ОГЛАВЛЕНИЕ

[Определения, обозначения и сокращения 7](#_Toc453926900)

[Введение 8](#_Toc453926901)

[1 Задача раскроя и генетические алгоритмы 11](#_Toc453926902)

[1.1 Задача раскроя 11](#_Toc453926903)

[1.2 Генетический алгоритм 12](#_Toc453926904)

[1.2.1 Описание генетического алгоритма 13](#_Toc453926905)

[1.2.2 Классическая схема генетического алгоритма 15](#_Toc453926906)

[1.2.3 Типы генетических алгоритмов 16](#_Toc453926907)

[1.2.4 Виды селекции 17](#_Toc453926908)

[1.2.5 Виды кроссинговера 18](#_Toc453926909)

[1.3 Параллельные генетические алгоритмы 21](#_Toc453926910)

[1.3.1 Схема «Мастер – раб» 21](#_Toc453926911)

[1.3.2 Крупнозернистая схема 23](#_Toc453926912)

[1.3.3 Мелкозернистая схема 24](#_Toc453926913)

[1.3.4 Смешанные схемы 24](#_Toc453926914)

[2 Обзор средств реализации генетических алгоритмов 26](#_Toc453926915)

[2.1 Требования к средствам разработки 26](#_Toc453926916)

[2.2 Обзор языков программирования 26](#_Toc453926917)

[2.3 Обзор математических пакетов 29](#_Toc453926918)

[2.4 Стандарты OpenMP и MPI 30](#_Toc453926919)

[2.5 Обоснование выбора средств разработки 31](#_Toc453926920)

[3 Разработка параллельного генетического алгоритма 33](#_Toc453926921)

[3.1 Анализ структуры приложения 33](#_Toc453926922)

[3.2 Разработка структуры генетического алгоритма 33](#_Toc453926923)

[3.2.1 Реализация класса гена 34](#_Toc453926924)

[3.2.2 Реализация класса особи 37](#_Toc453926925)

[3.2.3 Реализация класса популяции 40](#_Toc453926926)

[3.2.4 Реализация последовательного генетического алгоритма 42](#_Toc453926927)

[3.3 Описание визуализатора решений 44](#_Toc453926928)

[3.4 Параллельный генетический алгоритм 47](#_Toc453926929)

[3.4.1 Описание реализации схемы «мастер-раб» с делением на подпопуляции 47](#_Toc453926930)

[3.4.2 Описание реализации схемы «мастер-раб» с делегированием вычисления пригодности 48](#_Toc453926931)

[3.4.3 Описание реализации схемы «мастер-раб» с делегированием вычисления всех функций алгоритма 49](#_Toc453926932)

[3.5 Тестирование алгоритма 50](#_Toc453926933)

[3.5.1 Тестирование эффективности последовательного алгоритма 50](#_Toc453926934)

[3.5.2 Сравнение быстродействия 54](#_Toc453926935)

[3.5.3 Сравнение эффективности 55](#_Toc453926936)

[3.5.4 Выводы из результатов тестирования 56](#_Toc453926937)

[Заключение 58](#_Toc453926938)

[Список использованных источников и литературы 60](#_Toc453926939)

[Приложение А Код класса GA\_gene 63](#_Toc453926940)

[Приложение Б Код класса GA\_unit 68](#_Toc453926941)

[Приложение В Код класса GA\_population 78](#_Toc453926942)

Приложение Г Диск с программой……………………………………………………89

# ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящем текстовом документе применяются следующие определения, обозначения и сокращения:

ген – элемент хромосомы;

кроссинговер (кроссовер, скрещивание) – операция, при которой хромосомы участвующих особей обмениваются генами и порождают новую особь;

мутация – случайное изменение одного или нескольких генов хромосомы особи;

особь (решение, индивид) – элемент популяции, хранящий хромосому;

популяция – совокупность особей;

пригодность – численный показатель удачности особи;

селекция – операция выбора особей из популяции для скрещивания;

функция пригодности (фитнес-функция) – функция, вычисляющая пригодность особи;

хромосома – вектор или массив значений.

# ВВЕДЕНИЕ

В то время, когда наши космические корабли бороздят просторы Вселенной, велика роль оптимизации программ для работы на многопроцессорных системах. С каждым днём нарастает потребность в быстрых и точных вычислениях для решения сложных и ресурсоёмких задач. Чтобы удовлетворить потребности науки в вычислительной мощности, были построены мощные суперкомпьютеры, способные выдавать десятки петафлопс, потребляя при этом мегаватты энергии. Велики также и затраты на обслуживание таких ЭВМ – день работы мощнейшего российского суперкомпьютера «Ломоносов» оценивается в 20 тысяч долларов. Однако, статистика показывает, что при отсутствии свободного времени в расписании суперкомпьютера наблюдаемая загруженность колеблется в районе 2-3 процентов. Из этого следует, что, несмотря на то, что компьютер постоянно занят вычислениями, делает он это крайне неэффективно.

Причины низкой загруженности суперкомпьютеров заключаются в программах, исполняемых на них. Достигнув практического предела в повышении производительности последовательных процессоров, наука стала развиваться в сторону увеличения их количества. Появление многопроцессорных компьютеров повлекло за собой появление новой, параллельной парадигмы программирования, но не все алгоритмы удалось удачно перенести на новую платформу. Таким образом, одной из главных задач в области прикладной математики и информатики является исследование алгоритмов на возможность их эффективной работы в многопроцессорных системах и реализация использующих эти алгоритмы программ.

Генетический алгоритм относится к классу эвристических алгоритмов. Он широко применяется для решения сложных или трудоёмких оптимизационных задач, а также NP-полных задач и задач нерешаемых аналитически. В основе идеи алгоритма лежит теория Ч. Дарвина об эволюционном происхождении многообразия живых организмов на нашей планете. Создание эффективного метода перенесения алгоритма на многопроцессорные системы позволит увеличить скорость выполнения алгоритма, что позволит расширить область его применения.

Задача раскроя представляет собой NP-полную задачу, сводящуюся к задаче о рюкзаке. Примеры применения задачи можно встретить во многих областях, в частности можно выделить задачу нарезки производимого рулона бумаги на необходимые заказчикам форматы, раскрой листового металла и древесно-стружечных плит, так как эти задачи актуальны для региона.

Актуальность работы заключается в том, что решение задачи фигурного раскроя полосы материала находит применение в самых разных отраслях производства. Нахождение наиболее оптимального решения данной задачи позволит снизить количество отходов материала, что увеличит прибыль предприятия. Использование современных технологий и алгоритмов, таких как высокопроизводительные вычисления и параллельные генетические алгоритмы оптимизации, позволит добиться высоких результатов и скорости решения задачи фигурного раскроя.

Объектом исследования являются параллельные генетические алгоритмы оптимизации.

Предмет исследования – параллельный генетический алгоритм решения задачи раскроя на кластере САФУ.

Целью выпускной квалификационной работы является разработка параллельного генетического алгоритма решения задачи раскроя на кластере САФУ.

Для достижения цели необходимо выполнить следующие задачи:

* анализ предметной области и обзор инструментов для разработки параллельных программ;
* проектирование структуры параллельного генетического алгоритма решения задачи раскроя;
* разработка программы для кластера САФУ.

Вопросы генетических алгоритмов рассмотрены в российской и зарубежной литературе, научных статьях, среди которых: Т.В. Панченко «Генетические алгоритмы», посвящённая генетическим алгоритмам, их модификациям и их решению с помощью математического пакета Matlab; Рутковская Д., Пилиньский М. Рутковский Л. «Нейронные сети, генетические алгоритмы и нечёткие системы», содержащая раздел, посвящённый генетическим алгоритмам и решениям задач с их помощью в специализированных программах; Erik Cantu-Paz, A Survey of Parallel Genetic Algorithms; Sean Luke, Essentials of Metaheuristics. При написании работы была использована информация вышеприведённых источников.

Методы исследования базировались на использовании анализа, системного подхода, сравнения, методов параллельных вычислений.

Разработка параллельного генетического алгоритма, позволяющего оптимизировать процесс раскроя, даст возможность освободить рабочую силу для других задач, уменьшить расход материала, а также сэкономить время использования суперкомпьютеров, что значительно уменьшит стоимость расчётов и поспособствует снижению стоимости разработки в целом.

В ходе преддипломной практики была проведена апробация работы, в ходе которой была написана статья в сборник «Ломоносовские научные чтения студентов, аспирантов и молодых учёных – 2016». Также работа была выдвинута на конкурс студенческих научных работ 2016.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ И ОБЗОР ИНСТРУМЕНТОВ ДЛЯ РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ
   1. Задача раскроя

Задача раскроя, также известная как задача упаковки, это NP-полная проблема оптимизации расположения объектов на заданном поле, целью которой является уменьшение затрачиваемого материала или пространства.

про исследования

Задачу раскроя можно поделить по пространственному критерию:

* одномерная;
* двумерная;
* трёхмерная;
* многомерная.

Одномерная задача раскроя отвечает на вопросы, возникающие при нарезке изделий или заготовок одинаковой или несущественной ширины, где определяющую роль играет длина заготовки.

Двумерные задачи раскроя отвечают на вопросы оптимального расположения изделий на плоскости заготовок. Именно этим задачам посвящена данная работа.

Трёхмерные задачи раскроя чаще встречаются в виде задач упаковки, где необходимо заполнить некоторое объёмное пространство трёхмерными объектами.

Задачи многомерного раскроя редко востребованы в производстве, но при необходимости возможно добавление к трёхмерной задаче сколь угодного количества измерений.

Рассмотрим подробнее виды двумерных задач:

* гильотинный раскрой;
* прямоугольный раскрой;
* фигурный раскрой.

Задача гильотинного раскроя находит применение в стекольной промышленности, когда необходимо оптимальным образом нарезать материал, используя только разрезы, проходящие по всей длине или ширине листа.

Задача прямоугольного раскроя встречается в большом количестве производств, где необходимо разрезать полотно на прямоугольные заготовки разных размеров.

В данной работе рассматривается задача фигурного раскроя.

Допустим, что имеется полубесконечная полоса материала с заданной шириной и массив объектов , содержащих массив последовательно соединённых точек двумерного пространства , образующих многоугольник. Задача состоит в том, чтобы задать расположение элементов так, что будет использован минимально длинный отрезок полубесконечной полосы. Расположение объектов должно удовлетворять нескольким условиям:

* все объекты массива должны быть помещены на полубесконечную полосу;
* стороны объектов не должны пересекаться;
* точки объектов должны лежать только в полосе шириной , начиная с нулевых координат.

В случае, если поставленные условия выполнены, то полученный массив координат объектов можно считать решением задачи, а максимально отдалённая от нулевой отметки полосы точка является показателем оптимальности расположения объектов. Таким образом, целью оптимизации является сокращение используемой длины полубесконечной полосы материала.

* 1. Генетический алгоритм

Генетические алгоритмы являются эффективными поисковыми и оптимизационными алгоритмами, основанными на принципах естественного отбора и генетики. Они применяются для поиска решений в прикладных задачах инженерии, экономики, науки. Генетические алгоритмы способны находить качественные решения за приемлемое время, но при их применении к более сложным и большим задачам время нахождения решения начинает возрастать до неудовлетворяющих значений. Одним из способов сокращения времени, затрачиваемого на поиск решения, является увеличение скорости сходимости алгоритма к оптимуму. В случае генетического алгоритма, использование технологий параллельного программирования успешно ложится на модель алгоритма и позволяет применить разные техники при оптимизации.

* + 1. Описание генетического алгоритма

Генетические алгоритмы относятся к классу эвристических алгоритмов. Данные алгоритмы применяются тогда, когда классические методы терпят неудачу. Цель эвристических алгоритмов заключается в поиске решения, оптимального по точности и времени нахождения. Полученное решение не обязано быть единственным или абсолютно точным – оно должно удовлетворять погрешности и времени нахождения.

Основой принципа работы генетического алгоритма служит эволюционная теория Ч. Дарвина. Алгоритм создаёт популяцию особей, каждая из которых является потенциальным решением, и находит оптимальную, используя принципы естественного отбора и скрещивания индивидов. В процессе поиска решения вычисляются пригодности особей популяции, и, исходя из этого показателя, строится новая популяция посредством скрещивания особей. Особи с низким показателем пригодности удаляются из популяции, оставляя в ней только лучших индивидов[8, 15, 17, 23]. Процесс создания новой популяции и удаления неудачных особей повторяется до тех пор, пока не будет достигнута пригодность, удовлетворяющая погрешности. Рассмотрим основные элементы генетического алгоритма.

Одним из основных элементов поиска решения генетическим алгоритмом является механизм скрещивания, так же называемый кроссинговером. Существует множество видов кроссинговера, о которых будет рассказано позже, однако их всех   
  
объединяет следующее:

* в кроссинговере участвует более одного решения;
* кроссинговер не порождает особей, одинаковых к особям-родителям;
* полученные в результате кроссинговера особи имеют признаки особей-родителей.

Исходя из перечисленных требований к кроссинговеру, можно заключить, что данная операция должна каким-либо образом порождать из нескольких особей нового, отличного от них индивида, но также состоять только из частей особей-родителей.

Следующим важным элементом генетического алгоритма является функция пригодности. От того, как будет оцениваться решение, зависит то, как популяция будет стремиться к оптимуму. К примеру, при поиске минимума функции достаточно в качестве функции пригодности взять её значение в точке, заданной генами особи. Тогда индивиды популяции, дающие в функции максимальные значения, будут удаляться из неё, а новые особи, имеющие более низкие значения, будут помещены на место удалённых.

Важным элементом поиска решений является мутация. При её отсутствии популяция решений стремится к локальному минимуму. Отсутствие мутации приводит к вымиранию генетического фонда особей, что не позволяет естественному отбору приблизиться к оптимальному решению из-за отсутствия разнообразия хромосом, скрестив которые можно получить более хороший результат. Таким образом, мутация является важным элементом генетического алгоритма.

* + 1. Классическая схема генетического алгоритма

Рассмотрим классическую схему[15, 17, 23] генетического алгоритма. Она может быть представлена в виде блок-схемы (рисунок 1).

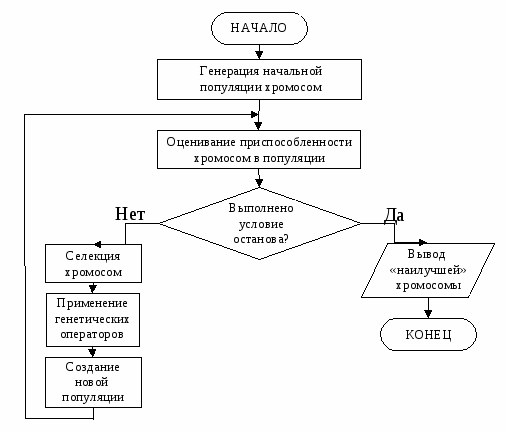


Рисунок 1 – Классическая схема генетического алгоритма

Из данной схемы видно, что основной цикл классического генетического алгоритма состоит из следующих операций:

* селекция особей;
* применение к выбранным особям кроссинговера или мутации;
* запись особей в новую популяцию;
* вычисление приспособленности.

По завершении цикла следует проверка на соответствие лучшей особи популяции критерию остановки алгоритма. В случае удовлетворения необходимого условия, алгоритм завершает работу и выводит лучшую особь как ответ на задачу с поставленной погрешностью.

Классическая схема алгоритма не является ни критерием, определяющим генетический алгоритм, ни строгой рекомендацией к его структуре. Она лишь показывает стандартный подход к решениям задач при помощи генетического алгоритма.

* + 1. Типы генетических алгоритмов

Генетические алгоритмы можно разделить на два вида, основываясь на структуре генов в хромосоме:

* бинарная хромосома, представляющая собой бинарную строку;
* вещественная хромосома, представляющая собой совокупность вещественных показателей.

Бинарные хромосомы могут применяться там, где возможно эффективно представить гены в бинарном виде. В таких случаях хромосома записывается в виде бинарной строки и все преобразования происходят при помощи булевых функций. Данный подход позволяет использовать одну реализацию генетического алгоритма для решения разных задач без каких-либо существенных изменений, однако это не всегда даёт схожий с изначальной задачей эффект оптимизации.

