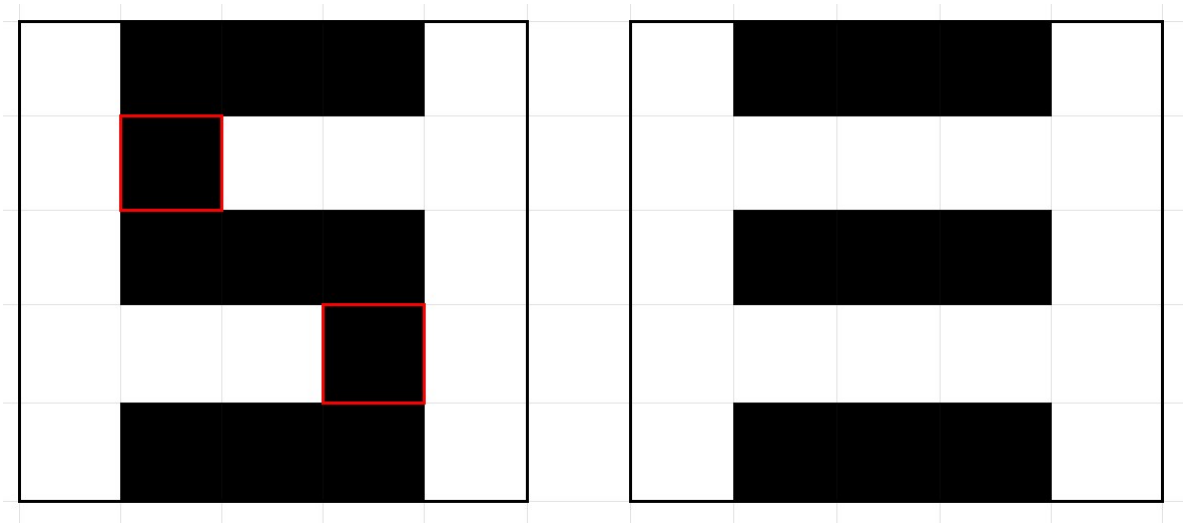

Рисунок 27 – Слева — паттерн «цифра ’пять’ размера 5×5». Возможная вариация с двумя ошибками показана изображением справа.

Далее изложены предположения о том, какие изменения алгоритма теоретически могли бы улучшить поиск клеточных автоматов. Подробные исследования на данную тему не проводились ввиду значительного увеличения объема работы и времени экспериментов.

Альтернативное множество начальных состояний. Если вместо случайного заполнения всех ячеек клеточного автомата заполнить лишь одну клетку (например, в центре сетке), то некоторые клеточные автоматы смогут производить более упорядоченные изображения на первых итерациях обновления клеточного автомата.

Клеточные автоматы высшего порядка. Клеточному автомату можно добавить память, другими словами, возможность запоминать свои состояния на предыдущих итерациях. Такие клеточные автоматы называются клеточными автоматами высшего порядка. При переходе ячеек в новые состояния учитываются значение в них на предыдущих итерациях. Добавление памяти увеличивает количество всех возможных клеточных автоматов и одновременно с этим усложняет поиск клеточного автомата, удовлетворяющего условию.

Другое множество правил перехода. В данной работе правила перехода строились как комбинация состояний ячеек в окрестности Мура порядка 2. Как минимум еще один возможный вариант: окрестность фон Неймана, как изображено на рисунке 28. Таким образом, количество возможных правил перехода изменится, и вместе с этим и множество клеточных автоматов.

