

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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

«Распределенная среда моделирования эволюции»

Студент ИУ9-52

Ковальский А.С.

(ф.и.о.)

_____ (подпись, дата)

Руководитель курсового проекта

Коновалов А.В.

(ф.и.о.)

_____ (подпись, дата)

2018г.

Оглавление

ВВЕДЕНИЕ.....	3
1. РАЗРАБОТКА МОДЕЛИ ГЕНЕТИЧЕСКОГО КОДА.....	4
1.1 Основные принципы развития многоклеточных организмов.....	4
1.2 Формирование градиентов сигнальных веществ.....	5
1.3 Выбранная модель.....	6
2. РЕАЛИЗАЦИЯ МОДЕЛИ ГЕНЕТИЧЕСКОГО КОДА.....	11
2.1 Инициализация серверного и клиентских приложений.....	11
2.2 Протокол взаимодействия сервера с клиентами.....	12
2.3 Реализация модели.....	13
2.4 Проблемы реализации и исправления.....	19
2.5 Ассемблирование генома.....	20
3. ТЕСТИРОВАНИЕ.....	22
3.1 Фитнес функция.....	22
3.2 Тесты.....	22
3.2.1. Тестирование эмбриогенеза.....	23
3.2.2 Тестирование модели эволюции.....	25
ЗАКЛЮЧЕНИЕ.....	29
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	30

ВВЕДЕНИЕ

Эта работа посвящена разработке программы для моделирования процесса эволюции клеточных организмов. Объектом исследования являются эвристические алгоритмы, используемые для решения задач оптимизации и моделирования. Предметом исследования были выбраны генетические алгоритмы моделирования эмбриогенеза существ и процесса естественного отбора взрослых особей, а также их практическая реализация. Практическое значение этих алгоритмов определяется их относительной простотой и наглядностью.

Целью данной работы является разработка приложения, позволяющего моделировать процесс эволюции клеточных организмов при заданных начальных условиях, а именно процесс эмбриогенеза существ в одинаковой среде, имеющих различные структуры ДНК, и процесс естественного отбора взрослых существ, при котором выживают наиболее приспособленные к данной среде организмы. В ходе работы выполняются следующие задачи: изучение литературы по соответствующей тематике, с целью ознакомления с предметной областью, рассмотрение и анализ генетических алгоритмов, их практическая реализация.

1. РАЗРАБОТКА МОДЕЛИ ГЕНЕТИЧЕСКОГО КОДА

1.1 Основные принципы развития многоклеточных организмов

Прежде чем разработать алгоритм для моделирования процесса дифференциации клеток многоклеточного организма, необходимо определиться с основными понятиями предметной области.

Чтобы понять, как смоделировать развитие организма необходимо иметь более четкое представление о том, как именно происходит передача «инструкций», по которым «строится» организм, между клетками.

Определим для этого следующие понятия.

Транскрипция — биосинтез молекул РНК на соответствующих участках ДНК; первый этап реализации генетической информации в клетке, в процессе которого последовательность нуклеотидов ДНК «переписывается» в нуклеотидную последовательность РНК[6].

Экспрессия генов — это процесс, в котором вся наследственная информация от гена преобразуется в функциональный продукт — РНК или белок. Экспрессия генов может регулироваться на всех стадиях процесса[7].

Факторы транскрипции — белки, контролирующие процесс синтеза мРНК на матрице ДНК путём связывания со специфичными участками ДНК.

Факторы транскрипции необходимы для регуляции экспрессии генов. Однако, помимо факторов транскрипции, необходимых для включения экспрессии всех генов, существуют и факторы транскрипции для включения или выключения каждого конкретного гена в определенный момент[6].

МикроРНК — малые некодирующие молекулы РНК длиной 18-25 нуклеотидов (в среднем 22), обнаруженные у растений, животных и некоторых вирусов,

принимающие участие в транскрипционной и посттранскрипционной регуляции экспрессии генов путём РНК-интерференции[3].

РНК-интерференция — процесс подавления экспрессии гена на стадии транскрипции, трансляции, или деградации мРНК при помощи малых молекул РНК.

Оперон — группа генов в хромосоме, включающая структурные гены и ген-оператор. Структурные гены управляют синтезом ферментов, задействованных в образовании клеточного составляющего или в потреблении питательных веществ. Ген-оператор связан с молекулой репрессора и может существовать в открытом или закрытом состоянии. Когда ген-оператор открыт, гены, которые он контролирует, являются функциональными, т.е., производят белки. Взаимодействуя с репрессором, ген-оператор закрывается[7].