Вещественные хромосомы применяются тогда, когда гены представляют сложную структуру, которую невозможно эффективно представить в виде бинарной строки. В этом случае необходимо реализовать все операции генетического алгоритма для применения к заданным значениям. Данный алгоритм невозможно применить к другим задачам, но он способен эффективно решать поставленную изначально проблему.

При построении генетического алгоритма необходимо выбрать тип хромосомы и её генов. От данного выбора будут зависеть операторы кроссинговера, скрещивания, мутации, селекции и функция пригодности.

* + 1. Виды селекции

Существует множество способов выбора особей для скрещивания. Удачный выбор метода селекции может способствовать ускорению схождения алгоритма к глобальному оптимуму, однако в противном случае алгоритм может сойтись к локальному оптимуму и остаться в нём. Рассмотрим наиболее распространённые виды селекции.

Панмиксия – самый простой оператор отбора. Выбирается две случайных особи из популяции, однако они не должны быть одной особью. Несмотря на простоту, такой подход универсален для решения различных классов задач. Однако, он достаточно критичен к численности популяции, поскольку эффективность алгоритма, реализующего такой подход, снижается с ростом численности популяции.

Селекция состоит в том, что родителями могут стать только те особи, значение приспособленности которых не меньше пороговой величины, например, среднего значения приспособленности по популяции[15, 17]. Такой подход обеспечивает более быструю сходимость алгоритма. Однако из-за быстрой сходимости селективный выбор родительской пары не подходит тогда, когда ставится задача определения нескольких экстремумов, поскольку для таких задач алгоритм, как правило, быстро сходится к одному из решений. Кроме того, для некоторых многомерных задач со сложным ландшафтом целевой функции быстрая сходимость может превратиться в преждевременную сходимость к локальному оптимуму. Этот недостаток может быть отчасти компенсирован использованием подходящего механизма отбора, который бы «тормозил» слишком быструю сходимость алгоритма.

При турнирном отборе из популяции, содержащей особей, выбираются случайным образом особей, и лучшая из них особь записывается в промежуточный массив. Эта операция повторяется раз. Особи в полученном промежуточном массиве затем используются для скрещивания. Количество особей   называют численностью турнира[15, 17]. Преимуществом данного способа является то, что он не требует дополнительных вычислений.

В методе рулетки особи отбираются с помощью «запусков» рулетки, где  – размер популяции. Колесо рулетки содержит по одному сектору для каждого члена популяции. Размер i-го сектора пропорционален вероятности попадания в новую популяцию , вычисляемую по формуле:

(1)

где – пригодность i-ой особи.

Ожидаемое число копий i-ой хромосомы после оператора рулетки определяется по формуле . При таком отборе члены популяции с более высокой приспособленностью с большей вероятностью будут чаще выбираться, чем особи с низкой приспособленностью.

* + 1. Виды кроссинговера

Существует множество способов задать операцию кроссинговера. Конечный выбор метода скрещивания зависит от типа генетического алгоритма, поставленной задачи, выбранной структуры генов и вида хромосомы. Реализации кроссинговера могут отличаться от задачи к задаче, однако существуют классические виды скрещивания. Операции кроссинговера можно поделить на две группы:

* дискретная рекомбинация;
* бинарная рекомбинация.

Как видно из названий, операции скрещивания делятся по типу скрещиваемых хромосом. Рассмотрим каждую группу.

Дискретная рекомбинация применяется для скрещивания вещественных хромосом, однако некоторые из её видов можно успешно применять и для бинарных хромосом. Среди дискретных рекомбинаций можно выделить следующие виды:

* простая дискретная рекомбинация;
* промежуточная рекомбинация;
* линейная рекомбинация.

Рассмотрим простую дискретную рекомбинацию. Допустим, что наша хромосома состоит из трёх вещественных генов . Тогда составим простую дискретную рекомбинацию используя номера генов и определим вид хромосомы потомка. Введём две хромосомы

Существует много различных вариантов составления хромосомы-потомка из данных хромосом-родителей. Выберем любой из них:

Таким образом, имея мы получим потомка:

Данная рекомбинация легко масштабируется до необходимого количества разнообразных потомков особей-родителей. Также она может применяться и для бинарных хромосом.

Далее рассмотрим промежуточную рекомбинацию. Данный вид скрещивания возможно применить только к вещественным хромосомам. Целью этой рекомбинации является определение промежутка значений гена потомка исходя из значений генов родителей. Общая формула для каждого гена будет иметь вид:

|  |  |
| --- | --- |
|  | (2) |

где – множители, определённые для каждого элемента хромосомы.

Возьмём гены из предыдущей рекомбинации и определим для них вектор множителей . Тогда гены потомка будут рассчитаны следующим образом:

Таким образом, хромосома .

Данная рекомбинация масштабируема на получение любого количества потомков от особей-родителей[15, 17].

Следующая рекомбинация называется линейной и отличается от промежуточной тем, что множитель выбирается для каждой хромосомы-потомка один раз и применяется ко всем генам.

Далее рассмотрим бинарные кроссинговеры. Самым простым представителем бинарного скрещивания является одноточечный кроссинговер. В данном случае важно, чтобы хромосомы выбранных особей имели одинаковую длину. Для выполнения одноточечного кроссинговера необходимо случайным образом одинаково разрезать хромосомы особей-родителей на две части и записать в две особи-потомка смешанные хромосомы.

Двухточечный кроссинговер выглядит практически также, однако в нём допускается, что вырезаемый участок может начинаться в конце хромосомы и заканчиваться в начале. Таким образом, хромосома представляется в виде замкнутого круга, где за последним элементом следует первый. Далее выбираются две точки на круге и вырезаются элементы между ними. В особи потомки также записываются смешанные хромосомы, развёрнутые после обмена вырезанными участками.

В многоточечном кроссинговере составляется массив точек разрыва. Используя данный массив, формируются особи-потомки[15, 17]. Существует реализация, в которой элементы массива мутируют, то есть изменяются на некоторую случайную величину, через определённое количество скрещиваний.

Однородный кроссинговер записывает гены в особи-потомки согласно случайно генерируемой маске, в которой указано, какие гены должна отдать та или иная особь-родитель.

Триадный кроссинговер отличается от однородного тем, что после отбора пары родителей из остальных членов популяции случайным образом выбирается особь, которая в дальнейшем используется в качестве маски. Далее 10% генов маски мутируют. Затем гены первого родителя сравниваются с генами маски: если гены одинаковы, то они передаются первому потомку, в противном случае на соответствующие позиции хромосомы потомка переходят гены второго родителя. Генотип второго потомка отличается от генотипа первого тем, что на тех позициях, где у первого потомка стоят гены первого родителя, у второго потомка стоят гены второго родителя и наоборот.

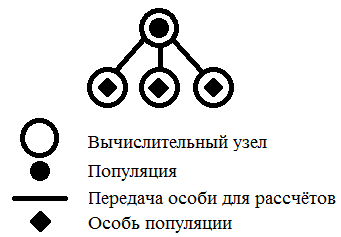
В перетасовочном кроссинговере особи, отобранные для кроссинговера, случайным образом обмениваются генами. Затем выбирают точку для одноточечного кроссинговера и проводят обмен частями хромосом. После скрещивания созданные потомки вновь тасуются. Таким образом, при каждом кроссинговере создаются не только новые потомки, но и модифицируются родители, что позволяет сократить число операций по сравнению с однородным кроссинговером.

* 1. Параллельные генетические алгоритмы

Благодаря своей структуре, генетические алгоритмы удобно и эффективно реализуются на параллельных системах. Параллельные генетические алгоритмы базируются на множественности популяций или же множественному подходу к их обработке. Существует множество схем работы таких алгоритмов, далее будут рассмотрены основные их виды.

* + 1. Схема «Мастер – раб»

Данная схема использует одну популяцию, однако операции генетического алгоритма обрабатываются параллельно (рисунок 2). Чаще всего, центральный вычислительный узел-«мастер» хранит особи популяции, отсылая необходимых индивидов узлам-«рабам» для вычисления приспособленности, так как данная операция чаще всего является самой ресурсоёмкой в генетическом алгоритме[15, 21].

  
Рисунок 2 – Схема «Мастер – раб»

На следующем рисунке представлена блок-схема с распараллеливание вычисления пригодности.

C:\Users\Администратор\Downloads\master-slave.png

Рисунок 3 – Блок-схема «Мастер – раб»

Приведённый выше алгоритм распараллеливания удобен при использовании компьютеров с общей памятью, так как в нём времязатратная операция перемещения данных между вычислительными узлами используется часто. Также алгоритм «мастер – раб» не сильно отличается от последовательного генетического алгоритма, что позволяет работать с ним как с ускоренной последовательной реализацией.

* + 1. Крупнозернистая схема

В данной схеме акцент распараллеливания смещён на образование небольших популяций, соединённых друг с другом с помощью миграций особей (рисунок 4).

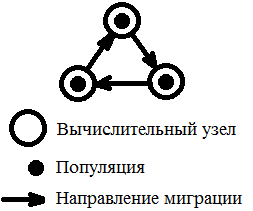


Рисунок 4 – Пример крупнозернистой схемы распараллеливания

Каждая отдельная популяция ведёт себя как последовательный генетический алгоритм, однако через определённое количество циклов происходит миграция определённых особей в другие популяции, расположенные на других вычислительных узлах[15, 21]. Миграция происходит в заранее определённых направлениях.

* + 1. Мелкозернистая схема

В этой схеме используется одна большая популяция, особи которой распределены по вычислительным узлам, связанным определённой топологией (рисунок 5).

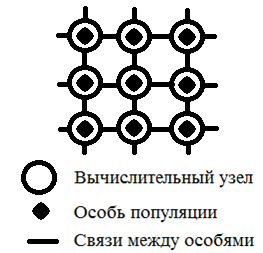


Рисунок 5 – Часть мелкозернистой схемы

Данная схема эффективна при ресурсозатратных фитнес-функциях. Также, для скрещивания особям доступны лишь соседние по топологии особи, что позволяет получать различные результаты ускорения на различных топологиях вычислительной сети.

* + 1. Смешанные схемы

Описанные выше схемы можно комбинировать, получая более высокий показатель ускорения. Например, вложив в мелкозернистую схему в крупнозернистую, можно получить множество связанных миграцией популяций, эффективно вычисляемых на разных узлах[21]. Пример данной схемы представлен на следующем рисунке.

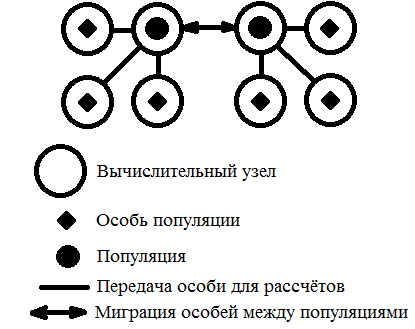


Рисунок 6 – Смешанная схема «Мастер – раб» и крупнозернистая схема

Приведённые выше схемы распараллеливания генетического алгоритма давно изучены, однако не существует гарантий того, как будут вести себя данные алгоритмы на конкретных задачах. Таким образом, при реализации параллельного генетического алгоритма необходимо осознавать важность выбора схемы распараллеливания.

1. ПРОЕКТИРОВАНИЕ СТРУКТУРЫ ПАРАЛЛЕЛЬНОГО ГЕНЕТИЧЕСКОГО АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ РАСКРОЯ
   1. Требования к структуре параллельного генетического алгоритма

Для создания гибкой архитектуры необходимо заранее учесть особенности программ для систем с распределённой памятью, таких, как кластер САФУ. Рассмотрим некоторые из них применимо к генетическому алгоритму.

Распределение настроек алгоритма по узлам и их хранение – необходимо для того, чтобы программа позволяла гибко оперировать настройками алгоритма для каждой конкретной популяции. Например, наличие таких настроек на каждом узле позволит создавать параллельные схемы, в которых разные популяции имеют разный размер, параметры мутации, скрещивания. Это позволит исследовать эффективность смешанных параллельных генетических алгоритмов.

Создание и хранение специфичной схемы распараллеливания – необходимо, чтобы программа имела возможность гибкой настройки схемы миграции особей между популяциями. Для этого необходимо, чтобы каждый узел знал своё место в данной схеме и мог передать особь, а принимающий узел был готов к приёму. Решение данной проблемы позволит отделить схему параллелизма алгоритма от конкретных популяций, разделяя сложную задачу на две более простые.

На основе данных проблем была разработана общая схема распараллеливания генетического алгоритма для систем с распределённой памятью (рисунок ?).

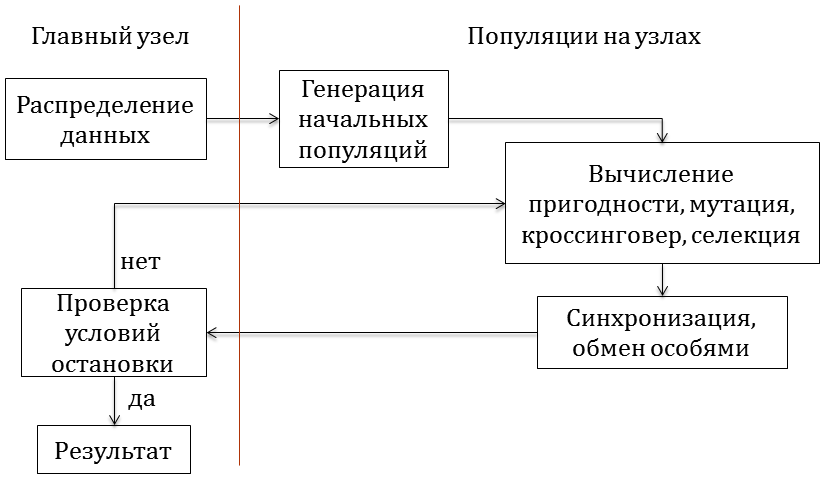


Рисунок ? – общая схема алгоритма

Рассмотрим подробнее элементы генетического алгоритма и сравним со схемой на рисунке, чтобы распределить данные и функции между сущностями в соответствии с концепцией параллельных алгоритмов.

Самой малой сущностью генетического алгоритма является ген. Как видно на рисунке ?, прямого взаимодействия с генами не происходит, таким образом, весь функционал и данные генов можно спрятать в особь – сущность хранящую гены. Особь же, напротив, активно участвует в параллельном сегменте алгоритма на этапе синхронизации и обмена особями. Отсюда можно заключить, что для сущности особей необходим общий интерфейс, предоставляющий функционал независимый от реализации самих особей.

Следующая сущность алгоритма – популяция. Из схемы видно, что кроме функций генетического алгоритма, таких как кроссинговер, селекция и мутация особей, популяция также осуществляет предоставление особей другим популяциям на других узлах.

Рассмотрим распределение данных на узлы. Популяции, расположенные на узлах, ничего не знают о начальных данных. Главный поток получает эти данные от пользователя и затем должен разослать их остальным потокам. Для этого требуется ещё одна сущность, которая хранит в себе начальные данные. Главный поток будет принимать начальные данные от пользователя, формировать их в сущность и распределять между узлами, которые будут ждать получения сущности с данными.

Исходя из проведённого выше анализа схемы распараллеливания генетического алгоритма для систем с распределённой памятью можно нарисовать следующую схему, представляющую уровни абстракции проектируемого алгоритма (рисунок ?).