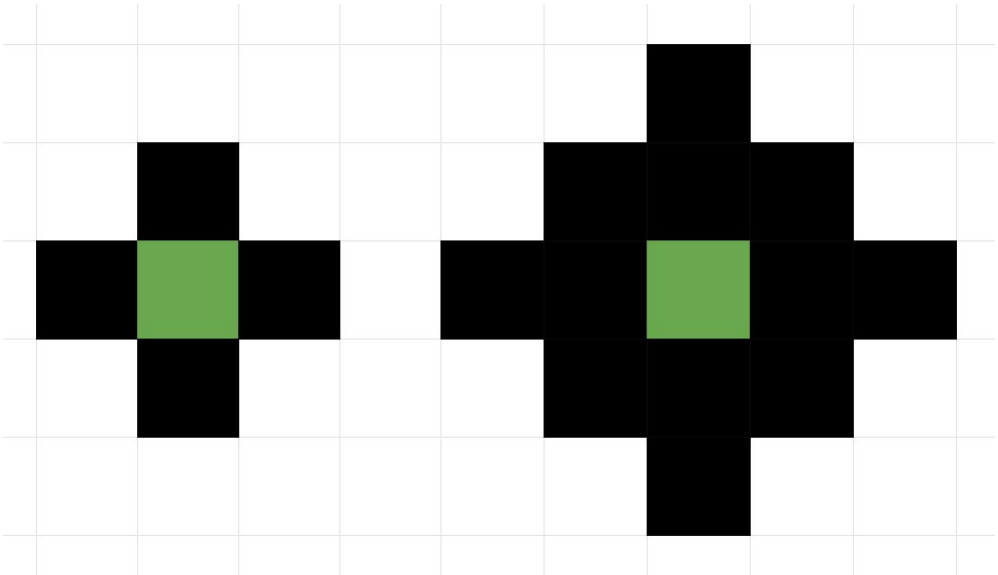

Рисунок 28 – Окрестность фон Неймана порядка 1 и 2 для некоторой ячейки (изображена зеленым цветом).

Измененные правила подсчета приспособленности. Для особо сложных паттернов, для которых сложно заранее определить количество возможных ошибок, можно подсчитывать приспособленность как максимальное количество клеток по всей сетке, совпавших с целевым изображением. Также при

сильном различии количества белых и черных клеток в целевом изображении, можно добавить дополнительное условие, чтобы разница между количеством клеток разных цветов не превышала определенного порога, и если порог превышен, то приспособленность считать равной 0. В таком случае есть риск, что критерий приспособленности будет слишком «жестким», однако при этом исключается возможность получения в результате эксперимента неправильного паттерна.

Защита от мутации особей с наивысшей приспособленностью. В текущей реализации возможна ситуация, что особь с наивысшей приспособленностью после шага мутации будет изменена, что приведет к потере последовательности битов в гене, дающих хороший результат. С другой стороны, зачастую одна особь, дающая высокую приспособленность на каком-то шаге, не будет давать высокие показатели на последующих. Причина этого — случайная начальная заполненность ячеек клеточного автомата.

Перебор случайных генов при низкой приспособленности. Даже когда приспособленность особей близка к нулю, генетический алгоритм скрещивает биты генов, создавая новые гены. Возможно, будет более эффективным брать новые случайные гены вместо скрещивания.

ЗАКЛЮЧЕНИЕ

В результате дипломной работы было создано полноценное приложение для нахождения и визуализации двумерных клеточных автоматов первого порядка, воспроизводящих заданное целевое изображение. Данное приложение применимо для проведения экспериментов по подбору наилучших параметров для быстрого и эффективного (по значению приспособленности) поиска заданных клеточных автоматов. С помощью приложения были проведены эксперименты для различных параметров генетического алгоритма. На основе обработанных данных были определены оптимальные параметры генетического алгоритма. Оптимизации также проведены в коде программы, что привело к сокращению времени исполнения алгоритмов в несколько раз. Полученные результаты можно использовать и для других клеточных автоматов: высшего порядка, с большим числом состояний, с другим множеством начальных состояний.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 [Клеточные автоматы - реализация и эксперименты]. — URL: https://www.osp.ru/pcworld/2003/08/166226/ (Дата обращения 27.05.2020). Загл. с экр. Яз. рус.

2 *Rosin, P.* Edge Detection Using Cellular Automata / P. Rosin, X. Sun. — 2014. — Pp. 85–103.

3 *Zhu, B.-P.* Public-key cryptosystem based on cellular automata / B.-P. Zhu, L. Zhou, F.-Y. Liu. — 10 2007. — Vol. 31. — Pp. 612–616.

4 [Conway's Game of Life]. — URL: https://www.conwaylife.com/ (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

5 *Ladd, A.* An application of lattice-gas cellular automata to the study of brownian motion / A. Ladd, D. Frenkel, M. Colvin. — 01 1988. — Vol. 60. — Pp. 975–978.