Таким образом, можно заметить, что регуляция экспрессии генов, например, путем воздействия на факторы транскрипции или микроРНК, регулирует и весь процесс дифференцирования, морфогенеза и адаптации клеток. В процессе эволюции идет жесткий контроль за местом и количественными характеристиками экспрессии генов в конкретное время, что влияет не только на конкретный ген, но и на весь организм.

Однако до сих пор не до конца ясно, как из клеток одного типа образуются принципиально разные ткани? Подробнее этот вопросы мы рассмотрим на примере эмбриогенеза дрозофил.

1.2 Формирование градиентов сигнальных веществ

Мушка *Drosophila melanogaster* была введена в качестве организма для наблюдения эмбриогенеза еще в 1909 году Томасом Морганом, и поныне она

остается одним из самых любимых модельных организмов для исследователей эмбриогенеза. Рассмотрим ее развитие и мы.

Как и других многоклеточных, развитие дрозофил начинается с дробления яйца и гаструляции. В ходе последовательных митотических делений, выделяется три типа зародышевых клеток — энтодерма, мезодерма и эктодерма. К этому моменту в зарождающемся организме уже распределены (неравномерно) сигнальные вещества, которые после будут влиять на экспрессию генов.

Как же они распределяются? Основа закладывается во время оогенеза, еще задолго до оплодотворения и откладки яиц. Во время созревания ооцита (яйцеклетки) синтезируется большое количество РНК и белков. Развивающийся ооцит имеет градиенты концентраций мРНК. Гены, которые кодируют такие мРНК, оказывают большое влияние на развитие организма. В частности, у мушек-дрозофил эти гены называются *bicoid*, *hunchback*, *nanos* и *caudal*. Они отвечают за формирование осей тела. Когда мРНК генов транслируется в белки, образуются градиенты генов *bicoid* и *nanos*, на переднем и заднем полюсе яйца соответственно. Белок *bicoid* блокирует трансляцию мРНК белка *caudal*, 12 поэтому белки типа *caudal* образуются только на переднем полюсе. Аналогично на переднем полюсе белок *nanos* блокирует белок *hunchback*.

Таким образом, белки *nanos*, *bicoid* и *hunchback* являются факторами транскрипции. Воздействуя на их градиент, ученым удавалось получить мутировавших мушек-дрозофил, не имеющих, например, средней части тела.

1.3 Выбранная модель

При выборе модели реализации сначала необходимо определить, какие свойства объекта значимы и будут промоделированы, а какие — нет. Было

решено абстрагироваться от ряда особенностей реальных организмов, таких как:

- формирование органов, кровеносной системы и тканей в процессе эмбриогенеза существа;
- изменения функций клеток, их неравномерного роста, запрограммированного апоптоза;
- трёхмерности организма, деления клеток на две половинки в трёх измерениях;
- функциональных (не сигнальных) веществ;
- взаимодействия веществ между собой, их распада, ингибирования;
- стохастическое взаимодействие оператора с репрессором;
- разделение ДНК на триплеты, гены, стоп-кодоны и пр.,

а так же абстрагироваться от любых взаимодействий между существами, кроме непосредственно скрещивания и возможного обмена фрагментами ДНК при мутациях.

Таким образом, выбранная модель отражает:

- одинаковые начальные условия для развития существ;
- начальную анизотропию веществ клетки;
- изменение сигнальных веществ под действием генов;
- независимое развитие существ;
- диффузию сигнальных веществ в соседние клетки;
- регуляцию генов.

Руководствуясь этими данными, было решено построить модель следующим образом.

Различные процессы эволюции разделены между серверным приложением и приложением клиентов. Задача серверного приложения

заключается в управлении популяцией существ (создание начальной популяции, генерация последующих поколений, мутации, естественный отбор).

Серверное приложение работает только с цепочками ДНК, при этом единственная информация которой оно владеет о них, это длина одного гена – 16 бит. Таким образом серверное приложение работает с цепочками 16-битных слов.

В процессе работы серверное приложение случайным образом генерирует цепочки ДНК для первого поколения существ. После этого на основе сгенерированных данных скрещиванием существ создается следующее поколение. Для внесения разнообразия в генофонд с некоей вероятностью каждая из созданных цепочек может мутировать одним из нескольких способов:

- Две цепочки ДНК могут обмениваться своими фрагментами;
- Случайный фрагмент цепочки ДНК может скопироваться или сдвинуться в новое место цепочки, а так же дублироваться или обратиться;
- В цепочке ДНК может появиться новый фрагмент, либо из цепочки может удалиться случайный фрагмент;
- В случайном гене может произойти замена нуклеотидных оснований.

Далее все цепочки ДНК передаются подключенным клиентским приложениям для их дальнейшей обработки.