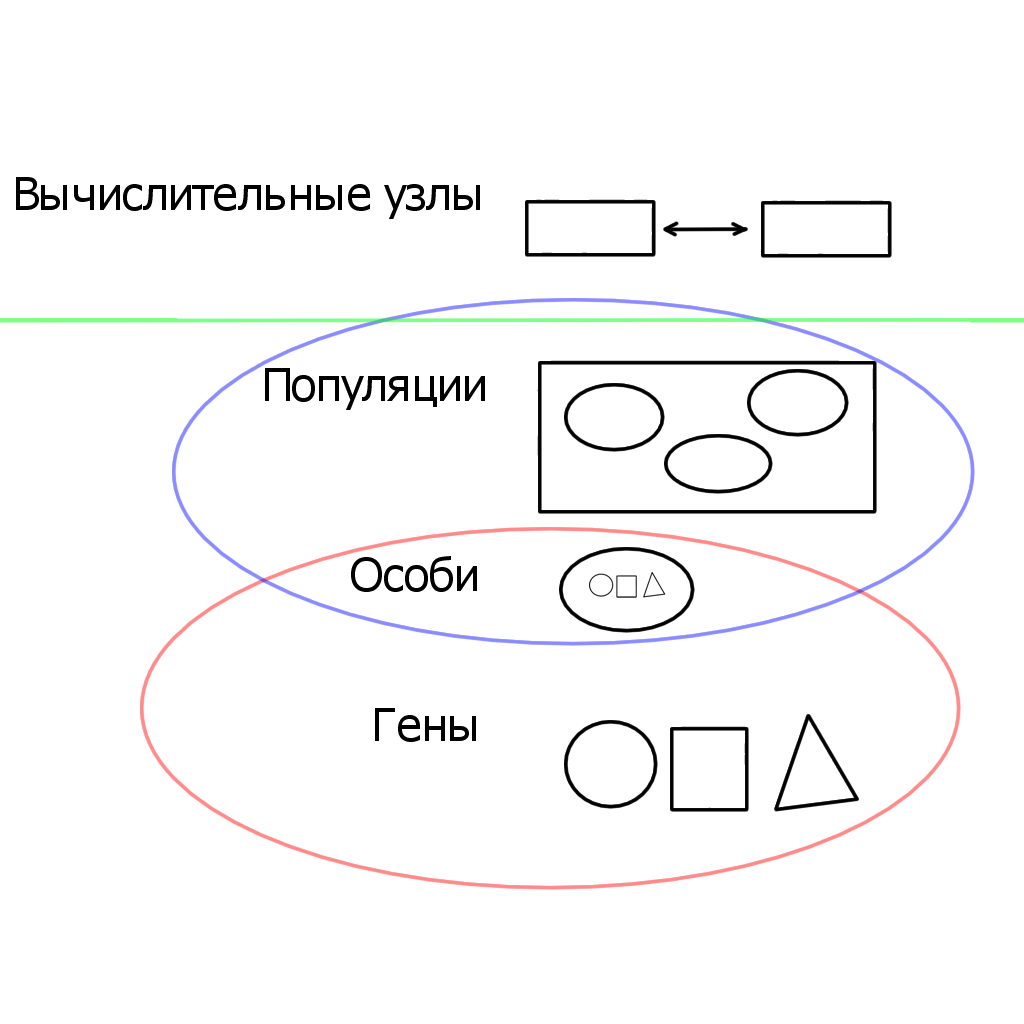


Рисунок ? – уровни абстракции проектируемого алгоритма на примере решения задачи раскроя

На рисунке ? изображены три области, пересекающиеся двух местах. Нижняя область – связка ген-особь. При анализе общей схемы алгоритма было выявлено, что никто из сущностей, кроме особи, что содержит гены, не знает об их существовании. Данный факт позволяет реализовывать гены любым способом.

Вторая связка, представленная на рисунке – связка особь-популяция. Ранее было решено, что популяции не должны ничего знать о генах, соответственно всё, что необходимо популяциям содержат в себе особи, обладающие единым интерфейсом, через который они осуществляют взаимодействие с популяцией.

Третья связка – связка популяция-узел. Для осуществления миграции особей между популяциями расположенными на разных узлах, необходимо сущности, контролирующей узел, предоставить функционал в популяции, позволяющий выделять особь и отправлять её на другой узел. Соответственно, сущность, контролирующая выполнение программы на узле, должна иметь функционал приёма-передачи данных между популяциями.

Учитывая рассмотренные ранее особенности, были составлены схемы, которые помогли определить сущности будущей реализации параллельного генетического алгоритма на системе с распределённой памятью. Подведём итог, описав каждую сущность.

Сущность особи:

* скрывает в себе реализацию генов;
* предоставляет интерфейс генетических операторов, таких как кроссинговер, мутация и вычисление пригодности, для вызова сущностью популяции;
* имеет интерфейс приёма-передачи, так как только особи известно, какое строение данных скрыто в генах и только она может осуществить их приём и передачу.

Сущность популяции:

* обращается к особям по общему интерфейсу, позволяя не учитывать особенности решаемой задачи;
* предоставляет интерфейс приёма-передачи особей между популяциями для сущности, контролирующей узел;
* реализует основные функции генетического алгоритма, такие как отбор, выполнение мутации и кроссинговера особей.

Сущность, контролирующая узел:

* реализует схему взаимодействия популяций между узлами;
* распределяет данные между популяциями на узлах.

Распределив таки образом обязанности для сущностей, мы получаем следующие качества нашего общего алгоритма:

* особи отделены от реализации популяции, что позволяет менять задачу, которую решает алгоритм, простой реализацией особи для другой задачи;
* популяции отделены от особей и параллельной части, что позволяет менять генетические операторы над группами особей;
* популяции и особи реализуют интерфейсы передачи данных, позволяя менять особи и популяции, оставляя параллельную схему алгоритма и наоборот.

В результате выполнения данных требований к сущностям, мы получаем гибкую систему для решения задач генетическим алгоритмом на многопроцессорных системах с распределённой памятью, состоящую из легко заменяемых блоков и способную решать генетическим алгоритмом как разные задачи одинаковыми методами, так и одну задачу разными подходами.

* 1. Проектирование базовых классов генетического алгоритма

Ранее были определены сущности и функции, которые они будут выполнять в алгоритме. Далее необходимо рассмотреть подробнее каждую сущность и их взаимосвязь.

Рассмотрим сущность особи и её взаимосвязь с популяцией. Несмотря на то, что в особи и её генах может храниться любая задача, особь должна иметь набор функций, которые обеспечивали бы популяции полноценный доступ к необходимым генетическому алгоритму параметрам, таким как пригодность особи. Зная пригодность, популяция сможет осуществлять сортировку, селекцию и отбор особей для применения генетических операторов. Вычисление пригодности также должно вызываться популяцией.

Исходя из результатов проектирования, диаграмма базового класса особей будет иметь вид, представлены на рисунке ?.

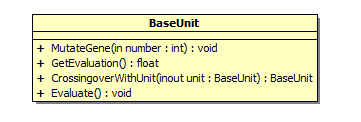


Рисунок ? – диаграмма базового класса особей

Далее рассмотрим сущность популяции. Главной исполняемой функцией популяции является функция итерации алгоритма. Также, в популяции должны быть выделены функции кроссинговера, мутации, отбора и проверки условия остановки алгоритма. Получаем диаграмму базового класса популяций, представленную на рисунке ?.

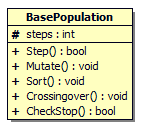


Рисунок ? – базовый класс популяции

Рассмотрим сущность, хранящую настройки алгоритма. Данный класс должен предоставлять популяции всю информацию о особях решаемой задачи. Применим для этого паттерн «Фабричный метод», спрятав от популяции в наследник базовых настроек создание новых особей. Благодаря такому подходу мы полностью отвяжем популяцию от конкретных настроек и особей.

Разместим в классе настроек такие базовые параметры алгоритма, как количество особей, количество новых особей для создания разными операторами алгоритма и количество шагов алгоритма.

Для некоторых задач и их реализаций может потребоваться произвести какие-либо подготовительные операции. Для этого добавим функцию инициализации, которую популяция будет запускать перед началом выполнения алгоритма.

В итоге, базовый класс настроек примет вид, показанный на рисунке ?.

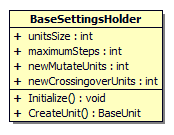


Рисунок ? – базовый класс настроек генетического алгоритма

* 1. Проектирование параллельной части генетического алгоритма

Использование стандарта MPI для реализации параллельной части алгоритма требует организации приёма и передачи данных между узлами. Для решения данной задачи применим паттерн «Упаковка» и определим небольшой класс, который будет унаследован любым классом, если будет необходимо передавать данный класс на другой узел. Например, вызов метода отправки сообщения в классе настроек вызовет эти же методы применимо к хранимым в настройках данным.

В классе будет присутствовать функции приёма и передачи, которые необходимо реализовать под конкретный класс, наследующий данный класс (рисунок ?).

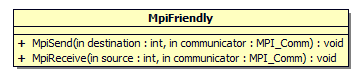


Рисунок ? – Класс коммуникации по стандарту MPI

Также, для внешнего управления отправкой особей популяциями, необходимо сделать публичные функции отправки приёма особи по её номеру.

Подводя промежуточный итог, получается следующая диаграмма классов параллельного генетического алгоритма, представленная на рисунке ?.

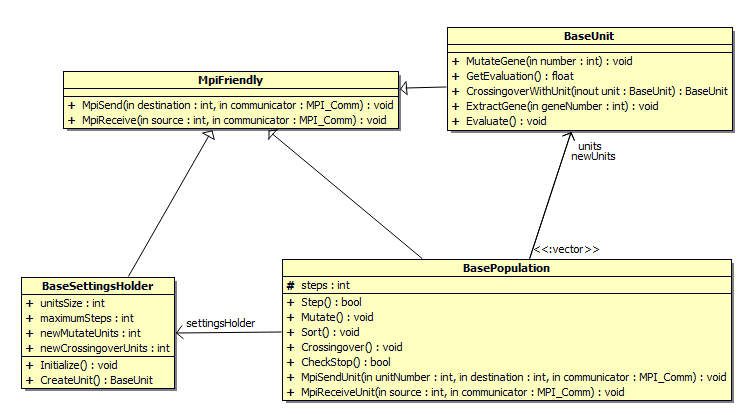


Рисунок ? – диаграмма базовых классов

Полученные базовые классы будут далее использованы для проектирования классов решения задачи раскроя.

* 1. Проектирование решения задачи раскроя

Основным классом, реализующим решение задачи, является класс особи. Рассмотрим необходимые вспомогательные классы, которые будут использованы при решении задачи раскроя.

В первую очередь, рассмотрим саму задачу. Поскольку решаемая задача находится в двух измерениях, потребуется класс точки с двумя координатами, которые будут реализовывать вершины контура фигуры. Класс фигуры будет содержать точки вершин и вспомогательные данные, например параллельный осям ограничивающий параллелепипед. Классы будут иметь вид, представленный на рисунке ?.

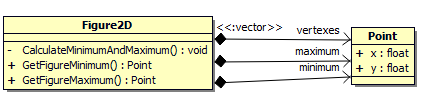


Рисунок ? – Диаграмма классов фигур и точек

При решении поставленной задачи понадобится вращать фигуры относительно начала координат вершин контура, а также проверять пересечение фигур при размещении их на полубесконечной полосе. Добавим эти функции на диаграмму и получим классы, представленные на рисунке ?.

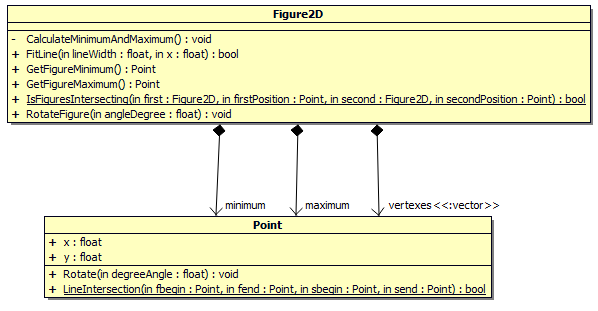


Рисунок ? – диаграмма классов фигур и точек с дополнительными функциями

Далее рассмотрим структуру генов и способ вычисления пригодности. Вычисление пригодности будет проходить следующим образом: повернутые фигуры будут по очереди размещаться на заданном расстоянии от вертикальной границы полосы и опускаться вниз до пересечения с краями полосы или другими фигурами. Отсюда следует, что гены будут хранить три переменные: номер фигуры в очереди, угол поворота фигуры и смещение по горизонтали. Порядок генов будет строго определять принадлежность гена к фигуре.

Набор фигур не будет изменяться с течением времени и будет общим для всех особей, поэтому целесообразно организовать к нему статичный доступ внутри класса особей, который будет наследовать функции базового класса особей. Для вычисления пригодности создадим отдельный класс, который будет выполнять все необходимые расчеты. Диаграмма классов генов, особей и класса расчета представлена на рисунке ?.

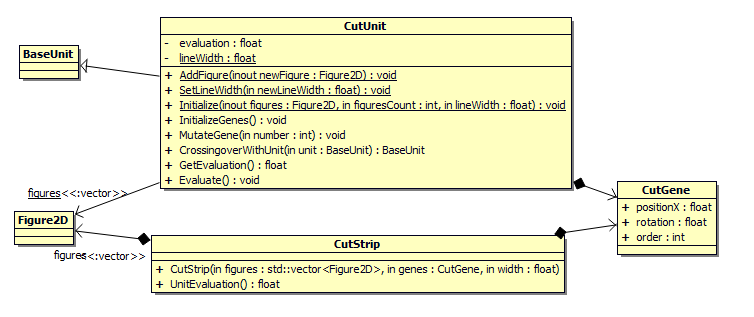


Рисунок ? – диаграмма классов особей, класса вычисления пригодности и генов

Проектирование классов для алгоритма завершено и теперь можно перейти к реализации.

1. РАЗРАБОТКА ПАРАЛЛЕЛЬНОГО ГЕНЕТИЧЕСКОГО АЛГОРИТМА
   1. Анализ структуры приложения

Для удобства разработки приложения и работы с ним было решено отделить вычислительную часть от графического интерфейса анализа результатов. Таким образом, параллельный генетический алгоритм будет реализован на языке программирования С++ с использованием технологий OpenMP, а анализатор полученных решений будет реализован на языке программирования Visual C#.

* 1. Разработка структуры генетического алгоритма

В поставленной задаче нам необходимо разместить на ограниченной плоскости фигуры, состоящие из соединённых линиями точек. Таким образом, местоположение и угол поворота каждой фигуры будут являться генами нашего алгоритма. Выделим данные свойства в класс GA\_gene.

Далее необходимо определить вид особей. Поскольку особь должна содержать набор генов, которые будут определять её пригодность, то она должна содержать в себе все гены, которые будут использованы в генетическом алгоритме. Следовательно, в выделенный класс особи GA\_unit будут записаны массив генов, содержащий экземпляры класса GA\_gene и значение пригодности данной особи.

Последним элементом, который необходимо определить для генетического алгоритма, является вид популяции. Выделим класс GA\_population и запишем в него массив особей, содержащий экземпляры класса GA\_unit. После вычисления приспособленности особей будем производить сортировку по убыванию, что позволит говорить о том, что самая первая особь нашего массива является самым приспособленным индивидом нашей популяции.

Последним элементом генетического алгоритма, который необходимо определить, является функция пригодности. Будем считать, что сумма самых нижних точек фигур является пригодностью данной особи. Чтобы исключить возможные ошибки расположения фигур, такие как наложение друг на друга и расположение за полотном, будем считать их количество и умножать сумму нижних точек на это значение. Таким образом, данная операция позволит отсеивать неудачные расположения фигур, увеличивая их пригодность и уменьшая шансы на выживание.

Для наглядности полученной структуры изобразим диаграмму классов полученного алгоритма(рисунок 7).

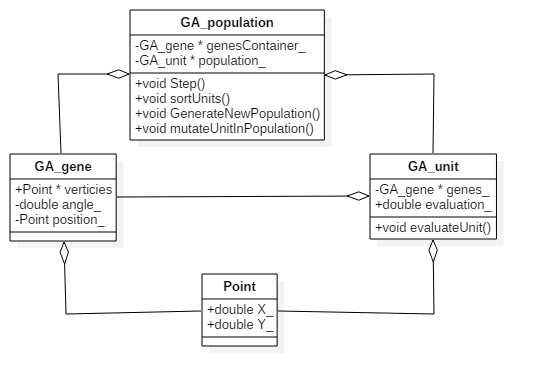


Рисунок 7 – Диаграмма классов генетического алгоритма

Теперь, когда мы определили все элементы генетического алгоритма, можно приступить к его реализации на языке С++.

* + 1. Реализация класса гена

Класс гена называется GA\_gene и будет представлять собой фигуру, хранящуюся как массив точек на плоскости, её расположение на плоскости и её угол поворота(рисунок 8).

private:

double angle\_ = 0;// в градусах

Point position\_;

public:

int size\_ = 0;

Point \* vertices\_ = new Point[0];

Point maxXY\_;

Point minXY\_;

Рисунок 8 – Данные, хранимые классом GA\_gene

Воспользуемся вспомогательным классом Point для хранения координат на плоскости. Составим из него массив, который будет хранить нашу фигуру. Также используем его для хранения местоположения фигуры на плоскости. Будем считать, что фигура хранится с углом поворота 0 градусов, для вычислений будем поворачивать фигуру в копии гена.