6 *Oono, Y.* Discrete model of chemical turbulence / Y. Oono, M. Kohmoto // *Phys. Rev. Lett.* — Dec 1985. — Vol. 55. — Pp. 2927–2931. https://link.aps.org/doi/10.1103/PhysRevLett.55.2927.

7 *Sander, L.* Fractal growth processes / L. Sander // *Nature*. — 08 1986. — Vol. 322. — Pp. 789–793.

8 [Game of Life]. — URL: https://mathworld.wolfram.com/GameofLife.html (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

9 *Chavoya, A.* Using a genetic algorithm to evolve cellular automata for 2d/3d computational development. — Vol. 1. — 01 2006. — Pp. 231–232.

10 *Mordvintsev, A.* Growing neural cellular automata / A. Mordvintsev, E. Randazzo, E. Niklasson, M. Levin // *Distill*. — 2020. — https://distill.pub/2020/growing-ca.

11 [Moore Neighborhood]. — URL: https://mathworld.wolfram.com/MooreNeighborhood.html (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

12 *Packard N.H., W. S.* Two-dimensional cellular automata / W. S. Packard, N.H. // *Journal of Statistical Physics*. — 1985. — Vol. 38. — Pp. 901–946.

13 [Fundamentals of Computer Programming]. — Pp. 150–151. — URL: https://books.google.ru/books?id=xYgCAQAAQBAJ&pg=PA150&

lpg=PA150&dq=bit+operations+performance+c%23&source=bl&ots=FPdhyQlS5h&sig=ACfU3U3dz1jBOUOWD_0553muTAbXAM2jFw&hl=ru&sa=X&ved=2ahUKEwiH8M_WntnpAhXMyKYKHbohC3QQ6AEwDXoECAoQAQ (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

14 *Dennunzio, A.* On the dynamical behaviour of linear higher-order cellular automata and its decidability / A. Dennunzio, E. Formenti, L. Manzoni, L. Margara, A. Porreca // *Information Sciences*. — 02 2019. — Vol. 486.

15 [Git - What is Git?]. — URL: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

16 [FLOPs per Cycle for CPUs, GPUs and Xeon Phis]. — URL: https://www.karlrupp.net/2016/08/flops-per-cycle-for-cpus-gpus-and-xeon-phis/ (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

17 [Rendering Pipeline Overview]. — URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

18 [Global Optimization Algorithms]. — URL: http://www.it-weise.de/projects/book.pdf (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

19 *v. Werkhoven, B.* Performance models for cpu-gpu data transfers // 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. — 2014. — Pp. 11–20.

20 [Bit Twiddling Hacks]. — URL: https://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel (Дата обращения 27.05.2020). Загл. с экр. Яз. англ.

ПРИЛОЖЕНИЕ А

Код оптимизированной функции переходов

```
private void UpdateCA(byte[] CAField, int ind)
{
    long size2D = screenSizeInPixels * screenSizeInPixels;
    int signal = 0;

    for (int i = screenSizeInPixels - 2; i < screenSizeInPixels; i++)
    {
        var iPix = i * screenSizeInPixels;
        for (int j = 0; j < screenSizeInPixels; j++)
        {
            screenSignals[iPix + j] = 0;
        }
    }
    for (short i = 0; i < screenSizeInPixels - 2; i++)
    {
        var iPix = i * screenSizeInPixels;
        var iPixm1 = (iPix + size2D - screenSizeInPixels) % size2D;
        var iPixm2 = (iPix + size2D - screenSizeInPixels - screenSizeInPixels) % size2D;
        signal = CAField[iPix + (screenSizeInPixels) - 1] * 2 + CAField[iPix];
        for (short j = 1; j < screenSizeInPixels; j++)
        {
            signal = (signal << 1) % 8 + CAField[iPix + j];
            var jm1 = j - 1;
            screenSignals[iPixm2 + jm1] += signal << 6;
            screenSignals[iPixm1 + jm1] += signal << 3;
            screenSignals[iPix + jm1] = signal;
        }
        signal = (signal << 1) % 8 + CAField[iPix];
        screenSignals[iPixm2 + screenSizeInPixels - 1] += signal << 6;
        screenSignals[iPixm1 + screenSizeInPixels - 1] += signal << 3;
        screenSignals[iPix + screenSizeInPixels - 1] = signal;
    }

    for (int i = screenSizeInPixels - 2; i < screenSizeInPixels; i++)
    {
        var iPix = i * screenSizeInPixels;
        var iPixm1 = (iPix + size2D - screenSizeInPixels) % size2D;
        var iPixm2 = (iPix + size2D - screenSizeInPixels - screenSizeInPixels) % size2D;
        signal = CAField[iPix + (screenSizeInPixels) - 1] * 2 + CAField[iPix];
```

```
            for (short j = 1; j < screenSizeInPixels; j++)
            {
                signal = (signal << 1) % 8 + CAField[iPix + j];
                var jm1 = j - 1;
                screenSignals[iPixm2 + jm1] += signal << 6;
                screenSignals[iPixm1 + jm1] += signal << 3;
                screenSignals[iPix + jm1] += signal;
            }
            signal = (signal << 1) % 8 + CAField[iPix];
            screenSignals[iPixm2 + screenSizeInPixels - 1] += signal << 6;
            screenSignals[iPixm1 + screenSizeInPixels - 1] += signal << 3;
            screenSignals[iPix + screenSizeInPixels - 1] = signal;
        }

        for (int i = 0; i < size2D; i++)
        {
            CAField[i] = (byte)allRules[ind][screenSignals[(i - screenSizeInPixels + size2D) %
                size2D]];
        }
    }
```