Каждое клиентское приложение декодирует полученные цепочки: каждое полученное слово становится либо частью оператора, либо частью оперона. Структура оперона включает в себя индекс вещества, на которое действует оперон, коэффициент, определяющий интенсивность генерации или разрушения вещества, а также знак, определяющий, увеличивается или уменьшается концентрация вещества, на которое влияет оперон. Оператор включает в себя индекс вещества, которое отвечает за активацию гена, значение порога и знак, определяющий взаимоотношение между значением вещества и порога.

Алгоритм клеточной дифференциации состоит из двух частей — подсчета изменений сигнальных веществ и перетекания сигнальных веществ из одной клетки в другую. В первой части алгоритма для каждого гена, каждого оператора в нем, вычисляется разность между порогом и значением вещества или значением вещества и порогом (на это влияет знак оператора). После, для каждого оперона в гене и каждого значения разности, вычисляется значение сигма-функции от разности, умножается на коэффициент взаимодействия из оперона и, в зависимости от знака оперона, прибавляется к текущему веществу или отнимается от него.

Решение вычислять значение сигма-функции от разности, а не работать с линейным приращением было принято вследствие стохастичности процесса экспрессии. На самом деле, либо ген полностью блокирован, либо полностью работает. Но репрессор может случайным образом прицепляться и отцепляться. Чем больше концентрация репрессора, тем чаще его молекула ингибирует экспрессию гена. Поэтому, вместо моделирования стохастического процесса, мы усредняем его, считая, что непрерывное изменение концентрации непрерывно меняет репрессию. Сигма-функция была выбрана из-за подходящей для этой модели асимптотики.

Вторая часть алгоритма включает в себя диффузию веществ между клетками. Процесс протекает независимо для каждой из компонент вектора веществ.

Каждый организм состоит из $N \times N$ клеток, каждая из клеток имеет вектор сигнальных веществ. Начальная анизотропия веществ определяется случайным образом, но при этом у всех существ она одинакова. Таким образом клиентское приложение никак не может повлиять на результат работы генов, полученных от серверного приложения. Также были выделены три не сигнальных вещества — цветовые компоненты.

В начале работы клиентского приложения организм состоит из 4-х клеток. В процессе работы, для каждой клетки, вычисляются новые значения вектора

веществ на основе данных, полученных из цепочек ДНК, а раз в несколько итераций этих вычислений, клетки делятся, моделируя рост существа в 2 раза.

Как только существо достигает своего максимального размера, процесс эмбриогенеза заканчивается. Специальная фитнес-функция сравнивает полученное изображение с заранее заданным и определяет параметр схожести. Этот параметр и возвращается серверному приложению.

Получив данные о схожести для каждой из отправленных цепочек ДНК, серверное приложение отбирает максимально подходящие и запускает процесс заново уже для них.

Таким образом, с каждой итерацией цикла работы программы строятся существа, все сильнее и сильнее похожие на заданное.

2. РЕАЛИЗАЦИЯ МОДЕЛИ ГЕНЕТИЧЕСКОГО КОДА

2.1 Инициализация серверного и клиентских приложений

Для связи был выбран протокол TCP/IP. Есть несколько подходов к созданию TCP-серверов. В данной работе был реализован классический вариант. Подробное описание TCP/IP приложений приведено в [1] и [4]. Мы не будем останавливаться на создании сервера и клиентов, опишем лишь в общих чертах их инициализацию в данной работе.

Сперва опишем процесс инициализации сервера:

1. Задаем дескрипторы сокета и инициализируем все клиентские сокеты как непроверенные

2. Создаем мастер сокет, разрешаем ему работать с несколькими клиентами одновременно

3. Задаем параметры в структуре `address` и привязываем сокет к порту, указанному в поле `address.sin_port`

4. Ждем входящих подключений на мастер-сокете. Если что-то происходит на нем, то в первую очередь проверяем на запрос на подключение. Если это не входящее подключение, то считываем входящее сообщение.

- 4.1 Если оно пустое, то кто-то отключился. В таком случае, закрываем сокет и обнуляем его позицию в массиве

- 4.2 Если сообщение пришло с информацией, обрабатываем его.

Для того, чтобы инициализировать клиентское приложение, необходимо:

1. Задать `ip`-адрес сервера и указать нужный порт

2. Задать структуру `address`, и подключиться к серверу по данным `ip` и этой структуры

2.2 Протокол взаимодействия сервера с клиентами

Опишем протокол, по которому сервер взаимодействует с несколькими клиентами одновременно, который был реализован и используется в данной модели.

