

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**(ДГТУ)**

Факультет «Информатика и вычислительная техника»

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

**Лабораторная работа №6 по дисциплине:**

**«Эвристические методы»**

Выполнил студент группы ВПР32:

Бекеров Максим Валерьевич

Проверил:

профессор Кобак Валерий Григорьевич

г. Ростов-на-Дону

2025г

1. **Общие сведения**

Предметом области исследования расписаний является круг задач проектирования и организационного управления в различных системах, в которых требуется найти наилучшее (оптимальное) значение выбранных критериев их функционирования с учетом имеющихся ограничений.

Программирование для многопроцессорных машинных систем связано с распараллеливанием и синхронизацией вычислений и организацией выполнения параллельных вычислительных процессов. Это выдвигает целый ряд сложных задач, среди которых весьма важными являются расчет характеристик времени и количества операций, требующихся для выполнения параллельных программ и построения расписаний (планов), выполнения параллельных программ на многопроцессорных и многомашинных вычислительных системах.

Модели параллельных программ и операционные характеристики процессов их выполнения служат основой для планирования параллельных вычислительных процессов, т.е. для построения расписаний указанных процессов. Расписания параллельных вычислительных процессов определяют порядок выполнения программы на вычислительной системе, включая распределение частей программы по процессам. С увеличением числа распределяемых частей программ и количества используемых процессоров сложность построения оптимальных расписаний обычно резко возрастает. Поэтому важное значение имеют простые в построении и удобные в реализации приближенные расписания параллельных вычислительных процессов, близкие к оптимальным расписаниям с точки зрения времени выполнения параллельных программ.

1. **Постановка задачи**

Имеется вычислительная система (ВС), состоящая из  несвязанных идентичных устройств (приборов, процессоров и т.п.) .

На обслуживание в ВС поступает набор из  независимых параллельных заданий (работ) , известно время решения  задания  на любом из устройств. При этом каждое задание может выполняться на любом из устройств (процессоре), в каждый момент времени отдельный процессор обслуживает не более одного задания и выполнение задания не прерывается для передачи на другой процессор. Требуется найти такое распределение заданий по процессорам, при котором суммарное время выполнения заданий на каждом из процессоров было бы минимальным. Под расписанием следует понимать отображение , такое, что если , то говорят, что задание  в расписании , назначенного на процессор . При сделанных выше допущениях расписание можно представить разбиением множества заданий на  непересекающихся подмножеств , где .

Критерий, используемый для минимизации времени завершения обслуживания заданий, является минимальным критерием и определяется в следующем виде: , где  - время завершения работы процессора .

1. **Генетический алгоритм**

Генетические алгоритмы (ГА) являются одной из парадигм эволюционных вычислений, представляют собой алгоритмы поиска лучшего, а не оптимального решения задачи, построены на принципах, сходных с принципами естественного отбора и генетики.

ГА имеет вероятностную природу и в связи с этим результаты, получаемые с помощью него отличаются в каждом запуске и определяются случайной последовательностью, переданной в схему алгоритма. Точность алгоритма зависит не только от входной последовательности случайных чисел, но и от условий задачи, таких как размерность задачи и конкретное распределение весов.

Рассмотрим общую схему работы генетического алгоритма:

Ш.1 Формируется начальное поколение, состоящее из заданного числа особей.

Ш.2 Отбор особей и применение ГА операторов кроссовера и мутации с заданной вероятностью для создания нового поколения.

Ш.3 Проверка условия останова, которая обычно заключается в неизменности лучшего решения в течение заданного числа поколений. Если проверка прошла не успешно, то переход на Ш.2.

Ш.4 Лучшая особь выбирается как найденное решение.

ГА является общим алгоритмом для решения любой задачи, и при его применении к конкретной необходимо выбрать механизм кодирования параметров задачи (фенотипа) в гены особи (генотипа), определить оптимизационную функцию  (fitness function) и выбрать условия останова.

В данном случае решения задачи теории расписания минимаксный критерий будет являться оптимизационной функцией, а условием останова - неизменность лучшего решения в течение заданного числа поколений.