Для хранения угла поворота фигуры зададим переменную типа double. Для хранения размера массива используем переменную типа int.

Для сокращения вычислений данные о минимальных и максимальных величинах координат будут храниться в полях класса Point. Для вычисления этих значений понадобятся 4 функции, вычисляющие минимальные и максимальные координаты по обеим осям.

Для тестирования разрабатываемого алгоритма реализуем конструктор, позволяющий получать правильные n-угольники заданного радиуса. В качестве тестируемой модели выбраны треугольники, так как легко и однозначно анализируются на оптимальность расположения на полотне.

GA\_gene(){

size\_ = 3;

delete[] vertices\_;

vertices\_ = new Point[size\_];

double rad = 1;

for (int i = 0; i < size\_; i++){

double step = 2 \* 3.1415 / (double)size\_;

vertices\_[i] = Point(rad\*cos(step\*i), rad\*sin(step\*i));

}

double newAngle = rand() % 360;

angle\_ = newAngle;

position\_ = Point(0, 0);

}

Рисунок 9 – Конструктор генов, содержащих треугольники

Для вычисления фактического положения точек, задающих фигуру, реализуем функции поворота точки относительно центра координат фигуры, и используем эту функцию в методе разворота всей фигуры.

Point rotatePoint(Point p, double angle){

double pi = 3.14159265358979323846;

double cosinus = cos(angle\*pi / 180.0);

double sinus = sin(angle\*pi / 180.0);

Point tmp = p;

p.X\_ = tmp.X\_\*cosinus - tmp.Y\_\*sinus;

p.Y\_ = tmp.X\_\*sinus + tmp.Y\_\*cosinus;

return p;

}

void rotatePointArray(){

for (int i = 0; i < size\_; i++){

Point checking = vertices\_[i];

vertices\_[i] = rotatePoint(vertices\_[i],angle\_);

}

}

Рисунок 10 – Методы поворота фигур

Для алгоритма также потребуется метод, осуществляющий мутацию гена. Данный метод изменяет координаты фигуры или её угол поворота на определённую константой величину.

void mutate(int type){

switch (type){

case 0:{

double x = position\_.X\_ + (float)(rand() % 10) / 5.0 - 1;

double y = position\_.Y\_ + (float)(rand() % 10) / 5.0 - 1;

position\_ = Point(x, y);

angle\_ = ((int)angle\_ + (rand() % 10 - 5) + 360) % 360;

};

case 1:{

angle\_ = ((int)angle\_ + (rand() % 10 - 5) + 360) % 360;

};

case 2:{

double x = position\_.X\_ + (float)(rand() % 1000) / 50000.0 - 0.01;

double y = position\_.Y\_ + (float)(rand() % 1000) / 50000.0 - 0.01;

position\_ = Point(x, y);

angle\_ = ((int)angle\_ + (rand() % 10 - 5) + 360) % 360;

};

case 4:{

angle\_ = ((int)angle\_ + (rand() % 180 - 90) + 360) % 360;

}

}

}

Рисунок 11 – Метод мутации

Кроме того, потребуется переопределить оператор копирования, так как в классе имеется массив, под который выделяется память, а также деструктор, в котором эта память освободится.

Остальные реализованные функции носят вспомогательный характер и приведены в приложении А.

* + 1. Реализация класса особи

Класс GA\_unit будет представлять особь, поэтому его основной задачей будет хранение массива генов.

private:

int geneSize\_ = 0;

GA\_gene \* genes = new GA\_gene[0];

GA\_gene \* bufgenes = new GA\_gene[0];

double fieldSize\_=4;

public:

int errors\_=0;

double evaluation\_ = 0;

Рисунок 12 – Переменные класса GA\_unit

Далее классу понадобиться переменная типа int для хранения размера массива генов и переменная типа double для хранения пригодности особи. Дополнительно введем переменную для хранения количества ошибок типа int. Для расчетов также потребуется хранить значение ширины полосы типа double.

Для заполнения массива генов потребуется метод, добавляющий гены в массив. Данный метод позволит с лёгкостью подстраиваться под разное количество оптимизируемых фигур.

void addGene(GA\_gene newGene){// добавление гена

newGene.randomizePosition(fieldSize\_);

geneSize\_++;

delete[] bufgenes;

bufgenes = new GA\_gene[geneSize\_];

GA\_gene \* bufdelete = genes;

GA\_gene \* buf = new GA\_gene[geneSize\_];

for (int i = 0; i < geneSize\_ - 1; i++){

buf[i] = genes[i];

}

buf[geneSize\_ - 1] = newGene;

genes = buf;

delete[] bufdelete;

evaluation\_ = evaluateUnit();

while (errors\_ > 0) {

genes[geneSize\_ - 1].randomizePosition(fieldSize\_);

evaluation\_ = evaluateUnit();

}

}

Рисунок 13 – Функция добавления гена в особь

Добавление генов исключает появление таких хромосом, в которых существуют ошибки, так как ситуации, в которых вся популяция состоит из особей с ошибками, среди которых начинается конкуренция с низкой вероятностью выхода из данного положения, не приводят к правильному решению.

Далее нам потребуются методы генетического алгоритма, такие как мутация и кроссинговер. Метод мутации будет вызывать у случайного гена функцию мутации. Функция кроссинговера будет принимать два параметра: другую особь и номер обмениваемого гена. Заданный таким образом метод позволит с лёгкостью варьировать способ скрещивания, однако особь необходимо скопировать в новую переменную, так как операции выполняются над вызвавшей функцию особью.

void KrossingoverByGeneNumber(GA\_unit secondParent, int geneNumber){

GA\_gene tmp = GA\_gene(secondParent.genes[geneNumber]);

genes[geneNumber] = tmp;

}

Рисунок 14 – Метод кроссинговера

Далее необходимо определить функцию, рассчитывающую пригодность.

float evaluateUnit(){

double result = 0;

for (int i = 0; i < geneSize\_; i++){

GA\_gene rotatedGene = genes[i];

rotatedGene.updateGene();

result += rotatedGene.maxXY\_.Y\_;

}

int errors = 0;

for (int i = 0; i < geneSize\_; i++){

GA\_gene rotatedGene = genes[i];

rotatedGene.updateGene();

if (rotatedGene.maxXY\_.Y\_ < 0) {

errors++;

result -= rotatedGene.maxXY\_.Y\_ \* 10;

}

if (rotatedGene.maxXY\_.X\_ > fieldSize\_) {

errors++;

result += (rotatedGene.maxXY\_.X\_ - fieldSize\_)\*10;

}

if (rotatedGene.minXY\_.X\_ < 0) {

errors++;

result -= rotatedGene.minXY\_.X\_ \* 10;

}

if (rotatedGene.minXY\_.Y\_ < 0) {

errors++;

result -= rotatedGene.minXY\_.Y\_ \* 10;

}//check units out of range

}

for (int i = 0; i < geneSize\_ - 1; i++){

for (int j = i + 1; j < geneSize\_; j++){

GA\_gene rotatedGene1 = genes[i];

GA\_gene rotatedGene2 = genes[j];

rotatedGene1.updateGene();

rotatedGene2.updateGene();

if (UnitsIntersection(rotatedGene1, rotatedGene2)){

errors++;// check intersection

}

}

}

errors\_ = errors;

return result \* (errors+1);

}

Рисунок 15 – Функция расчета пригодности

Так как необходимо считать не только сумму максимальных координат, но и количество ошибок, нам потребуются вспомогательные функции, которые помогут определить, сколько пересечений фигур и их выходов за пределы полотна присутствуют в данном расположении особей. В качестве проверки на пересечение фигур вводится функция LineIntersection(), которая будет проверять две выбранные линии на пересечение. В случае пересечения эта функция будет возвращать true, иначе false. Операция проверки на пересечение линий будет использована в функции UnitsIntersection(), которая будет перебирать все линии двух выбранных особей и булево складывать результат проверки. На выходе данная функция вернёт значение, полученное в результате булевого сложения. Таким образом, применив данную функцию ко всем парам генов и посчитав положительные результаты, можно установить количество ошибок оцениваемой раскладки фигур. К вспомогательным функциям также относятся методы печати особи в файл и консоль. Описанные функции представлены в приложении Б.

В методах класса необходимо переопределить оператор копирования, так как в классе присутствует массив генов, для которого выделяется память, и деструктор, в котором выделенная память очищается.

* + 1. Реализация класса популяции

Данный класс представляет собой популяцию, следовательно, главной его задачей является хранение особей и применение к ним операций генетического алгоритма.

private:

double fieldSize\_;

int steps\_;

int size\_ = 0;// размер популяции

GA\_unit \* units\_;

GA\_gene \* genesContainer\_;// гены популяции

int genesContainerSize\_;

int elites\_ = 0;

bool square\_ = false;

public:

double mutateFrequency\_ = 0;

double best;

Рисунок 16 – Переменные класса GA\_population

Для хранения особей понадобится массив типа GA\_unit и переменная типа int для хранения его размера. Также, для удобства генерирования популяции необходимо хранить гены особей для дальнейшей передачи сгенерированным индивидам. Для этого массива потребуется переменная, хранящая его размер.

В методах необходимо переопределить оператор копирования, так как в классе присутствует два массива, под которые выделена память. В переопределённом деструкторе необходимо эту память освободить.

Определим метод, который будет содержать операции одного цикла. В него поместим следующие функции: генерация новых особей, мутация особей и сортировка особей. Данный метод будет варьироваться в зависимости от реализуемого алгоритма, его варианты будут рассмотрены далее.

Рассмотрим каждую функцию итерации алгоритма подробнее.

Операция мутации будет применяться к определённому константой количеству особей. После выбора индивида для него будет вызван метод мутации, а затем метод пересчёта пригодности.

void mutateUnitInArray(GA\_unit \* array, int size, double frequency){

for (int i = 0; i < size \* frequency; i++){

int k = rand() % size;

array[k].MutateUnit();

}

}

Рисунок 17 – Метод мутации

Операция сортировки будет получать массив особей и сортировать его по возрастанию, в качестве критерия сортировки выступит приспособленность особи. После применения данного метода будет получен отсортированный массив, первый элемент которого будет лучшей особью.

void sortUnits(GA\_unit \* units, int size){

int tmp;

for (int i = 0; i < size - 1; i++){

tmp = i;

for (int j = i + 1; j < size; j++){

if (units[j].evaluation\_ < units[tmp].evaluation\_) tmp = j;

}

if (tmp != i){

GA\_unit buf = units[i];

units[i] = units[tmp];

units[tmp] = buf;

}

}

}

Рисунок 18 – Функция сортировки массива особей

Операция генерации новых особей создаёт массив, в который будут складываться новые особи популяции. Новые особи получаются путем применения оператора скрещивания к выбранным парам особей. Когда массив будет заполнен, в конец популяции вместо неэлитных особей записываются лучшие индивиды массива потомков.

* + 1. Реализация последовательного генетического алгоритма

Последовательный алгоритм работает при вызове метода Step().

void Step(){

GenerateNewPopulations(units\_, size\_, elites\_);

sortUnits();

}

Рисунок 19 – Последовательный метод итерации

Первая функция метода вызывает функцию генерации новых особей популяции и передаёт в неё массив особей, размер этого массива и количество элитных особей.

void GenerateNewPopulations(GA\_unit \* unitsArray, int size, int elites){

randomGeneKrossingover(unitsArray, size, elites, mutateFrequency\_);

}

Рисунок 20 – Функция генерации новых особей

Данная функция вызывает выбранный метод генерации новых особей, в данном случае это случайный одноточечный кроссинговер. Рассмотрим его подробнее.

void randomGeneKrossingover(){

GA\_unit \* children = new GA\_unit[size\_ - elites\_];

for (int i = 0; i < size\_-elites\_; i++){

int first = 0, second = 0;

while (first == second){

first = rand() % elites\_;

second = elites\_ + rand() % (size\_-elites\_);

}

GA\_unit candidate = GA\_unit(units\_[first]);

int geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(units\_[second], geneExchangeNumber);

children[i] = candidate;

}

mutateUnitInArray(children, size\_-elites\_,mutateFrequency\_);

sortUnits(children,size\_-elites\_);

for (int i = elites\_; i < size\_; i++){

units\_[i] = children[i-elites\_];

}

delete[] children;

}

Рисунок 21 – Случайный одноточечный кроссинговер

В данной реализации одноточечного кроссинговера выбираются две случайные особи, одна из которых принадлежит к числу элитных особей, а вторая не принадлежит. Далее случайно определяется ген, который будет взят у второй особи на замену этому гену в первой особи. Затем следует вызов метода замены гена и запись полученной особи в массив особей-потомков.

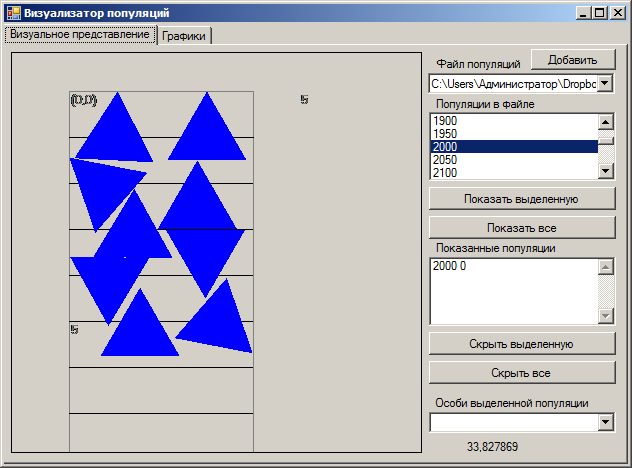
После заполнения массива особей-потомков алгоритмом, записанным выше, данный массив передаётся в метод мутации, где у определённой части особей происходят случайные изменения генов. Затем особи-потомки сортируются и заменяют неэлитные особи популяции. Далее происходит сортировка полученной популяции.

Метод Step()вызывается циклически необходимое количество раз. При необходимости можно вызывать вспомогательную функцию printPopulation() для фиксирования состояния популяции в файл для дальнейшего изучения в визуализаторе.

* 1. Описание визуализатора решений

Для наглядного представления работы алгоритма был разработан визуализатор решений. Основными функциями данной программы являются вычисление средних и лучших величин пригодности популяций и построение их графиков, и построение графических изображений особей популяции.

Рассмотрим интерфейс визуализатора.

  
Рисунок 22 – Вкладка визуального представления

При запуске программы открывается вкладка визуального представления. На данной вкладке расположены графическое окно отображения генов, элементы загрузки и выбора популяции, а также средства управления отображением особей в графическом окне.

Работа с программой начинается с добавления файла популяций кнопкой «Добавить». Затем в выпадающем списке выбирается одна из добавленных популяций. Далее из файла выбирается популяция и перемещается в список показанных популяций кнопкой «Показать выделенную». По умолчанию сразу будет отражена нулевая особь добавленной популяции, однако можно изменить номер особи в выпадающем списке «Особи выделенной популяции». Кнопка «Скрыть выделенную» позволяет убрать изображение выделенной популяции. Кнопка «Показать все» добавит все популяции файла в список «Показанные популяции», что вызовет отрисовку их нулевых особей в графическом окне. Кнопкой «Скрыть все» можно очистить список показанных популяций и графическое окно.

Графическое окно обладает функциями масштабирования при помощи колеса мыши, а также навигации при помощи перемещения курсора мыши с зажатой левой кнопкой.

Рассмотрим вкладку «Графики»

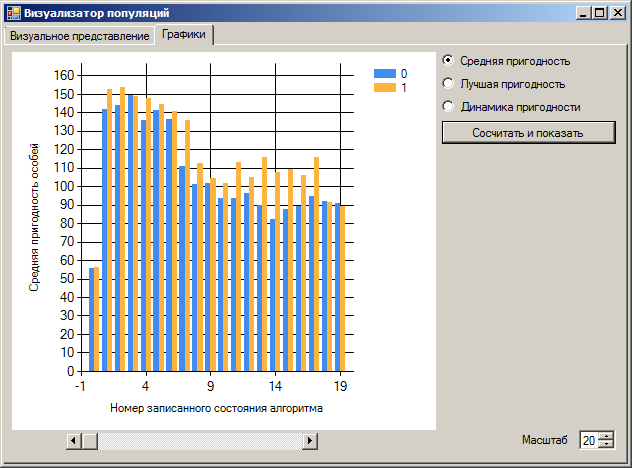


Рисунок 23 – Вкладка «Графики»

На данной вкладке расположено окно построения графиков и управляющие построением элементы. Здесь предлагается выбрать какой график необходимо построить и количество записей, отображаемых на нём. Далее следует нажать кнопку «Сосчитать и показать», после чего программа построит выбранный график с выбранным масштабом. Для удобства просмотра файлов с большим количеством популяций под окном графиков расположена полоса прокрутки, позволяющая просматривать все популяции в файле.