Сервер:

1. Сервер ждет входящие подключения от клиентов.
2. В начале каждого цикла поколений сервер ждет данных от всех клиентов о их готовности к работе.
3. После генерации нового поколения, сервер отправляет каждому из подключенных клиентов данные о сформированных геномах – длину, порядковый номер, массив 16-битных слов.
4. Сервер получает от клиентов номер существа и данные о похожести этого существа на идеальное
5. Как только сервер получит данные для всех существ, он отправляет клиентам стоп-слово, говорящее о смене поколений

Клиент:

1. Клиент принимает данные о геномах - длину, порядковый номер, массив 16-битных слов.
2. После обработки этих данных отправляет серверу номер существа и данные о похожести этого существа на идеальное.
3. Как только клиент получит стоп-слово, он отправит серверу сообщение о готовности к новому циклу.

При этом действуют следующие правила передачи данных между серверным и клиентским приложениями:

- Сервер отправляет по одному геному на каждый из доступных клиентов;

- После того, как каждый клиент получает геном для обработки (либо как только все геномы будут отправлены), сервер ждет ответа от всех клиентов с результатами работы, и только после этого отправит новую партию данных;

- Обмен происходит кадрами фиксированного размера (в текущей реализации принят размер буфера 65535 байт). Каждый кадр содержит ASCIIZ-строку (байты после конечного нуля не имеют значения), которая содержит либо управляющую строку (команда 12345678 – число, которое заведомо не встретится в процессе обмена данными), либо десятичную запись целого или вещественного числа. При этом, так как протоколом жестко определено, когда отправляется та или иная информация, оба приложения знают точно когда полученный кадр нужно рассматривать как целое число, а когда как вещественное.

2.3 Реализация модели

Для реализации модели был выбран язык C++. Модель, представленная в пункте 1.2, реализуется в виде набора следующих структур и функций.

Структура клетки

```
struct cell{  
    unsigned int v[SUBSTANCE_LENGTH];  
    int dv[SUBSTANCE_LENGTH];  
};
```

Клетка состоит из двух массивов – текущие значения сигнальных веществ и измененные значение веществ на текущий стадии алгоритма диффундирования.

Структура генома

Серверное приложение	Клиентское приложение
<pre>struct uint16_genome { uint16_t * genes; int size; };</pre>	<pre>struct genome{ struct gene *genes; int length; float similarity; }; struct gene{ struct cond *cond; struct operon *operons; int cond_length; int oper_length; };</pre>

Структура генома состоит из вектора генов и длины генома. Тем не менее, между серверным и клиентским приложениями эти структуры немного отличаются.

У серверного приложения массив генов представлен указателем на вектор 16-битных слов, а у клиентского приложения для каждого гена потребовалось вводить структуру, состоящую из указателей на вектора операторов и оперонов. При этом структуры отдельных операторов и оперонов имеют вид

Оперон	Оператор
<pre>struct operon{ unsigned char rate : 7; unsigned char sign : 1; unsigned char substance : 7; };</pre>	<pre>struct cond{ unsigned char threshold : 7; unsigned char sign : 1; unsigned char substance : 7; };</pre>

Структура оперона включает в себя индекс вещества, с которым взаимодействует оперон, коэффициент, влияющий на значения пороговой функции и знак, отвечающий за приращение или уменьшение сигнального вещества.

Структура оператора включает в себя индекс вещества и значение порога, которые влияют на активность оперонов. В дополнение к этому, также есть бит

знака, влияющий на вычисление разности значений между пороговым веществом и значением порога.

Клиентское приложение дешифрует 16-битное слово, исходя из данных в таблице 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
Type bit	Sign bit	Thereshold							Substance							
		Rate														

Таблица 1. Представление операторов и оперонов в виде 16-битного слова

Type bit определяет, является слово оператором или опероном;

Sign bit определяет знак (> | <, либо - | +);

Thereshold, Rate и Substance определяют соответствующие поля в структурах оперона и оператора.

Основной алгоритм работы серверного приложения представлен в таблице ниже. Для краткости записи каждая передача данных тут выражена одной функцией `recv` либо `send`. На самом деле, при передаче данных необходимо отслеживать возвращаемое значение, т.к. иначе данные могут быть отправлены / получены неполностью.

```

Do {
    <<<<<<<< Generate next generation >>>>>>>>

    int cur_clients = 0;
    for (i = 0; i < max_clients; i++) {
        if (client_socket[i] > 0) {
            cur_clients++;
        }
    }

    for (int i = 0; i < START_POPULATION; i++) {
        copyGenome_uint16(firstGeneration[i], nextGeneration[i]);
    }
    for (int i = START_POPULATION; i < NEXT_GENERATION_POPULATION; i++) {
        crossGenome_uint16(
            firstGeneration[RangedRandomNumber(0, START_POPULATION - 1)],
            firstGeneration[RangedRandomNumber(0, START_POPULATION - 1)],