ПРИЛОЖЕНИЕ Б

Код функции подсчета приспособленности

```
private float CalculateFitness(byte[] texturePixels, int texW, int texH, Pattern[] patterns)
{
  float fitness = 0;

  int textureWidth = texW; int textureHeight = texH;

  var patternHeight = patterns[0].patternSizeY;
  var patternWidth = patterns[0].patternSizeX;
  var patternErrors = patterns[0].patternErrors;
  var patternsCount = patterns.Length;
  int[] currentErrors = new int[patternsCount];

  for (short i = 0; i < textureHeight; i++)
  {
    for (short j = 0; j < textureWidth; j++)
    {
      int cornerPixel = j + i * textureWidth;
      for (int p = 0; p < patternsCount; p++)
```

```
        {
          currentErrors[p] = 0;
        }

        for (byte ir = 0; ir < patternHeight; ir++)
        {
          for (byte jr = 0; jr < patternWidth; jr++)
          {
            int pixelIndex = (cornerPixel + ir * textureWidth + jr)
              % (textureWidth * textureHeight);
            for (int p = 0; p < patternsCount; p++)
            {
              currentErrors[p] += texturePixels[pixelIndex] ==
                patterns[p].pattern[ir * patternWidth + jr] ? 0 : 1;
            }
          }
        }
        int minErrors = patternErrors + 1;
        for (int p = 0; p < patternsCount; p++)
        {
          minErrors = currentErrors[p] < minErrors ? currentErrors[p] : minErrors;
        }

        fitness += minErrors <= patternErrors ?
          (1 + patternErrors - minErrors) * 1f / (patternErrors + 1) : 0;
      }
    }
    return fitness * 100 / (textureWidth * textureHeight);
}
```