* 1. Параллельный генетический алгоритм

Распараллеливание реализованного генетического алгоритма происходит путём реализации параллельных версий метода Step(). В данной работе было реализовано четыре схемы распараллеливания генетического алгоритма. Рассмотрим реализацию каждой из них.

* + 1. Описание реализации схемы «мастер-раб» с делением на подпопуляции

Рассмотрим метод Step\_threads() представляющий данную схему.

void Step\_threads(){

int thrd;

#pragma omp parallel private(thrd)

{

int thrds = omp\_get\_num\_threads();

int \* sizes = new int[thrds+1];

int lastsize = size\_;

GA\_unit \*start;

sizes[0] = 0;

for (int i = 1; i < thrds; i++){

sizes[i] = sizes[i-1]+ (int)(size\_ / (double)thrds);

}

sizes[thrds] = lastsize;

thrd = omp\_get\_thread\_num();

GenerateNewPopulations(&units\_[sizes[thrd]], sizes[thrd+1]-sizes[thrd], (int) elites\_/thrds

}

sortUnits();

}

Рисунок 24 – Метод Step\_threads()

В данном методе популяция делится на количество запущенных потоков. Каждый поток производит все операции генетического алгоритма над своей частью популяции. Таким образом, образованные потомки имеют родителей только из своей части популяции. После завершения выполнения всех потоков популяция сортируется.

* + 1. Описание реализации схемы «мастер-раб» с делегированием вычисления пригодности

В данной схеме распараллеливается только функция вычисления пригодности, так как в худшем случае она является самой вычислительно сложной функцией в программе. Рассмотрим метод StepMasterSlaveEvaluation(), в котором реализовано распараллеливание.

void StepMasterSlaveEvaluation(){

GA\_unit \* children = new GA\_unit[size\_];

for (int i = 0; i < size\_; i++){

GenerateNewPopulationsSingleChild(units\_, children, i, size\_, elites\_);

}

mutateUnitInArray(children, size\_, mutateFrequency\_);

int p;

#pragma omp parallel private (p)

{

int thrds = omp\_get\_num\_threads();

int thrd = omp\_get\_thread\_num();

for (p = thrd; p < size\_; p+=thrds)

{

children[p].recalcEvaluation();

}

}

sortUnits(children, size\_);

for (int i = 0; i < size\_ - elites\_; i++)

units\_[i + elites\_] = children[i];

delete[] children;

sortUnits();

}

Рисунок 25 – Метод StepMasterSlaveEvaluation()

В приведённом методе операции создания потомков и мутации происходят до вычисления пригодности в параллельных потоках. После параллельного блока происходит сортировка и создание новой популяции.

* + 1. Описание реализации схемы «мастер-раб» с делегированием вычисления всех функций алгоритма

Рассмотрим метод Stepp() представляющий данную схему.

void Stepp(){

int thrd;

GA\_unit \* children = new GA\_unit[size\_];

#pragma omp parallel private(thrd)

{

int thrds = omp\_get\_num\_threads();

int \* sizes = new int[thrds + 1];

int lastsize = size\_;

GA\_unit \*start;

sizes[0] = 0;

for (int i = 1; i < thrds; i++){

sizes[i] = sizes[i - 1] + (int)(size\_ / (double)thrds);

}

sizes[thrds] = lastsize;

thrd = omp\_get\_thread\_num();

for (int i = sizes[thrd]; i < sizes[thrd + 1]; i++){

GenerateNewPopulationsSingleChild(units\_, children, i, size\_, elites\_);

}

mutateUnitInArray(&children[sizes[thrd]], sizes[thrd + 1] - sizes[thrd], mutateFrequency\_);

for (int i = sizes[thrd]; i < sizes[thrd + 1]; i++){

children[i].recalcEvaluation();

}

}

sortUnits(children, size\_);

for (int i = 0; i < size\_ - elites\_; i++){

units\_[i + elites\_] = children[i];

}

sortUnits();

delete[] children;

}

Рисунок 26 – Метод Stepp()

Структура данного метода схожа с предыдущей реализацией, однако здесь каждый поток не имеет своей части популяции. Вместо этого, задачей каждого потока является создание определённого количества особей-потомков, мутация и вычисление их пригодности.

* 1. Тестирование алгоритма

Тестирование алгоритма проводилось на компьютере со следующими характеристиками: материнская плата Gigabyte GA-H61MA-D3V (Rev. 2.0), процессор Intel® Core™ i5-2320 3.00GHz, оперативная память HyperX Fury Red Series 16GB (2 x 8GB) 240-Pin DDR3 SDRAM, видеокарта NVIDIA GeForce GTX 560.

Для тестирования алгоритма были проведены серии запусков алгоритма с разными размерами популяций и замерена их скорость и эффективность работы. Далее представлен анализ данных, полученных при исследовании последовательного алгоритма и сравнение эффективности и быстродействия различных параллельных реализаций.

* + 1. Тестирование эффективности последовательного алгоритма

Исследуем эффективность последовательного алгоритма. Для этого рассмотрим сравнение итоговой пригодности на разных количествах шагов.

Рисунок 27 – Зависимость пригодности от количества шагов

На данном рисунке видно, что показание пригодности финальной особи уменьшается с увеличением количества шагов. Рассмотрим несколько гистограмм, показывающих динамику лучшей особи популяции.

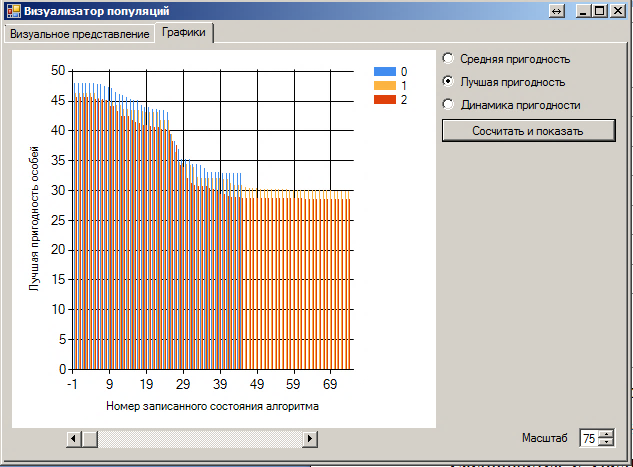


Рисунок 28 – Сравнение динамики пригодности популяций на 1000(1), 5000(2) и 10000(3) шагов

На рисунке видно, что популяции сохраняют некоторое расстояние между своими лучшими особями. Это происходит из-за различий в генерируемых начальных популяциях.

Исследуя гистограмму можно заключить, что для успешности алгоритма требуется сгенерировать удачную популяцию.

Рассмотрим расположение генов в лучших особях популяции на разных шагах.

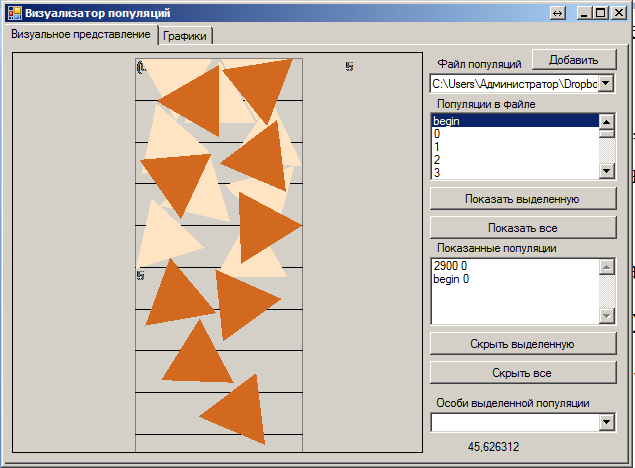


Рисунок 29 – Лучшие особи на 0(оранжевые фигуры) и 2900(бежевые фигуры) шагах

Данный рисунок иллюстрирует, как преобразуются гены за 2900 шагов. Можно заметить, что положение фигур на 2900 шаге в некоторой степени напоминает сжатое положение треугольников на 0 шаге. Это демонстрирует высокий показатель стремления реализованной функции пригодности к минимуму, что является отрицательным свойством данной реализации. Однако с другой стороны, популяция очень быстро достигает локального минимума, что позволяет при множественных запусках алгоритма перейти от количества к качеству. Рассмотрим положение фигур на 2900 и 10000 шагах.

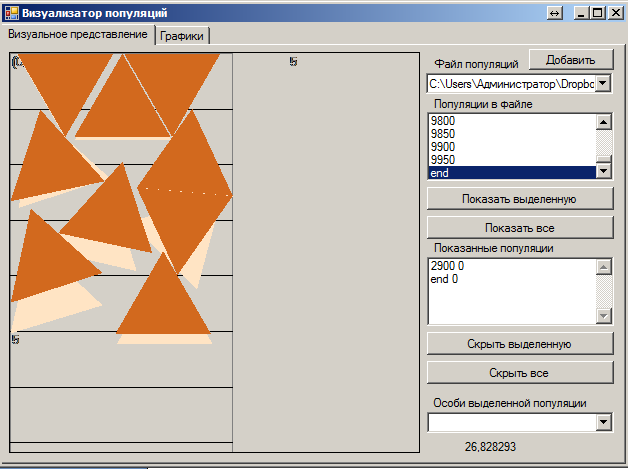


Рисунок 30 – Сравнение положения фигур особей популяций на 2900(бежевые фигуры) и 10000(оранжевые фигуры) шагах

На рисунке видно, что на промежутке от 2900 до 10000 шагов алгоритм улучшает достигнутое положение фигур, однако не способен его изменить. Отсутствие возможности возникновения особей с иным раскладом генов обусловлено высокой конкуренцией среди индивидов популяции, что влечёт отсутсвие генетического разнообразия. Данный эффект реализованных функций генетического алгоритма имеет свои плюсы и минусы. Отрицательным эффектом является отсутствие генетического разнообразия. Положительный эффект помогает алгоритму быстрее достигать выбранного в процессе отбора локального минимума.

Таким образом, реализованный генетический алгоритм имеет высокую скорость сходимости к локальному минимуму. Данная особенность помогает быстрее достигать минимальных значений, однако отрицательно сказывается на возможности алгоритма искать разные решения.

* + 1. Сравнение быстродействия

В процессе тестирования изучалось насколько быстро выполняются 10000 итераций. Заданное число операций было выбрано потому, что за 10000 повторений алгоритма становиться отчётливо виден результат его работы, который, тем не менее, не является оптимальным.

Всего было исследовано четыре описанных ранее реализаций шага алгоритма. Каждый из них был протестирован на 100, 200 и 400 особей в популяции. Рассмотрим получившийся график времени работы алгоритмов.

Рисунок 31 – График сравнения времени работы алгоритма

Как следует из рисунка, все реализованные алгоритмы показывают ускорение. Сравнение графиков Stepp() и StepMSE() показывает, что ускорение распределением вычисления функции пригодности падает с ростом популяции, так как возрастает число операций с памятью, проводимых в функциях мутации и скрещивания, которые распараллелены в методе Stepp().

Рассмотрим график ускорения параллельных алгоритмов к последовательному.

Рисунок 32 – График ускорения

Данный график показывает особенности каждой реализации. Методы Step\_threads() и Stepp() показывают рост ускорения с возрастанием количества особей популяции, однако метод StepMSE() теряет свою эффективность из-за распараллеливания только одной функции.

* + 1. Сравнение эффективности

Рассмотрение эффективности реализаций Stepp() и StepMSE() не требуется, так как в данных методах используется стандартные структуры генетического алгоритма. Функция Step\_threads() имеет отличную от классической структуру. Рассмотрим гистограмму эффективности.

Рисунок 33 – Сравнение эффективности

Данная гистограмма показывает, что метод Step\_threads() значительно проигрывает по эффективности классической схеме, которая, в свою очередь, показывает рост эффективности с увеличением размера популяции.

* + 1. Выводы из результатов тестирования

Исходя из результатов тестирования, можно сделать следующие выводы:

* реализованные методы ускоряют работу алгоритма с разной эффективностью;
* схема, реализованная в методе Step\_threads() имеет более низкую эффективность отыскания минимума по сравнению с классической схемой;
* рост популяции положительно влияет на эффективность поиска лучшего решения.

Таким образом, исследование показало эффективность распараллеливания генетического алгоритма. Изучение схемы «Мастер – раб» с разделением популяции на подпопуляции показало отрицательную эффективность в поиске решения. Из этого следует, что данная схема не подходит для решения задачи реализованным алгоритмом. Причина этого кроется в функции пригодности, которая оказалось неудачна по отношению к схеме «Мастер – раб».

Дальнейшее развитие алгоритма может пойти по нескольким путям. Результат тестирования показал, что алгоритм быстро сходится к локальному минимуму, следовательно, одним из вариантов усовершенствования программы является поиск возможностей дополнительного ускорения этой сходимости. Альтернативным вариантом является изменение функций алгоритма таким образом, чтобы поиск не застревал в локальном минимуме, а искал глобальное решение.

Тестирование алгоритма проводилось на компьютере с 4-х ядерным процессором, что не позволило оценить весь потенциал параллельной реализации. Дальнейшее исследование алгоритма подразумевает запуск на компьютерах с большим количеством ядер и суперкомпьютерах, так как исходя из результатов тестирования, прогнозируется увеличение скорости работы алгоритма.

Заключение

В работе была дана общая характеристика теории генетических алгоритмов, проанализирована поставленная задача, сделан обзор инструментов для создания параллельных генетических алгоритмов и обоснован выбор средств разработки. Был разработан последовательный генетический алгоритм, реализованы параллельные методы для данного алгоритма, проведено тестирование и проанализированы полученные в ходе тестов результаты. Также разработан визуализатор решений, упрощающий анализ выходных данных алгоритма.

В результате проведённой работы были применены методы параллельных вычислений, использованы стандарт OpenMP и объектно-ориентированная парадигма. Выпонены следующие задачи выпускной квалификационной работы:

* дана общая характеристика теории генетических алгоритмов;
* проанализирована поставленная задача, сделан обзор инструментов для создания параллельных генетических алгоритмов и обоснован выбор средств разработки;
* разработан параллельный генетический алгоритм на примере задачи раскроя.

Дальнейшее изучение распараллеливания генетических алгоритмов включает следующие задачи:

* реализация неисследованных схем распараллеливания;
* реализация возможности использования алгоритма на реальных данных, полученных из систем САПР;
* исследование возможности встраивания алгоритма в прикладное ПО.

Тестирование показало, что разные параллельные модификации алгоритма имеют разное ускорение. Модификация «Мастер – раб» показала наилучшее уменьшение времени работы, что позволяет предположить, что с ростом количества ядер эффективность данной реализации продолжит расти.

Одним из перспективных направлений развития алгорима является реализация крупнозернистых схем параллельных генетических алгоритмов с использованием стандарта MPI. Распределение памяти позволит выполнять вычисления на нескольких узлах одновременно, что позволит логически распределить по ним популяции, а структура MPI предоставит инструменты для реализации миграций особей между данными популяциями.

На данный момент реализованные классы могут применяться для дальнейшего исследования возможностей распараллеливания генетических алгоритмов. Описанные в них методы позволяют легко реализовывать различные схемы параллельных генетических алгоритмов, что упрощает дальнейшее исследование.

В итоге проделанной работы были выполнены все поставленные задачи. Достигнут положительный результат распараллеливания генетических алгоритмов.