```

```

        nextGeneration[i]);
    }

    <<<<<<<< Mutations >>>>>>>>>

    int numberOfMutations = RangedRandomNumber
        (NEXT_GENERATION_POPULATION/2, 10*NEXT_GENERATION_POPULATION);
    for (int i = 0; i < numberOfMutations; i++) {
        int mutation = RangedRandomNumber(0,100);
        int creature = RangedRandomNumber(0, NEXT_GENERATION_POPULATION - 1);
        if (mutation < 13) {
            ChangeRandomBits(nextGeneration[creature]);
        } else if (mutation < 25) {
            RandomFragmentDeletion(nextGeneration[creature]);
        } else if (mutation < 38) {
            RandomFragmentInsetrion(nextGeneration[creature]);
        } else if (mutation < 50) {
            RandomFragmentDuplicate(nextGeneration[creature]);
        } else if (mutation < 63) {
            RandomFragmentMove(nextGeneration[creature]);
        } else if (mutation < 75) {
            RandomFragmentCopy(nextGeneration[creature]);
        } else if (mutation < 88) {
            int secondCreature = RangedRandomNumber
                (0, NEXT_GENERATION_POPULATION - 1);
            ExchangeDnaFragments
                (nextGeneration[creature], nextGeneration[secondCreature]);
        } else if (mutation < 100) {
            RandomFragmentReverse(nextGeneration[creature]);
        }
    }
}

<<<<<<<< Sending data to clients >>>>>>>>>
<<<<<<<< Receiving data from clients >>>>>>>>>

int pos = 0;
for (int i = 0; i < NEXT_GENERATION_POPULATION; i++) {
    strcpy(buffer, to_string(nextGeneration[i]->size).c_str());
    send(client_socket[pos], &buffer, bufsize, 0);

    strcpy(buffer, to_string(i).c_str());
    send(client_socket[pos], &buffer, bufsize, 0);

    for (int j = 0; j < nextGeneration[i]->size; j++) {
        strcpy(buffer, to_string(nextGeneration[i]->genes[j]).c_str());
        send(client_socket[pos], &buffer, bufsize, 0);
    }

    pos++;
    if (pos == cur_clients || i == NEXT_GENERATION_POPULATION - 1) {
        for (int j = 0; j < pos; j++) {
            rec = 0;
            do {
                int res = recv(client_socket[j], &buffer[rec], bufsize-rec, 0);
                rec+=res;
            } while (rec<bufsize);

```



```

        if (atoi(buffer) == 12345678) {
            recv(client_socket[j], &buffer, bufsize, 0);
        }
        int genomeNumber = atoi(buffer);
        recv(client_socket[j], &buffer, bufsize, 0);
        nextGeneration[genomeNumber]->similarity = atof(buffer);
    }
    pos = 0;
}
}

```

<<<<<<< **Selecting most similar** >>>>>>>

```

for (int i = 0; i < START_POPULATION; i++) {
    int mostSimilar = 0;
    int val = 1001;
    for (int j = 0; j < NEXT_GENERATION_POPULATION; j++) {
        if (nextGeneration[j]->similarity < val) {
            val = nextGeneration[j]->similarity;
            mostSimilar = j;
        }
    }
    firstGeneration[i] = nextGeneration[mostSimilar];
    nextGeneration[mostSimilar]->similarity = 1001;
}

for (int i = 0; i < max_clients; i++) {

    if (client_socket[i] > 0) {
        strcpy(buffer, to_string(12345678).c_str);
        send(client_socket[i], &buffer, bufsize, 0);
    }
}
}

```

Основной алгоритм работы клиентского приложения

```

Do {

    <<<<<<< Receive DNA >>>>>>>

    recv(client, buffer, bufsize, 0);
    if (atoi(buffer) == 12345678) {
        send(client, "ready", bufsize, 0);
    } else {
        nextGeneration->size = atoi(buffer);
        recv(client, &buffer, bufsize, 0);
        genomeNumber = atoi(buffer);
        nextGeneration->genes =
            (uint16_t*)malloc(nextGeneration->size * sizeof(uint16_t));
        for (int j = 0; j < nextGeneration->size; j++) {
            recv(client, &buffer, bufsize, 0);
            nextGeneration->genes[j] = atoi(buffer);
        }
    }
}