ПРИЛОЖЕНИЕ В

Оптимизированный подсчет паттерна

```
private float CalculateFitnessOptimised(byte[] texturePixels, int texW, int texH, Pattern[]
    patterns)
{
 float fitness = 0;

 int textureWidth = texW; int textureHeight = texH;
 var tex2D = texW * texH;
 var patternHeight = patterns[0].patternSizeY;
 var patternWidth = patterns[0].patternSizeX;
```

```
var patternErrors = patterns[0].patternErrors;
var patternRule = patterns[0].pattern;
int[] curErr = new int[patterns.Length];
int patternCycle = (1 << patternHeight);

int[] patternLines = new int[patterns[0].patternSizeX];
int newPatternLine = 0;
for (int i = 0; i < patternHeight; i++)
{
  newPatternLine = 0;
  for (int j = 0; j < patternWidth; j++)
  {
      newPatternLine = (newPatternLine << 1) + patterns[0].pattern[i * patternWidth + j];
  }
  patternLines[i] = newPatternLine;
}

int qOffset = 0;

int newScreenLine = 0;
for (qOffset = 0; qOffset < patternHeight - 1; qOffset++)
{
  newScreenLine = 0;
  for (int j = 0; j < patternWidth; j++)
  {
    newScreenLine = (newScreenLine << 1) + texturePixels[qOffset * patternWidth + j];
  }
  screenLines[qOffset] = newScreenLine;
}

int newLine = 0, iPix = 0, newPixInd = 0, cornerPixel = 0;
int screenLine = 0, newErr = 0, nextPixInd = 0;
for (short i = 0; i < textureHeight; i++)
{
  newLine = 0;
  iPix = ((i + patternHeight - 1) % textureHeight) * textureWidth;
  for (int j = 0; j < patternWidth; j++)
  {
    newPixInd = (iPix + j);
    newLine = (newLine << 1) + texturePixels[newPixInd];
  }
```

```
qOffset = (qOffset + 1) % patternHeight;
screenLines[qOffset] = (newLine);

for (short j = 0; j < textureWidth; j++)
{
  cornerPixel = j + i * textureWidth;
  for (byte p = 0; p < patterns.Length; p++)
  {
      curErr[p] = 0;
  }
  int minErrors = patternErrors + 1;

  for (byte k = 0; k < patternHeight; k++)
  {
    int ind = (qOffset + k) % patternHeight;
    screenLine = screenLines[ind];
    newErr = patternLines[k] ^ screenLine;

    for (byte p = 0; p < patterns.Length; p++)
    {
      newErr = newErr - ((newErr >> 1) & 0x55555555);
      newErr = (newErr & 0x33333333) + ((newErr >> 2) & 0x33333333);
      curErr[p] += (((newErr + (newErr >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
    }

    screenLine <<= 1;
    if (screenLine >= patternCycle) screenLine -= patternCycle;
    nextPixInd = (((i + k) % textureHeight) * textureWidth) + (j + patternWidth) %
        textureWidth;
    screenLine += texturePixels[nextPixInd];

    screenLines[(qOffset + k) % patternHeight] = (screenLine);
  }

  for (byte p = 0; p < patterns.Length; p++)
  {
      if (curErr[p] < minErrors) minErrors = curErr[p];
  }
  fitness += minErrors <= patternErrors ?
   (1 + patternErrors - minErrors) * 1f / (patternErrors + 1) : 0;
}
```

```
    }

    return fitness * patternWidth * patternHeight * 100 / (textureWidth * textureHeight);
}
```

ПРИЛОЖЕНИЕ Г

Файловая структура проекта

На приложенном диске можно ознакомиться со следующими файлами:

Папка `Assets\Scripts` — код приложения, моделирующего клеточные автоматы;

Папка `Assets\Scenes` — сцены для проекта в среде разработке `Unity`;

Файл `Statistics\Statistics.ipynb` — `IPython Notebook` для анализа данных, полученных после эксперимента;

Папка `Statistics\SimulationData` — данные, полученные в ходе экспериментов: результаты, гены и другие данные по каждому эксперименту.

Все остальные папки и файлы вспомогательные, нужны для корректной работы приложений.

ПРИЛОЖЕНИЕ Д
Статистика по экспериментам

Таблица 1 – Объединенные данные по экспериментам в текстовом формате

Название паттерна	Макс. ошибок в паттерне	Процент мутации	Макс. бит мутаций (до)	Кол-во запусков	Средняя приспособленность
Square55 «Квадрат 3×3 с рамкой 1 пиксель»	2	5	4	23	25.1729
		15	1	21	22.5189
		15	4	33	19.5767
		15	8	81	32.0888
		25	4	40	30.8100
		25	8	17	31.6848
Elka55 «Елка размера 5×5»	2	5	4	27	66.4780
		5	8	29	68.8953
		8	8	58	64.1287
		15	1	18	63.0735
		15	8	63	82.0487
		25	2	26	82.1514
		25	4	32	91.6327
		25	8	44	80.5325
		25	16	50	98.2182
	1	5	1	15	5.9828
		15	1	8	16.9715
		25	4	20	30.2779
Cross55 «Крест 5×5»	1	15	1	127	116.1352
		15	8	100	134.2668
Triangle57 «Треугольник 5×7»	6	15	8	28	3.0381
		25	8	26	4.0253
		25	16	30	4.5005
Five75 «Цифра ’5’ размера 7×5»	5	8	8	18	2.8075
		15	8	24	0.0001
		25	8	28	5.8196