Список использованных источников и литературы

1. Антонов А. С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. – Издательство Московского университета М.:, 2012. – С. 344.
2. Библиотека учебных материалов Parallel.ru [Электронный ресурс]. – режим доступа: http://parallel.ru/info/parallel/
3. Бьерн Страуструп Язык программирования C++. Специальное издание. Пер. с англ. – М.: Издательство Бином, 2011 – 1136 с: ил.
4. Васильев А. Н. C#. Объектно-ориентированное программирование. Учебный курс: – СПб.: ПИТЕР, 2012 – 320 с.
5. Введение в язык C# и .NET Framework [Электронный ресурс]. – режим доступа: https://msdn.microsoft.com/ru-ru/library/z1zx9t92.aspx (Дата обращения: 10.06.2016)
6. Генетические алгоритмы [Электронный ресурс]. – режим доступа: http://www.aiportal.ru/articles/genetic-algorithms/1/
7. Генетические алгоритмы на примерах решения задач раскроя [Электронный ресурс]. – режим доступа: http://cyberleninka.ru/article/n/geneticheskie-algoritmy-na-primerah-resheniya-zadach-raskroya
8. Генетический алгоритм. Просто о сложном. [Электронный ресурс]. – режим доступа: https://habrahabr.ru/post/128704/
9. Гергель В. П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем: учебник для студ. Вузов: Нижний Новгород: НГУ, 2010. – 539 с.
10. Гладков Л. А. Генетические алгоритмы: Гладков Л. А., Курейчик В. В., Курейчик В. М. / Под ред. В. М. Курейчика. – 2 изд., испр. и доп. – М.: ФИЗМАТЛИТ, 2006 – 320 с.
11. Гома, Х. UML. Проектирование систем реального времени, параллельных и распределённых приложений: Пер. с англ. – М.: ДМК Пресс, 2011. – 704 с.; ил.
12. Курейчик, В. М. Параллельный генетический алгоритм. Модели и проблемы построения. / В. М. Курейчик, Д. С. Кныш. – Коломна: Изд-во УГТУ, 2009. – С.1-3.
13. Научная электронная библиотека elibrary [Электронный ресурс]. – режим доступа: http://elibrary.ru/defaultx.asp
14. Немнюгин С. А., Параллельное программирование для многопроцессорных вычислительных систем: / Немнюгин С. А., Стесик О. – СПб.: "БХВ", 2002 – 396 с.
15. Панченко Т. В. Генетические алгоритмы : учебно-методическое пособие / под ред. Ю. Ю. Тарасевича. – Астрахань : Издательский дом «Астраханский университет», 2007. – 87 [3] с.
16. Практикум по методам параллельных вычислений: учебник для студ. Вузов / А. В. Старченко [и др.] ; ред. А. В. Старченко : томский гос. ун-т. – М. : МГУ, 2010. – 199 с.
17. Рутовская Д. Нейронные сети, генетические алгоритмы и нечёткие системы / Пилиньский М., Рутовский Л. : Пер. с польск. И. Д. Рудинского. –М.: Горячая линия – Телеком, 2006. – 452с.: ил.
18. Что такое технология Java и каково ее применение? [Электронный ресурс]. – режим доступа: <https://www.java.com/ru/download/faq/whatis_java.xml> (Дата обращения: 10.03.2016)
19. Шевченко В. Н. Линейное и целочисленное линейное программирование: / Шевченко В. Н., Золотых Н.Ю. – Нижний Новгород: Изд-во Нижегородского госуниверситета им. Н.И. Лобачевского, 2004. – 154 с.
20. Язык программирования С++ [Электронный ресурс]. – режим доступа: http://altcode.ru/c-plus/ (Дата обращения: 10.03.2016)
21. Cantu-Paz E. A Survey of Parallel Genetic Algorithms// IlliGAL Report, 1997.
22. C# [Электронный ресурс]. – режим доступа:   
    https://msdn.microsoft.com/ru-ru/library/kx37x362.aspx (Дата обращения: 10.03.2016)
23. Luke, S. Essentials of Metaheuristics [Electronic resource] / Sean Luke. -2nd Ed. – Department of Computer Science, George Mason University: Lulu, 2013. – 263 p. – Retrieved from http://cs.gmu.edu/~sean/book/metaheuristics/
24. MATLAB – высокоуровневый язык технических расчетов [Электронный ресурс]. – режим доступа: <http://matlab.ru/products/matlab> (Дата обращения: 10.03.2016)

ПРИЛОЖЕНИЕ А

(Справочное)

Код класса GA\_gene

#pragma once

#include "Point.h"

#include <cmath>

#include <stdlib.h>

#include <stdio.h>

#include <string>

#define PI 3.4159265358979323846

using namespace std;

class GA\_gene{//ген особи

private:

double angle\_ = 0;// в градусах

Point position\_;

double minX(){

double tmp = 0;

for (int i = 0; i < size\_; i++){

if (vertices\_[i].X\_ < tmp) tmp = vertices\_[i].X\_;

}

return tmp + position\_.X\_;

}

double maxX(){

double tmp = 0;

for (int i = 0; i < size\_; i++){

if (vertices\_[i].X\_ > tmp) tmp = vertices\_[i].X\_;

}

return tmp + position\_.X\_;

}

double maxY(){

double tmp = 0;

for (int i = 0; i < size\_; i++){

if (vertices\_[i].Y\_ > tmp) tmp = vertices\_[i].Y\_;

}

return tmp + position\_.Y\_;

}

double minY(){

double tmp = 0;

for (int i = 0; i < size\_; i++){

if (vertices\_[i].Y\_ < tmp) tmp = vertices\_[i].Y\_;

}

return tmp + position\_.Y\_;

}

public:

int size\_ = 0;

Point \* vertices\_ = new Point[0];

Point maxXY\_;

Point minXY\_;

// dummy init

GA\_gene(){

size\_ = 3;

delete[] vertices\_;

vertices\_ = new Point[size\_];

double rad = 1;

for (int i = 0; i < size\_; i++){

double step = 2 \* 3.1415 / (double)size\_;

vertices\_[i] = Point(rad\*cos(step\*i), rad\*sin(step\*i));

}

double newAngle = rand() % 360;

angle\_ = newAngle;

position\_ = Point(0, 0);

}

GA\_gene(float fieldsize){

angle\_ = 0;

size\_ = (int)(rand() % 10 + 3);

delete[] vertices\_;

vertices\_ = new Point[size\_];

double rad = rand() % 10;//случайный размер

for (int i = 0; i < size\_; i++){

double step = 2 \* 3.1415 / (double)size\_;

vertices\_[i] = Point(cos(step\*i), sin(step\*i));

}

position\_ = Point((rand() % ((int)fieldsize)), rand() % 100);

minXY\_ = Point(minX(), minY());

maxXY\_ = Point(maxX(), maxY());

}

~GA\_gene(){

if (vertices\_)

delete[] vertices\_;

}

GA\_gene(const GA\_gene &obj)

{

\*this = obj;

}

//перегруженный оператор

GA\_gene& operator=(const GA\_gene& obj){

if (this != &obj){

delete[] vertices\_;

Point \* newVerticies = new Point[obj.size\_];

for (int i = 0; i < obj.size\_; i++)

newVerticies[i] = obj.vertices\_[i];

//std::copy(obj.vertices\_, obj.vertices\_ + obj.size\_, newVerticies);

vertices\_ = newVerticies;

size\_ = obj.size\_;

angle\_ = obj.angle\_;

maxXY\_ = obj.maxXY\_;

minXY\_ = obj.minXY\_;

position\_ = obj.position\_;

}

return \*this;

}

void randomizePosition(int fieldsize){

position\_ = Point(minX() + (rand() % ((int)fieldsize - (int)minX())), rand() % 1000/100.0);

double newAngle = rand() % 360;

//rotatePointArray(newAngle);

angle\_ = newAngle;

//updateGene();

}

// использовать только во временных переменных, вращает фигуру

void updateGene(){

rotatePointArray();

minXY\_ = Point(minX(), minY());

maxXY\_ = Point(maxX(), maxY());

}

Point getPosition(){

return position\_;

}

void setPosition(Point newPosition){

position\_ = newPosition;

//updateGene();

}

double getAngle(){

return angle\_;

}

void setAngle(double a){

//rotatePointArray(a);

angle\_ = a;

//updateGene();

}

void drop(double value){

position\_.Y\_ -= value;

}

void rise(double value){

position\_.Y\_ += value;

}

void mutate(int type){

switch (type){

case 0:{

double x = position\_.X\_ + (float)(rand() % 10) / 5.0 - 1;

double y = position\_.Y\_ + (float)(rand() % 10) / 5.0 - 1;

position\_ = Point(x, y);

angle\_ = ((int)angle\_ + (rand() % 10 - 5) + 360) % 360;

};

case 1:{

angle\_ = ((int)angle\_ + (rand() % 10 - 5) + 360) % 360;

};

case 2:{

double x = position\_.X\_ + (float)(rand() % 1000) / 50000.0 - 0.01;

double y = position\_.Y\_ + (float)(rand() % 1000) / 50000.0 - 0.01;

position\_ = Point(x, y);

angle\_ = ((int)angle\_ + (rand() % 10 - 5) + 360) % 360;

};

case 4:{

angle\_ = ((int)angle\_ + (rand() % 180 - 90) + 360) % 360;

}

}

}

Point rotatePoint(Point p, double angle){

double pi = 3.14159265358979323846;

double cosinus = cos(angle\*pi / 180.0);

double sinus = sin(angle\*pi / 180.0);

Point tmp = p;

p.X\_ = tmp.X\_\*cosinus - tmp.Y\_\*sinus;

p.Y\_ = tmp.X\_\*sinus + tmp.Y\_\*cosinus;

return p;

}

void rotatePointArray(double newAngle){

for (int i = 0; i < size\_; i++){

Point checking = vertices\_[i];

vertices\_[i] = rotatePoint(vertices\_[i], newAngle - angle\_);

}

}

void rotatePointArray(){

for (int i = 0; i < size\_; i++){

Point checking = vertices\_[i];

vertices\_[i] = rotatePoint(vertices\_[i],angle\_);

}

}

void setPositionZero(){

position\_ = Point(0, 0);

}

void printGene(){

printf("Unit position: (%.2f, %.2f)\n", position\_.X\_, position\_.Y\_);

}

void printGeneToFile(string &result){

result += to\_string(angle\_)+"\n";

result += to\_string(size\_)+"\n";

result += to\_string(position\_.X\_) + " " + to\_string(position\_.Y\_);

for (int i = 0; i < size\_; i++){

result += to\_string(vertices\_[i].X\_)+" "+to\_string(vertices\_[i].Y\_);

}

result+="\n";

}

void printGeneToFileUnit(string &result){

result += to\_string(angle\_) + "\n";

result += to\_string(position\_.X\_) + " " + to\_string(position\_.Y\_) + "\n";

}

void printFullGene(){

printf("Unit size=%d, angle=%f position:\n", size\_, angle\_);

printf(" (%f, %f)\nvertices: ", position\_.X\_, position\_.Y\_);

for (int i = 0; i < size\_; i++){

printf(" (%f, %f)", vertices\_[i].X\_, vertices\_[i].Y\_);

}

printf("\n");

}

};

ПРИЛОЖЕНИЕ Б

(Справочное)

Код класса GA\_unit

#include "Point.h"

#include "GA\_gene.h"

#include <cmath>

#include <stdlib.h>

#include <stdio.h>

#include <algorithm>

#include <string>

#include <omp.h>

#pragma once

using namespace std;

class GA\_unit{// особь популяции

private:

string id\_ = "p";

int geneSize\_ = 0;

GA\_gene \* genes = new GA\_gene[0];

GA\_gene \* bufgenes = new GA\_gene[0];

double fieldSize\_=4;

public:

int errors\_=0;

double evaluation\_ = 0;

GA\_unit(){}

GA\_unit(int geneSize, double fieldSize){// инициализация с рандомными генами

geneSize\_ = geneSize;

fieldSize\_ = fieldSize;

delete[] genes;

delete[] bufgenes;

genes = new GA\_gene[geneSize\_];

bufgenes = new GA\_gene[geneSize\_];

for (int i = 0; i < geneSize\_; i++){

genes[i] = GA\_gene(fieldSize\_);

}

//evaluation\_ = evaluateUnit();

evaluation\_ = evaluateDroppedUnit();

}

GA\_unit(double fieldSize){// инициализация без генов

evaluation\_ = 0;

geneSize\_ = 0;

fieldSize\_ = fieldSize;

delete[] genes;

delete[] bufgenes;

bufgenes = new GA\_gene[0];

genes = new GA\_gene[0];

}

~GA\_unit(){

delete[] bufgenes;

delete[] genes;

}

void addGene(GA\_gene newGene){// добавление гена

newGene.randomizePosition(fieldSize\_);

geneSize\_++;

delete[] bufgenes;

bufgenes = new GA\_gene[geneSize\_];

GA\_gene \* bufdelete = genes;

GA\_gene \* buf = new GA\_gene[geneSize\_];

for (int i = 0; i < geneSize\_ - 1; i++){

buf[i] = genes[i];

}

buf[geneSize\_ - 1] = newGene;

genes = buf;

delete[] bufdelete;

evaluation\_ = evaluateUnit();

while (errors\_ > 0) {

genes[geneSize\_ - 1].randomizePosition(fieldSize\_);

evaluation\_ = evaluateUnit();

}

}

void testaddGene(GA\_gene newGene){// добавление гена

geneSize\_++;

delete[] bufgenes;

bufgenes = new GA\_gene[geneSize\_];

GA\_gene \* bufdelete = genes;

GA\_gene \* buf = new GA\_gene[geneSize\_];

for (int i = 0; i < geneSize\_ - 1; i++){

buf[i] = genes[i];

}

buf[geneSize\_ - 1] = newGene;

genes = buf;

delete[] bufdelete;

evaluation\_ = evaluateUnit();

//evaluation\_ = evaluateDroppedUnit();

}

GA\_unit(const GA\_unit &obj)

{

\*this = obj;

}

//перегруженный оператор

GA\_unit& operator=(const GA\_unit& obj){

if (this != &obj){

id\_ = obj.id\_;

geneSize\_ = obj.geneSize\_;

fieldSize\_ = obj.fieldSize\_;

errors\_ = obj.errors\_;

evaluation\_ = obj.evaluation\_;

delete[] genes;

delete[] bufgenes;

GA\_gene \* newGenes = new GA\_gene[obj.geneSize\_];

GA\_gene \* newBufGenes = new GA\_gene[obj.geneSize\_];

//std::copy(obj.genes, obj.genes + obj.geneSize\_, newGenes);

for (int i = 0; i < geneSize\_; i++){

newGenes[i] = obj.genes[i];

newBufGenes[i] = obj.bufgenes[i];

}

genes = newGenes;

bufgenes = newBufGenes;

}

return \*this;

}

// мутировать особь

void MutateUnit(){

int mutateType = rand() % 4;

for (int i = 0; i < geneSize\_; i++){

genes[i].mutate(mutateType);

}

}

// скрестить особь с заменой гена

void KrossingoverByGeneNumber(GA\_unit secondParent, int geneNumber){

GA\_gene tmp = GA\_gene(secondParent.genes[geneNumber]);

genes[geneNumber] = tmp;

//recalcEvaluation();

}

void DroppedKrossingoverByGeneNumber(GA\_unit secondParent, int geneNumber){

GA\_gene tmp = GA\_gene(secondParent.genes[geneNumber]);

genes[geneNumber] = tmp;

recalcDroppedEvaluation();

}

// скрестить особь, ген принимает среднее между двумя генами значение

void KrossingoverByGeneNumberMean(GA\_unit secondParent, int geneNumber){

GA\_gene tmp = GA\_gene(secondParent.genes[geneNumber]);

Point tmpPoint = tmp.getPosition();

Point thisPoint = genes[geneNumber].getPosition();

thisPoint.mean(tmpPoint);

tmp.setPosition(thisPoint);

genes[geneNumber] = tmp;

recalcEvaluation();

}

//проверка на пересечение сторон генов

bool LineIntersection(Point fbegin, Point fend, Point sbegin, Point send){

// обработка пересечения параллельных осям линий

bool firstparallelX = false;

bool firstparallelY = false;

bool secondparallelX = false;

bool secondparallelY = false;

if (abs(fend.Y\_ - fbegin.Y\_) < 0.001) firstparallelX = true;

if (abs(fend.X\_ - fbegin.X\_) < 0.001) firstparallelY = true;

if (abs(send.Y\_ - sbegin.Y\_) < 0.001) secondparallelX = true;

if (abs(send.X\_ - sbegin.X\_) < 0.001) secondparallelY = true;

if (firstparallelX || firstparallelY || secondparallelX || secondparallelY){

if (firstparallelX){

bool yfit = send.Y\_<fend.Y\_&&sbegin.Y\_>fend.Y\_ || send.Y\_ > fend.Y\_&&sbegin.Y\_ < fend.Y\_;

double x = ((sbegin.X\_ - send.X\_)\*fend.Y\_ + (send.X\_\*sbegin.Y\_ - sbegin.X\_\*send.Y\_)) / (sbegin.Y\_ - send.Y\_);

bool xfit = fend.X\_<x&&fbegin.X\_>x || fend.X\_ > x&&fbegin.X\_ < x;

if (yfit && xfit) return true;