```

```

    }
    nextGeneration->similarity = 65536.0;

    <<<<<<< Growing up creature >>>>>>>

    initCreature(creature);
    int size = creature->n * creature->n;
    genome = (struct genome*)malloc(sizeof(struct genome));
    uint16genome_to_genome(nextGeneration, genome);
    step = 0;
    while(creature->n < MAX_CREATURE_SIZE) {
        calculateDvForCells_v2(creature, genome);
        applyDvForCells(creature);
        diff_v2(creature, matrix);
        applyDiff(creature);
        if (step % 2 == 0) {
            creature = grow(creature);
        }
        step++;
    }

    <<<<<<< Calculating similarity >>>>>>>

    float r = 0, g = 0, b = 0;
    size = creature->n * creature->n;
    for (int j = 0; j < creature->n; j++) {
        for (int k = 0; k < creature->n; k++) {
            float red = 0, green = 0, blue = 0;
            for (int h = 0; h < 10; h++) {
                red += creature->cells[j*creature->n+k].v[h];
                green += creature->cells[j*creature->n+k].v[h+10];
                blue += creature->cells[j*creature->n+k].v[h+20];
            }
            r += abs(red/10 - (float)image[3 * (j * IMAGE_WIDTH + k)]);
            g += abs(green/10 - (float)image[3 * (j * IMAGE_WIDTH + k) + 1]);
            b += abs(blue/10 - (float)image[3 * (j * IMAGE_WIDTH + k) + 2]);
        }
    }
    genome->similarity = (r + g + b) / size;

    <<<<<<< Send results >>>>>>>

    strcpy(buffer, to_string(genomeNumber).c_str());
    send(client, &buffer, bufsize, 0);

    strcpy(buffer, to_string(genome->similarity).c_str());
    send(client, &buffer, bufsize, 0);
}
} while (!isExit);

```

Функции, отвечающие за эмбриогенез, были взяты из модели, представленной в работе [5], и переработаны для работы без использования технологии CUDA.

Таким образом, используя теоретические сведения, представленные ранее, реализуется одна из простейших моделей распределенной эволюции многоклеточных организмов.

2.4 Проблемы реализации и исправления

- Одна из основных проблем – выделение памяти. В ранних версиях модели каждое клиентское приложение получало сразу часть всей популяции. В следствие того, что выделять память под это не выгодно, мы перешли к варианту, при котором клиентскими приложениями обрабатываются цепочки ДНК поочередно, а не разово.

- Так как протокол взаимодействия сервера и клиентов подразумевает то, что подключение новых клиентов сервер обрабатывает только в начале цикла поколения, было решено перед запуском программы явно указывать начальное число клиентов и ждать их подключения.

- Для того, чтобы облегчить вычисления, было решено мутировать только сгенерированные существа, а начальное поколение не трогать. Таким образом, мы гарантированно на каждой итерации получим заведомо не худший результат.

- В начальной модели было предложено считать первые 2 значения вектора веществ у клетки как определяющие положение клетки в 2-мерном пространстве, а значения с 3 по 5 считать как r-g-b данные. Это удобно для тестирования эмбриогенеза, но при работе со случайными числами довольно сложно случайно сгенерировать такой геном, который влиял бы на 3 значения из 128. Поэтому было предложено, и в дальнейшем реализовано, считать r-g-b

значения как среднее арифметическое у веществ № 2-11, 12-21 и 22-31 соответственно.

2.5 Ассемблирование генома

На ранних стадиях разработки приложения, одной из проблем тестирования корректной работы функций эмбриогенеза являлась сложная структура задания генома. Это существенно затрудняло задание значений условий и оперонов. Было решено представлять каждый ген генома строкой вида:

$$[<\text{substance}>] <\text{sign}> <\text{threshold}>, \dots = [<\text{substance}>] <\text{sign}> <\text{rate}>, \dots$$

Здесь слева от знака равенства стоят тройки, которые кодируют значения оператора, а справа — тройки, представляющие опероны. Каждая строка файла с описанием генома представляет собой один ген.

Однако такой геном, представленный в таком виде, неудобно преобразовывать непосредственно в структуру на языке C. Поэтому было решено сначала создавать по геному двоичный файл, который содержал бы в себе только операторы и опероны. Специально для этой цели была написана программа на языке C, осуществляющая разбор генома.

В полученном двоичном файле каждые два байта обозначают оператора или оперон. Оператор от оперона отличается старшим битом. Новый ген генома определяется наличием перехода от оперона к оператору.

Загрузка генома из файла происходит в функции `loadGenome`, находящейся в файле `genome.cpp`.

Так же в связи с тем, что серверное и клиентские приложения работают с разными моделями цепочки ДНК, потребовалось реализовать функции,

обеспечивающие преобразования генома в виде 16-битных слов в геном из модели в работе [5], и наоборот.