Введем следующие обозначения:

 - число особей (хромосом);

 - условие останова по количеству одинаковых поколений;

 - вектор особей (хромосом);

 - вектор приспособленности;

 - вероятность оператора кроссовера;

 - оператор кроссовера;

 - вероятность оператора мутации;

 - оператор мутации.

Рассмотрим подробнее Ш.2 алгоритма:

Ш.2.1 На вход шага поступает исходный вектор особей (хромосом) , каждая из особей представляет какое-то расписание с соответствующим ему  и соответственно приспособленностью особи.

Ш.2.2 , где  - порядковый номер особи в векторе .

Ш.2.3 Выполняется оператор кроссовера с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1]. Если , то , где - случайное число в интервале [1..], исключая совпадение с числом , иначе особь переходит в новое поколение без изменений, т.е. .

Ш.2.4 Выполняется оператор мутации с заданной вероятностью . Для этого генерируется случайное число  в интервале [0..1]. Если , то .

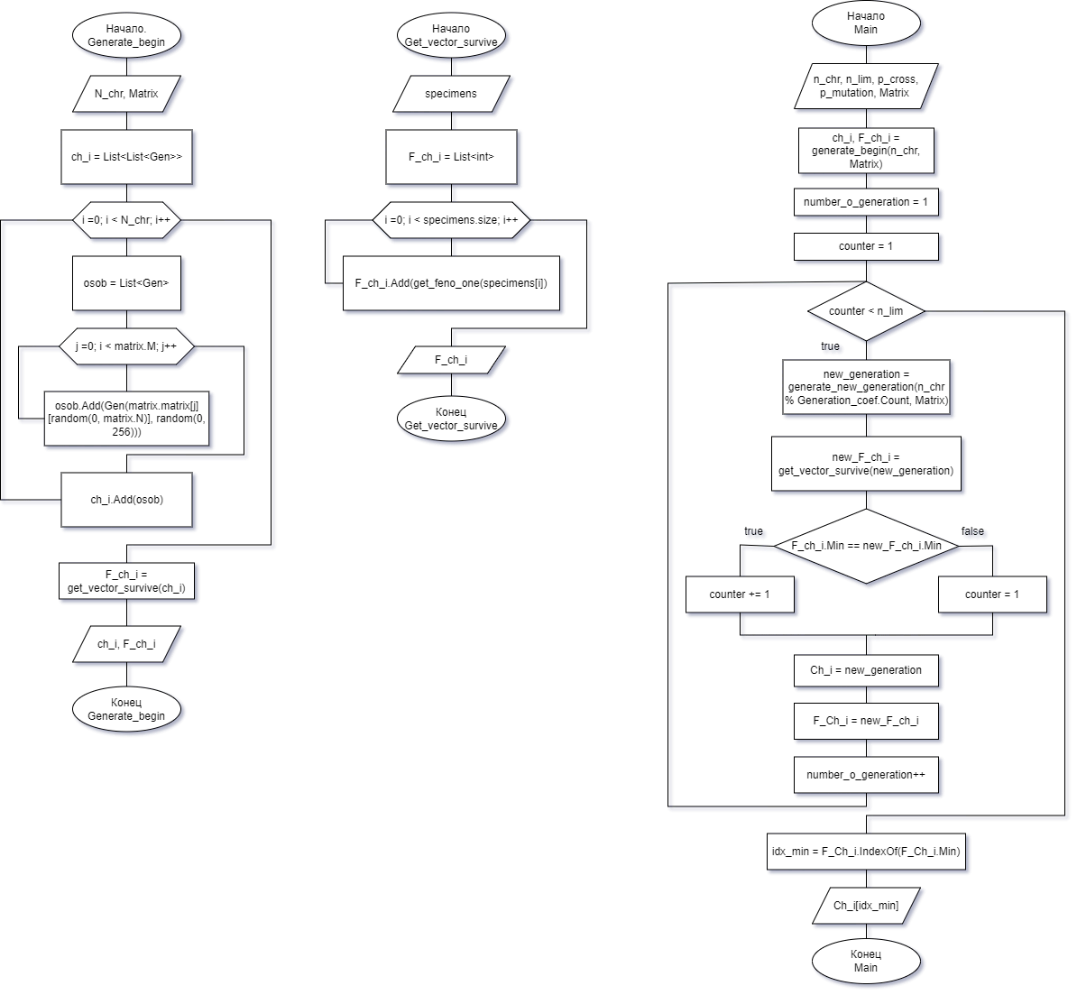
Ш.2.5 . Если , то перейти к Ш.2.3.