}

if (firstparallelY){

bool xfit = send.X\_<fend.X\_&&sbegin.X\_>fend.X\_ || send.X\_ > fend.X\_&&sbegin.X\_ < fend.X\_;

double y = ((sbegin.Y\_ - send.Y\_)\*fend.X\_ + (send.Y\_\*sbegin.X\_ - sbegin.Y\_\*send.X\_)) / (sbegin.X\_ - send.X\_);

bool yfit = fend.Y\_<y&&fbegin.Y\_>y || fend.Y\_ > y&&fbegin.Y\_ < y;

if (yfit && xfit) return true;

}

if (secondparallelX){

bool yfit = fend.Y\_<send.Y\_&&fbegin.Y\_>send.Y\_ || fend.Y\_ > send.Y\_&&fbegin.Y\_ < send.Y\_;

double x = ((fbegin.X\_ - fend.X\_)\*send.Y\_ + (fend.X\_\*fbegin.Y\_ - fbegin.X\_\*fend.Y\_)) / (fbegin.Y\_ - fend.Y\_);

bool xfit = send.X\_<x&&sbegin.X\_>x || send.X\_ > x&&sbegin.X\_ < x;

if (yfit && xfit) return true;

}

if (secondparallelY){

bool xfit = fend.X\_<send.X\_&&fbegin.X\_>send.X\_ || fend.X\_ > send.X\_&&fbegin.X\_ < send.X\_;

double y = ((fbegin.Y\_ - fend.Y\_)\*send.X\_ + (fend.Y\_\*fbegin.X\_ - fbegin.Y\_\*fend.X\_)) / (fbegin.X\_ - fend.X\_);

bool yfit = send.Y\_<y&&sbegin.Y\_>y || send.Y\_ > y&&sbegin.Y\_ < y;

if (yfit && xfit) return true;

}

}

double a1, b1, a2, b2;

bool pox = false;

a1 = (fend.Y\_ - fbegin.Y\_) / (fend.X\_ - fbegin.X\_);

a2 = (send.Y\_ - sbegin.Y\_) / (send.X\_ - sbegin.X\_);

b1 = (fend.X\_ \* fbegin.Y\_ - fbegin.X\_ \* fend.Y\_) / (fend.X\_ - fbegin.X\_);

b2 = (send.X\_ \* sbegin.Y\_ - sbegin.X\_ \* send.Y\_) / (send.X\_ - sbegin.X\_);

double b = b2 - b1;

double a = a1 - a2;

if (abs(a) <0.0001) return false;

else{

double x = b / a;

bool cross1 = fbegin.X\_ - fend.X\_ > 0.001 && x - fbegin.X\_<0.001 && x - fend.X\_>0.001 || fbegin.X\_ - fend.X\_ < 0 && x - fbegin.X\_>0 && x - fend.X\_<0;

bool cross2 = sbegin.X\_ - send.X\_>0.001 && x - sbegin.X\_<0.001 && x - send.X\_>0.001 || sbegin.X\_ - send.X\_ < 0 && x - sbegin.X\_>0 && x - send.X\_<0;

if (cross1&&cross2)

pox = true;

}

a1 = (fend.X\_ - fbegin.X\_) / (fend.Y\_ - fbegin.Y\_);

a2 = (send.X\_ - sbegin.X\_) / (send.Y\_ - sbegin.Y\_);

b1 = (fend.Y\_ \* fbegin.X\_ - fbegin.Y\_ \* fend.X\_) / (fend.Y\_ - fbegin.Y\_);

b2 = (send.Y\_ \* sbegin.X\_ - sbegin.Y\_ \* send.X\_) / (send.Y\_ - sbegin.Y\_);

b = b2 - b1;

a = a1 - a2;

if (abs(a) <0.0001) return false;

else{

double y = b / a;

bool cross1 = fbegin.Y\_ - fend.Y\_ > 0.001 && y - fbegin.Y\_<0.001 && y - fend.Y\_>0.001 || fbegin.Y\_ - fend.Y\_ < 0 && y - fbegin.Y\_>0 && y - fend.Y\_<0;

bool cross2 = sbegin.Y\_ - send.Y\_>0.001 && y - sbegin.Y\_<0.001 && y - send.Y\_>0.001 || sbegin.Y\_ - send.Y\_ < 0 && y - sbegin.Y\_>0 && y - send.Y\_<0;

if (cross1&&cross2&& pox)

return true;

}

return false;

}

//проверка на пересечение генов

bool UnitsIntersection(GA\_gene first, GA\_gene second){

bool ka = false;

for (int i = 0; i < first.size\_ +1; i++){

for (int j = 0; j < second.size\_ +1; j++)

{

//if (i != j){

Point f1, e1, f2, e2;

f1 = Point(first.vertices\_[i%first.size\_].X\_ + first.getPosition().X\_, first.vertices\_[i%first.size\_].Y\_ + first.getPosition().Y\_);

e1 = Point(first.vertices\_[(i + 1) % first.size\_].X\_ + first.getPosition().X\_, first.vertices\_[(i + 1) % first.size\_].Y\_ + first.getPosition().Y\_);

f2 = Point(second.vertices\_[j%second.size\_].X\_ + second.getPosition().X\_, second.vertices\_[j%second.size\_].Y\_ + second.getPosition().Y\_);

e2 = Point(second.vertices\_[(j + 1) % second.size\_].X\_ + second.getPosition().X\_, second.vertices\_[(j + 1) % second.size\_].Y\_ + second.getPosition().Y\_);

float check = (first.getPosition().X\_ - second.getPosition().X\_)\*(first.getPosition().X\_ - second.getPosition().X\_) + (first.getPosition().Y\_ - second.getPosition().Y\_)\*(first.getPosition().Y\_ - second.getPosition().Y\_);

if (LineIntersection(f1, e1, f2, e2))

ka = true;

//}

}

}

return ka;

}

//пересчитать пригодность особи

void recalcEvaluation(){

double tmp = evaluation\_;

evaluation\_ = evaluateUnit();

//if (abs(tmp - evaluation\_) > 0.0001)

//printf("===================evaluation new: %f - %f = %f\n", tmp, evaluation\_, tmp - evaluation\_);

}

void recalcDroppedEvaluation(){

double tmp = evaluation\_;

evaluation\_ = evaluateDroppedUnit();

//if (abs(tmp - evaluation\_) > 0.0001)

//printf("===================evaluation new: %f - %f = %f\n", tmp, evaluation\_, tmp - evaluation\_);

}

float distance(GA\_unit second){

double dist = 0;

for (int i = 0; i < geneSize\_; i++){

dist += abs(second.genes[i].getPosition().X\_ - genes[i].getPosition().X\_) + abs(second.genes[i].getPosition().Y\_ - genes[i].getPosition().Y\_);

}

return dist;

}

//поставить всё в ноль

void setToZero(){

for (int i = 0; i < geneSize\_; i++){

genes[i].setPositionZero();

}

}

// измерить приспособленность особи

float evaluateUnit(){

double result = 0;

for (int i = 0; i < geneSize\_; i++){

GA\_gene rotatedGene = genes[i];

rotatedGene.updateGene();

result += rotatedGene.maxXY\_.Y\_;

}

int errors = 0;

for (int i = 0; i < geneSize\_; i++){

GA\_gene rotatedGene = genes[i];

rotatedGene.updateGene();

if (rotatedGene.maxXY\_.Y\_ < 0) {

errors++;

result -= rotatedGene.maxXY\_.Y\_ \* 10;

}

if (rotatedGene.maxXY\_.X\_ > fieldSize\_) {

errors++;

result += (rotatedGene.maxXY\_.X\_ - fieldSize\_)\*10;

}

if (rotatedGene.minXY\_.X\_ < 0) {

errors++;

result -= rotatedGene.minXY\_.X\_ \* 10;

}

if (rotatedGene.minXY\_.Y\_ < 0) {

errors++;

result -= rotatedGene.minXY\_.Y\_ \* 10;

}//check units out of range

}

for (int i = 0; i < geneSize\_ - 1; i++){

for (int j = i + 1; j < geneSize\_; j++){

GA\_gene rotatedGene1 = genes[i];

GA\_gene rotatedGene2 = genes[j];

rotatedGene1.updateGene();

rotatedGene2.updateGene();

if (UnitsIntersection(rotatedGene1, rotatedGene2)){

errors++;// check intersection

}

}

}

errors\_ = errors;

return result \* (errors+1);// pick multiplication number for errors

}

float evaluateDroppedUnit(){

double result = 0;

double delta = 0;

GA\_gene \* bufferGenes = new GA\_gene[geneSize\_];

for (int i = 0; i < geneSize\_; i++){

//genes[i].updateGene();

GA\_gene rotatedGene = genes[i];

rotatedGene.updateGene();

if (rotatedGene.maxXY\_.Y\_ > delta) delta = rotatedGene.maxXY\_.Y\_;// find maximum Y

bufferGenes[i] = rotatedGene;

result += rotatedGene.maxXY\_.Y\_;

}

delta = delta / 4;

int errors = 0;

for (int i = 0; i < geneSize\_; i++){

if (bufferGenes[i].maxXY\_.Y\_ < 0) {

errors++;

}

if (bufferGenes[i].maxXY\_.X\_ > fieldSize\_) {

errors++;

}

if (bufferGenes[i].minXY\_.X\_ < 0) {

errors++;

}

if (bufferGenes[i].minXY\_.Y\_ < 0) {

errors++;

}//check units out of range

}

for (int i = 0; i < geneSize\_ - 1; i++){

for (int j = i + 1; j < geneSize\_; j++){

GA\_gene rotatedGene1 = bufferGenes[i];

GA\_gene rotatedGene2 = bufferGenes[j];

if (UnitsIntersection(rotatedGene1, rotatedGene2)){

errors++;// check intersection

}

}

}

if (errors == 0){

for (int i = 0; i < 100; i++)

{

for (int j = 0; j < geneSize\_; j++)

{

double dropH = delta;

for (int k = 0; k < 10; k++)

{

int buferrors = 0;

bufferGenes[j].drop(dropH);

bufferGenes[j].updateGene();

if (bufferGenes[j].minXY\_.Y\_ < 0) buferrors++;

for (int t = 0; t < geneSize\_; t++){

if (t != j){

if (UnitsIntersection(bufferGenes[j], bufferGenes[t])) buferrors++;

}

}

if (buferrors>0) bufferGenes[j].rise(dropH);

dropH = dropH / 2;

}

}

}

errors = 0;

result = 0;

for (int i = 0; i < geneSize\_; i++){

result += bufferGenes[i].maxXY\_.Y\_;// find maximum Y

//result += rotatedGene.maxXY\_.Y\_;

}

for (int i = 0; i < geneSize\_; i++){

if (bufferGenes[i].maxXY\_.Y\_ < 0) {

errors ++;

result -= bufferGenes[i].maxXY\_.Y\_ \* 10;

}

if (bufferGenes[i].maxXY\_.X\_ > fieldSize\_) {

errors ++;

result += (bufferGenes[i].maxXY\_.X\_ - fieldSize\_) \* 10;

}

if (bufferGenes[i].minXY\_.X\_ < 0) {

errors ++;

result -= bufferGenes[i].minXY\_.X\_ \* 10;

}

if (bufferGenes[i].minXY\_.Y\_ < 0) {

errors ++;

result -= bufferGenes[i].minXY\_.Y\_ \* 10;

}//check units out of range

}

for (int i = 0; i < geneSize\_ - 1; i++){

for (int j = i + 1; j < geneSize\_; j++){

GA\_gene rotatedGene1 = bufferGenes[i];

GA\_gene rotatedGene2 = bufferGenes[j];

if (UnitsIntersection(rotatedGene1, rotatedGene2)){

errors ++;// check intersection

}

}

}

}

else result \*= 10;

errors\_ = errors;

delete [] bufgenes;

bufgenes = bufferGenes;

return result \* (errors + 1);// pick multiplication number for errors

}

// напечатать только значение

void printUnit(){

printf("Unit evaluation: %.2f, errors: %d\n", evaluation\_, errors\_);

for (int i = 0; i < geneSize\_; i++){

genes[i].printGene();

}

printf("\n");

}

// напечатать особь полностью

void printFullUnit(){

double result = 0;

for (int i = 0; i < geneSize\_; i++){

if (genes[i].maxXY\_.Y\_ > result) result = genes[i].maxXY\_.Y\_;// find maximum Y

}

printf("Unit evaluation: %.2f, Max Y: %.2f\n", evaluation\_,result);

for (int i = 0; i < geneSize\_; i++){

genes[i].printFullGene();

}

printf("\n");

}

void printFullDroppedUnit(){

double result = 0;

for (int i = 0; i < geneSize\_; i++){

if (bufgenes[i].maxXY\_.Y\_ > result) result = genes[i].maxXY\_.Y\_;// find maximum Y

}

printf("Unit evaluation: %.2f, Max Y: %.2f\n", evaluation\_, result);

for (int i = 0; i < geneSize\_; i++){

bufgenes[i].printFullGene();

}

printf("\n");

}

void printUnitToFile(string &result){

result += to\_string(evaluation\_);

result += "\n";

for (int i = 0; i < geneSize\_; i++){

genes[i].printGeneToFileUnit(result);

}

}

void printDroppedUnitToFile(string &result){

result += to\_string(evaluation\_);

result += "\n";

for (int i = 0; i < geneSize\_; i++){

bufgenes[i].printGeneToFileUnit(result);

}

}

};

ПРИЛОЖЕНИЕ В

(Справочное)

Код класса GA\_population

#pragma once

#include "Point.h"

#include "GA\_gene.h"

#include "GA\_unit.h"

#include <cmath>

#include <stdlib.h>

#include <stdio.h>

#include <string>

#include <iostream>

#include <fstream>

#include <omp.h>

using namespace std;

class GA\_population {// = популяция

private:

double fieldSize\_;

int steps\_;

int size\_ = 0;// размер популяции

GA\_unit \* units\_;

GA\_gene \* genesContainer\_;// гены популяции

int genesContainerSize\_;

int newPopulations\_ = 0;

int elites\_ = 0;

bool square\_ = false;

int childrenbetter\_ = 0;

public:

double mutateFrequency\_ = 0;

double best;

double previuos\_best;

GA\_population(){

}

GA\_population(int populationSize, int elites, double mutateFrequency, bool square)

{

square\_ = square;

size\_ = populationSize;

elites\_ = elites;

mutateFrequency\_ = mutateFrequency;

if (square\_)

units\_ = new GA\_unit[size\_\*size\_];

else

units\_ = new GA\_unit[size\_];

}

~GA\_population(){

delete[] units\_;

delete[] genesContainer\_;

}

void sortPopulations(){

int tmp;

//qsort(units\_, size\_, sizeof(GA\_unit), ComparePopulations);

for (int i = 0; i < size\_-1; i++){

tmp = i;

for (int j = i+1; j < size\_; j++){

if (units\_[j].evaluation\_ < units\_[tmp].evaluation\_) tmp = j;

}

if (tmp != i){

GA\_unit buf = units\_[i];

units\_[i] = units\_[tmp];

units\_[tmp] = buf;

}

}

previuos\_best = best;

best = units\_[0].evaluation\_;

}

void sortPopulations(GA\_unit \* unitsArray, int size){

int tmp;

for (int i = 0; i < size\_ - 1; i++){

tmp = i;

for (int j = i + 1; j < size\_; j++){

if (unitsArray[j].evaluation\_ < unitsArray[tmp].evaluation\_) tmp = j;

}

if (tmp != i){

GA\_unit buf = unitsArray[i];

unitsArray[i] = unitsArray[tmp];

unitsArray[tmp] = buf;

}

}

}

void sortUnits(GA\_unit \* units, int size){

int tmp;

for (int i = 0; i < size - 1; i++){

tmp = i;

for (int j = i + 1; j < size; j++){

if (units[j].evaluation\_ < units[tmp].evaluation\_) tmp = j;