3. ТЕСТИРОВАНИЕ

3.1 Фитнес функция

Для тестов была реализована фитнес-функция, которая сравнивала полученное изображение с заданной картинкой по следующей формуле:

$$\begin{aligned} \text{red} &= \sum_{i=0}^{\text{cells}} |\text{red_component} - \text{ideal_red_component}| \\ \text{blue} &= \sum_{i=0}^{\text{cells}} |\text{blue_component} - \text{ideal_blue_component}| \\ \text{green} &= \sum_{i=0}^{\text{cells}} |\text{green_component} - \text{ideal_green_component}| \end{aligned}$$

$$\text{similarity} = (\text{red} + \text{green} + \text{blue}) / \text{cells}$$

Таким образом, выживали существа с наименьшей величиной схожести.

3.2 Тесты

Весь процесс тестирования можно разделить на 2 различных этапа:

- Проверка работоспособности эмбриогенеза
- Тестирование распределенной модели

Первый этап проверялся на тех же входных данных, на которых производилось тестирование в модели, описанной в [5]. Ниже приведены результаты этих тестов.

3.2.1. Тестирование эмбриогенеза

Геном (1), представленный в таблице ниже должен окрашивать верхнюю полуплоскость картинке в синий цвет, а нижнюю в черный.

$$\begin{array}{l} [20] < 127 = [4] + 30 \\ [0] > 0 = [5] + 15 \\ [5] > 30 = [4] - 40 \end{array}$$

геном (1)

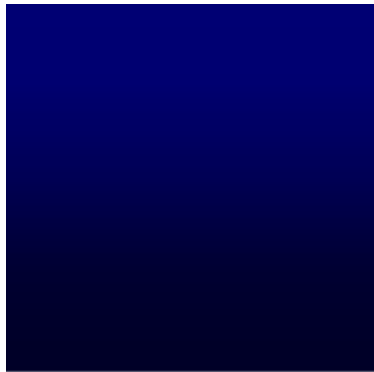


рис.1

Это происходит благодаря подавлению гена, отвечающего за окраску всего существа в синий цвет (первый ген) третьим геном. Результат его работы вы можете увидеть на рисунке 1.

Аналогично, геном (2) должен окрашивать левую половину картинке в красный цвет, а правую в черный.

$$\begin{array}{l} [20] < 127 = [2] + 30 \\ [1] > 0 = [5] + 15 \\ [5] > 30 = [2] - 40 \end{array}$$

геном (2)



рис.2

Это происходит благодаря подавлению гена, отвечающего за окраску всего существа в красный цвет (первый ген) третьим геном. Результат его работы вы можете увидеть на рисунке 2.

В завершение тестирования эмбриогенеза был построен более сложный пример, на основе генома (3).

$[20] < 127 = [10] + 25$
$[1] > 0 = [5] + 15$
$[5] > 30 = [10] - 25$
$[30] < 127 = [10] - 25$
$[0] > 0 = [6] + 15$
$[6] > 30 = [10] + 25$
$[20] < 127 = [11] - 25$
$[1] > 0 = [7] + 15$
$[7] > 30 = [11] + 25$
$[30] < 127 = [11] + 25$
$[0] > 0 = [8] + 15$
$[8] > 30 = [11] - 25$
$[30] < 127 = [3] + 30$
$[10] > 10 = [3] - 30$
$[11] > 10 = [3] - 30$

геном (3)



рис.3

В этом примере изначально формируются сигнальные вещества 10 и 11 в верхнем правом и нижнем левом углу изображения. Одновременно с этим, наращивается компонента, отвечающая зеленому цвету. После этого включаются гены, подавляющие компоненту зеленого цвета на участках, где сигнальные вещества 10 и 11

выражены наиболее сильно. Таким образом и

формируется итоговое изображение, показанное на рисунке 3.

После того, как было проверено, что эмбриогенез перенесен и работает корректно, можно было приступить к тестированию модели эволюции. Сначала она была реализована в одной программе, без деления на клиентскую и серверную части.

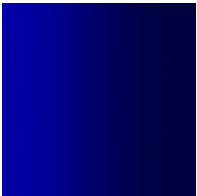




3.2.2 Тестирование модели эволюции

В этом разделе приведены результаты работы некоторых тестов для реализованной программы.



рис.4

Для первого теста была выбрана эталонная картинка, представленная на рис. 4. В таблице ниже представлен срез результатов работы программы у выбранных поколений существ. В этом тесте начальное число существ было равно 100, а после скрещивания доходило до 500.

Эталонная картинка	Поколение 0	Поколение 6	Поколение 15	Поколение 23
				
Similarity	272.938	191.122	145.914	115.095

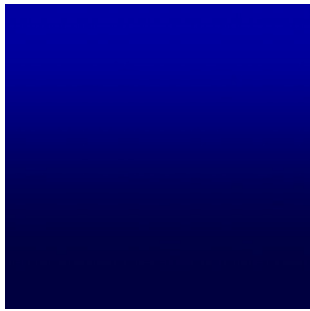







рис.5

Для следующего теста тоже была выбрана простая эталонная картинка (рис. 5). В таблице ниже представлен срез результатов работы программы у выбранных поколений существ. В этом тесте начальное число существ было равно 100, а после скрещивания доходило до 500.

Эталонная картинка	Поколение 0	Поколение 4	Поколение 19	Поколение 27
				
Similarity	301.912	226.3	125.3535	100.634

Как видно из этих тестов, программа работает корректно. Вне зависимости от того, насколько плохими были начальные параметры строящейся картинки, практически во всех проведенных тестах, включая эти, уже на первых поколениях подбираются геномы, которые строят картинку таким образом, что средняя разность между эталонной картинкой и построенной не превосходит значения 200 (максимум – 765). Однако для дальнейшего приближения результатов к эталонным требуется гораздо больше времени и затраченных средств. Именно для этого разрабатывалась распределенная модель, которая за счет параллельного вычисления данных дает ощутимый прирост в скорости вычислений.

Следующие тесты были проведены в результате распределения работы в соответствии с описанием модели.



рис.6

Для дальнейших тестов были выбраны более сложные картинки. Первая представлена на рис.6. В таблице ниже показан срез результатов работы программы у выбранных поколений существ. В этом тесте начальное число существ было равно 50, а после скрещивания доходило до 500. Программа выполнялась

на 3 подключенных клиентских машинах.






Эталон	Поколение 3	Поколение 9	Поколение 11	Поколение 14
				
Similarity	126.144	90.3717	86.515	80.357



рис.7

Для картинки, представленной на рис. 7, был проведен большой тест, для того, чтобы проверить зависимость числа существ в поколении от полученных результатов. В начальном поколении присутствовало 1000 существ, после скрещивания было получено еще 9000. Вся работа была произведена на 8 подключенных

клиентских приложениях.





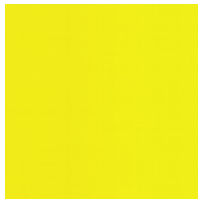


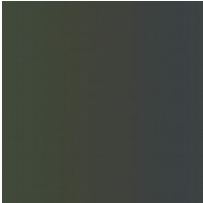


Эталон	Поколение 1	Поколение 2	Поколение 4	Поколение 7
				
Similarity	191.744	144.700	101.08	83.869



рис.8

Еще один тест был проведен на распределенной модели для одной из простейших картинок (рис.8). В начальном поколении присутствовало 25 существ, после скрещивания было получено 250. Вся работа была произведена на 3 подключенных клиентских приложениях.

Эталон	Поколение 0	Поколение 2	Поколение 8	Поколение 13
				
Similarity	220.4930	188.806	101.423	44.83

Существенной особенностью тестов, проведенных в распределенной среде, является выигрыш во времени. Так, тесты для рис. 4 и 5 выполнялись до 23 и 27 поколения соответственно в течение примерно 30 часов каждый. В то же время, тест для рис. 7, в котором было в 20 раз больше данных, выполнялся всего 48 часов и уже на 7 поколении приблизил нас к эталону сильнее, чем ранние тесты на 27 поколении, при этом для гораздо более сложной картинки. Тесты у картинок на рис.6 и 8 выполнялись примерно по 24 часа каждый.

ЗАКЛЮЧЕНИЕ

Результатом данной работы является пара приложений – клиентское и серверное, использующее протокол взаимодействия TCP/IP и моделирующие процесс эволюции многоклеточных организмов, их естественного отбора. В процессе работы были изучены биологические основы развития организмов, эвристические алгоритмы моделирования, выбрано и реализовано их решение.

Эта работа объясняет и наглядно демонстрирует процесс эволюции многоклеточных организмов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Р.Стивенс «Unix — Профессиональное программирование.» –1998г.
- [2] Марков А. “Рождение сложности” – 2010г.
- [3] Б. Аппель «Нуклеиновые кислоты: от А до Я» – 2013г.
- [4] W. R. Stevens “UNIX Network programming” – 1998г.
- [5] Рассчетно - пояснительная записка к курсовому проекту на тему
“Моделирование развития клеточного организма при помощи технологии
CUDA” – [https://github.com/bmstu-iu9/cuda-embriogenesis/blob/master](https://github.com/bmstu-iu9/cuda-embriogenesis/blob/master/Записка.pdf)
[/Записка.pdf](https://github.com/bmstu-iu9/cuda-embriogenesis/blob/master/Записка.pdf) , 2016г.
- [6] А. П. Горкин «Биология. Современная иллюстрированная
энциклопедия.» – 2006г.
- [7] Патрушев Л. И. «Экспрессия генов» -- 2000г.