}

if (tmp != i){

GA\_unit buf = units[i];

units[i] = units[tmp];

units[tmp] = buf;

}

}

}

void addGenesToContainer(GA\_gene newGene){

genesContainerSize\_++;

//GA\_gene \* bufdelete = genesContainer\_;

GA\_gene \* buf = new GA\_gene[genesContainerSize\_];

//genes[populationSize\_]

for (int i = 0; i < genesContainerSize\_ - 1; i++){

buf[i] = genesContainer\_[i];

}

buf[genesContainerSize\_ - 1] = newGene;

delete[] genesContainer\_;

genesContainer\_ = buf;

}

int CompareUnits(const void \* vfirst, const void \* vsecond)

{

const GA\_unit \*first = reinterpret\_cast<const GA\_unit\*>(vfirst);

const GA\_unit \*second = reinterpret\_cast<const GA\_unit\*>(vsecond);

return (int)(first->evaluation\_ - second->evaluation\_);

}

void receiveUnit(GA\_unit newUnit){

units\_[size\_ - 1] = newUnit;

}

void GenerateNewPopulations(GA\_unit \* unitsArray, int size, int elites){

randomGeneKrossingover(unitsArray, size, elites, mutateFrequency\_);

//twoPointKrossingover(unitsArray, size, elites, mutateFrequency\_);

}

void GenerateNewPopulationsMigration(GA\_unit \* unitsArray, GA\_unit \* children, int element, int size, int elites){

migrationRandomGeneKrossingover(unitsArray, children,element, size, elites, mutateFrequency\_);

//twoPointKrossingover(unitsArray, size, elites, mutateFrequency\_);

}

void GenerateNewPopulationsSingleChild(GA\_unit \* unitsArray, GA\_unit \* children, int element, int size, int elites){

parallelRandomGeneKrossingover(unitsArray, children, element, size, elites, mutateFrequency\_);

//twoPointKrossingover(unitsArray, size, elites, mutateFrequency\_);

}

void GenerateNewDroppedPopulations(GA\_unit \* unitsArray, int size, int elites){

droppedRandomGeneKrossingover(unitsArray, size, elites, mutateFrequency\_);

//twoPointKrossingover(unitsArray, size, elites, mutateFrequency\_);

}

// берёт среднее значение случайных генов

void randomMeanKrossingover(){

int first = 0, second = 0;

while (first == second){

first = rand() % size\_;

second = rand() % size\_;

}

int allPossibleCombinations = (genesContainerSize\_\*genesContainerSize\_ - 3);

GA\_unit best(fieldSize\_);

GA\_unit candidate(fieldSize\_);

for (int i = 0; i < newPopulations\_; i++){

best = units\_[first];

for (int j = 0; j < allPossibleCombinations; j++)

{

if (j < allPossibleCombinations / 2) candidate = GA\_unit(units\_[first]);

else candidate = GA\_unit(units\_[second]);

if (j < allPossibleCombinations / 2){

candidate.KrossingoverByGeneNumberMean(units\_[second], j % 2);

if (j >= 2) candidate.KrossingoverByGeneNumberMean(units\_[second], 2 + j % 2);

}

else{

candidate.KrossingoverByGeneNumberMean(units\_[first], (j - 1) % 2);

if (j - allPossibleCombinations / 2 >= 2) candidate.KrossingoverByGeneNumberMean(units\_[first], 2 + (j - 1) % 2);

}

if (candidate.evaluation\_ > best.evaluation\_){

best = candidate;

}

}

if (fabs(best.evaluation\_ - units\_[size\_ - 1].evaluation\_) > 1e-6){

units\_[size\_ - 1 -i] = best;

}

}

}

// выбирает случайную особь, ищет самого отличного к ней и скрещивает

// опыты показали, что популяция расходиться при таком скрещивании

void selectDistinctKrossingover(){

int first = 0, second = 0;

first = rand() % size\_;

double distance = 0;

for (int i = 0; i < size\_; i++){

if (distance < units\_[first].distance(units\_[i])) {

second = i;

distance = units\_[first].distance(units\_[i]);

}

}

int allPossibleCombinations = (genesContainerSize\_\*genesContainerSize\_ - 3);

GA\_unit best(fieldSize\_);

GA\_unit candidate(fieldSize\_);

for (int i = 0; i < newPopulations\_; i++){

best = units\_[first];

for (int j = 0; j < allPossibleCombinations; j++)

{

if (j < allPossibleCombinations / 2) candidate = GA\_unit(units\_[first]);

else candidate = GA\_unit(units\_[second]);

if (j < allPossibleCombinations / 2){

candidate.KrossingoverByGeneNumberMean(units\_[second], j % 2);

if (j >= 2) candidate.KrossingoverByGeneNumberMean(units\_[second], 2 + j % 2);

}

else{

candidate.KrossingoverByGeneNumberMean(units\_[first], (j - 1) % 2);

if (j - allPossibleCombinations / 2 >= 2) candidate.KrossingoverByGeneNumberMean(units\_[first], 2 + (j - 1) % 2);

}

if (candidate.evaluation\_ > best.evaluation\_){

best = candidate;

}

}

if (fabs(best.evaluation\_ - units\_[size\_ - 1].evaluation\_) > 1e-6){

units\_[size\_ - 1 - i] = best;

}

}

}

// случайный кроссинговер случайного гена элитной особи и неэлитной

void randomGeneKrossingover(){

GA\_unit \* children = new GA\_unit[size\_ - elites\_];

for (int i = 0; i < size\_-elites\_; i++){

int first = 0, second = 0;

while (first == second){

first = rand() % elites\_;

second = elites\_ + rand() % (size\_-elites\_);

}

GA\_unit candidate = GA\_unit(units\_[first]);

int geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(units\_[second], geneExchangeNumber);

children[i] = candidate;

}

mutateUnitInArray(children, size\_-elites\_,mutateFrequency\_);

sortUnits(children,size\_-elites\_);

for (int i = elites\_; i < size\_; i++){

units\_[i] = children[i-elites\_];

}

delete[] children;

}

void randomGeneKrossingover(GA\_unit \* unitsArray, int size, int elites, double mutate){

GA\_unit \* children = new GA\_unit[size - elites];

for (int i = 0; i < size - elites; i++){

int first = 0, second = 0;

while (first == second){

first = rand() % elites;

second = elites + rand() % (size - elites);

}

GA\_unit candidate = GA\_unit(unitsArray[first]);

int geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[second], geneExchangeNumber);

children[i] = candidate;

}

mutateUnitInArray(children, size - elites, mutate);

for (int i = 0; i < size - elites; i++)

children[i].recalcEvaluation();

sortUnits(children, size - elites);

childrenbetter\_ = 0;

for (int i = 0; i < size - elites; i++){

if (children[i].evaluation\_ < unitsArray[elites - 1].evaluation\_) childrenbetter\_++;

}

for (int i = elites; i < size; i++){

unitsArray[i] = children[i - elites];

}

delete[] children;

}

void migrationRandomGeneKrossingover(GA\_unit \* unitsArray, GA\_unit \* children, int element, int size, int elites, double mutate){

int first = element;

GA\_unit candidate = GA\_unit(unitsArray[first]);

int geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[(element+size)%(size\*size)], geneExchangeNumber);

children[(element + 1) \* 8 - 8] = candidate;

geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[(element - size + size\*size) % (size\*size)], geneExchangeNumber);

children[(element + 1) \* 8 - 7] = candidate;

geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[(element +1) % (size\*size)], geneExchangeNumber);

children[(element + 1) \* 8 - 6] = candidate;

geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[(element - 1 + size\*size) % (size\*size)], geneExchangeNumber);

children[(element + 1) \* 8 - 5] = candidate;

geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[(element + size + 1 + size\*size) % (size\*size)], geneExchangeNumber);

children[(element + 1) \* 8 - 4] = candidate;

geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[(element - size+1 + size\*size) % (size\*size)], geneExchangeNumber);

children[(element + 1) \* 8 - 3] = candidate;

geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[(element + size - 1 + size\*size) % (size\*size)], geneExchangeNumber);

children[(element + 1) \* 8 - 2] = candidate;

geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[(element - size-1 + size\*size) % (size\*size)], geneExchangeNumber);

children[(element + 1) \* 8 - 1] = candidate;

}

void parallelRandomGeneKrossingover(GA\_unit \* unitsArray, GA\_unit \* children, int element, int size, int elites, double mutate){

int first = 0, second = 0;

while (first == second){

first = rand() % elites;

second = elites + rand() % (size - elites);

}

GA\_unit candidate = GA\_unit(unitsArray[first]);

int geneExchangeNumber = rand() % (genesContainerSize\_);

candidate.KrossingoverByGeneNumber(unitsArray[second], geneExchangeNumber);

children[element] = candidate;

}

// двухточечный элитно-неэлитный кроссинговер

void twoPointKrossingover(GA\_unit \* unitsArray, int size, int elites, double mutate){

GA\_unit \* children = new GA\_unit[size - elites];

for (int i = 0; i < size - elites; i++){

int first = 0, second = 0;

while (first == second){

first = rand() % elites;

second = elites + rand() % (size - elites);

}

int startGene = rand() % genesContainerSize\_;

int endGene = rand() % genesContainerSize\_;

int genesChange = 0;

if (endGene > startGene){

genesChange = endGene - startGene+1;

}

else{

genesChange = genesContainerSize\_ - startGene + endGene + 1;

}

GA\_unit candidate = GA\_unit(unitsArray[first]);

for (int j = 0; j < genesChange; j++){

candidate.KrossingoverByGeneNumber(unitsArray[second], (startGene+j)%genesContainerSize\_);

}

children[i] = candidate;

}

mutateUnitInArray(children, size - elites, mutate);

sortUnits(children, size - elites);

for (int i = elites; i < size; i++){

unitsArray[i] = children[i - elites];

}

delete[] children;

}

void dummyInit(double fieldSize, int geneCount){

fieldSize\_ = fieldSize;

genesContainer\_ = new GA\_gene[geneCount];

genesContainerSize\_ = geneCount;

for (int i = 0; i < genesContainerSize\_;i++)

genesContainer\_[i] = GA\_gene();

if (square\_){

for (int i = 0; i < size\_\*size\_; i++){

units\_[i] = GA\_unit(fieldSize);

for (int j = 0; j < geneCount; j++){

units\_[i].addGene(genesContainer\_[j]);

}

}

sortPopulations(units\_, size\_\*size\_);

}

else

{

for (int i = 0; i < size\_; i++){

units\_[i] = GA\_unit(fieldSize);

for (int j = 0; j < geneCount; j++){

units\_[i].addGene(genesContainer\_[j]);

}

}

sortPopulations();

}

best = units\_[0].evaluation\_;

}

void Init(double fieldSize, int populationSize)

{

for (int i = 0; i < size\_; i++){

char \* buf = "";

units\_[i] = GA\_unit(populationSize, fieldSize);

}

}

void mutateUnitInArray(GA\_unit \* array, int size, double frequency){

for (int i = 0; i < size \* frequency; i++){

int k = rand() % size;

array[k].MutateUnit();

}

}

void Step(){

GenerateNewPopulations(units\_, size\_, elites\_);

sortPopulations();

}

// использовать с квадратными популяциями!

void StepMasterSlave(){

GA\_unit \* children = new GA\_unit[size\_\*size\_ \* 8];

#pragma omp parallel for

for (int i = 0; i < size\_\*size\_;i++)

{

GenerateNewPopulationsMigration(units\_, children, i, size\_, elites\_);

}

sortUnits(children, size\_\*size\_ \* 8);

for (int i = 0; i < size\_\*size\_ - elites\_; i++)

units\_[i + elites\_] = children[i];

sortPopulations();

}

void StepMasterSlaveEvaluation(){

GA\_unit \* children = new GA\_unit[size\_];

for (int i = 0; i < size\_; i++){

GenerateNewPopulationsSingleChild(units\_, children, i, size\_, elites\_);

}

mutateUnitInArray(children, size\_, mutateFrequency\_);

int p;

#pragma omp parallel private (p)

{

int thrds = omp\_get\_num\_threads();

int thrd = omp\_get\_thread\_num();

for (p = thrd; p < size\_; p+=thrds)

{

children[p].recalcEvaluation();

}

}

sortUnits(children, size\_);

for (int i = 0; i < size\_ - elites\_; i++)

units\_[i + elites\_] = children[i];

delete[] children;

sortPopulations();

}

void Stepp(){

int thrd;

GA\_unit \* children = new GA\_unit[size\_];

#pragma omp parallel private(thrd)

{

int thrds = omp\_get\_num\_threads();

int \* sizes = new int[thrds + 1];

int lastsize = size\_;

GA\_unit \*start;

sizes[0] = 0;

for (int i = 1; i < thrds; i++){

sizes[i] = sizes[i - 1] + (int)(size\_ / (double)thrds);

}

sizes[thrds] = lastsize;

thrd = omp\_get\_thread\_num();

for (int i = sizes[thrd]; i < sizes[thrd + 1]; i++){

GenerateNewPopulationsSingleChild(units\_, children, i, size\_, elites\_);

}

mutateUnitInArray(&children[sizes[thrd]], sizes[thrd + 1] - sizes[thrd], mutateFrequency\_);

for (int i = sizes[thrd]; i < sizes[thrd + 1]; i++){

children[i].recalcEvaluation();

}

}

sortUnits(children, size\_);

for (int i = 0; i < size\_ - elites\_; i++){

units\_[i + elites\_] = children[i];

}

sortPopulations();

delete[] children;

}

void Step\_threads(){

int thrd;

#pragma omp parallel private(thrd)

{

int thrds = omp\_get\_num\_threads();

int \* sizes = new int[thrds+1];

int lastsize = size\_;

GA\_unit \*start;

sizes[0] = 0;

for (int i = 1; i < thrds; i++){

sizes[i] = sizes[i-1]+ (int)(size\_ / (double)thrds);

//lastsize -= sizes[i];

}

sizes[thrds] = lastsize;

thrd = omp\_get\_thread\_num();

//printf("thrd %d from %d start", thrd, thrds);

GenerateNewPopulations(&units\_[sizes[thrd]], sizes[thrd+1]-sizes[thrd], (int) elites\_/thrds);// вдруг элитов будет ноль

}

sortPopulations();

}

string getSettings(){

string settings;

char buf[80];

settings+=itoa(size\_,buf,10);

settings += " ";

settings += itoa(elites\_, buf, 10);

settings += " ";

return settings;

}

void printPopulation(){

printf("Population print begin:");

printf("evaluation: %.2f\n", units\_[0].evaluation\_);

units\_[0].printUnit();

}

void printPopulation(char \* fileName, string populationName){

ofstream file(fileName,ofstream::app);

if (file.is\_open())

{

file << populationName+"\n";

file << fieldSize\_ << endl;

file << genesContainerSize\_ << endl;

for (int i = 0; i < genesContainerSize\_; i++){

file << genesContainer\_[i].size\_<<endl;

for (int j = 0; j < genesContainer\_[i].size\_; j++){

file << genesContainer\_[i].vertices\_[j].X\_ <<" "<< genesContainer\_[i].vertices\_[j].Y\_<<endl;

}

}

file<<size\_<<"\n";

for (int i = 0; i < size\_; i++){

string unitString;

units\_[i].printUnitToFile(unitString);

file << unitString;

}

file.close();

}

else printf("Unable to open file\n");

}

void printDroppedPopulation(char \* fileName, string populationName){

//strcat(fileName, ".gap");

ofstream file(fileName, ofstream::app);

if (file.is\_open())

{

file << populationName + "\n";

file << fieldSize\_ << endl;

file << genesContainerSize\_ << endl;

for (int i = 0; i < genesContainerSize\_; i++){

file << genesContainer\_[i].size\_ << endl;

for (int j = 0; j < genesContainer\_[i].size\_; j++){

file << genesContainer\_[i].vertices\_[j].X\_ << " " << genesContainer\_[i].vertices\_[j].Y\_ << endl;

}

}

file << size\_ << "\n";

for (int i = 0; i < size\_; i++){

string unitString;

units\_[i].printDroppedUnitToFile(unitString);

file << unitString;

}

file.close();

}

else printf("Unable to open file\n");

}

};

ПРИЛОЖЕНИЕ Г

Диск с программой