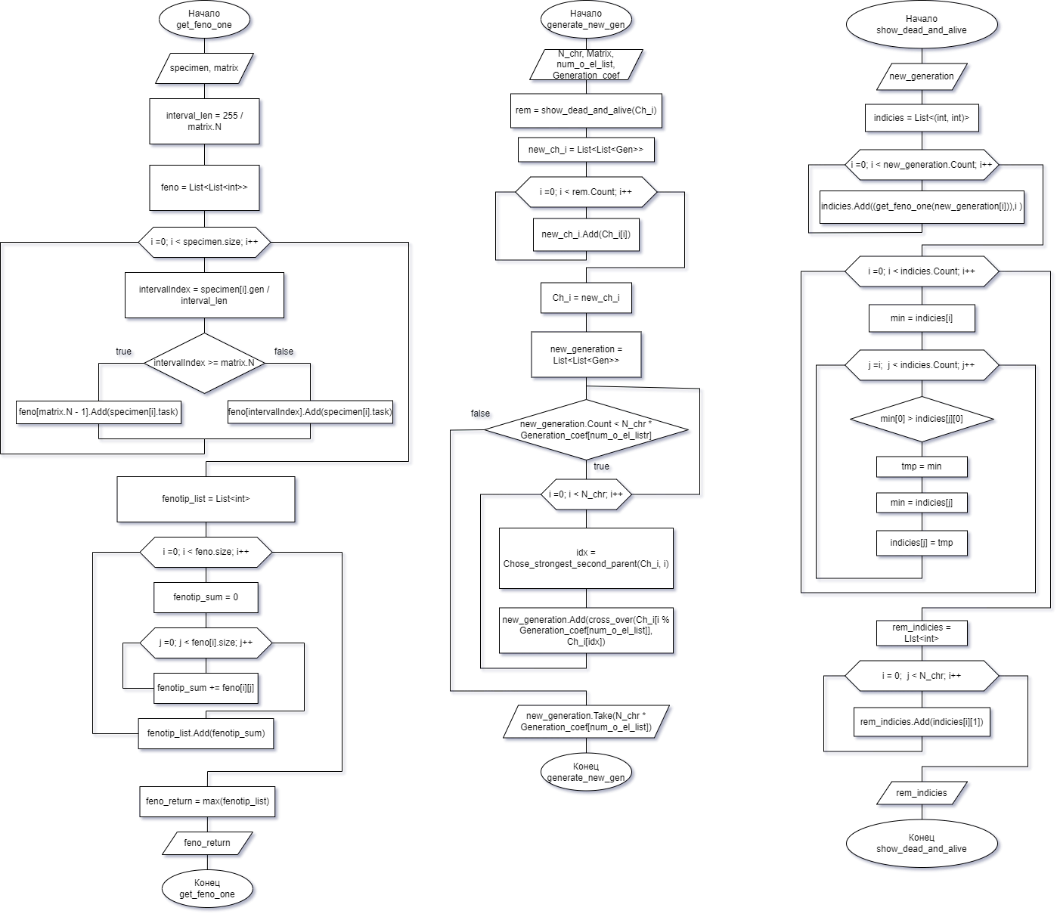
Ш.2.6 Сформировано новое поколение .

1. **Схема алгоритма**

Для написания программы использовался ООП подход, поэтому для наглядности на рисунке 1, представлена UML модель классов, чтобы закрепить методы и поля.

Диаграмма 1 – UML-диаграмма классов

Далее представлены блок-схемы методов, которые не связаны с представлением пользователю. Диаграмма главного метода, метода generate\_begin и метода get\_vector\_survive представлены на рисунке 2.

Рисунок 2 – Блок-схемы методов Main, generate\_begin, get\_vector\_survive

Методы get\_feno\_one, generate\_new\_generation, show\_dead\_and\_alive представлены на рисунке 3.

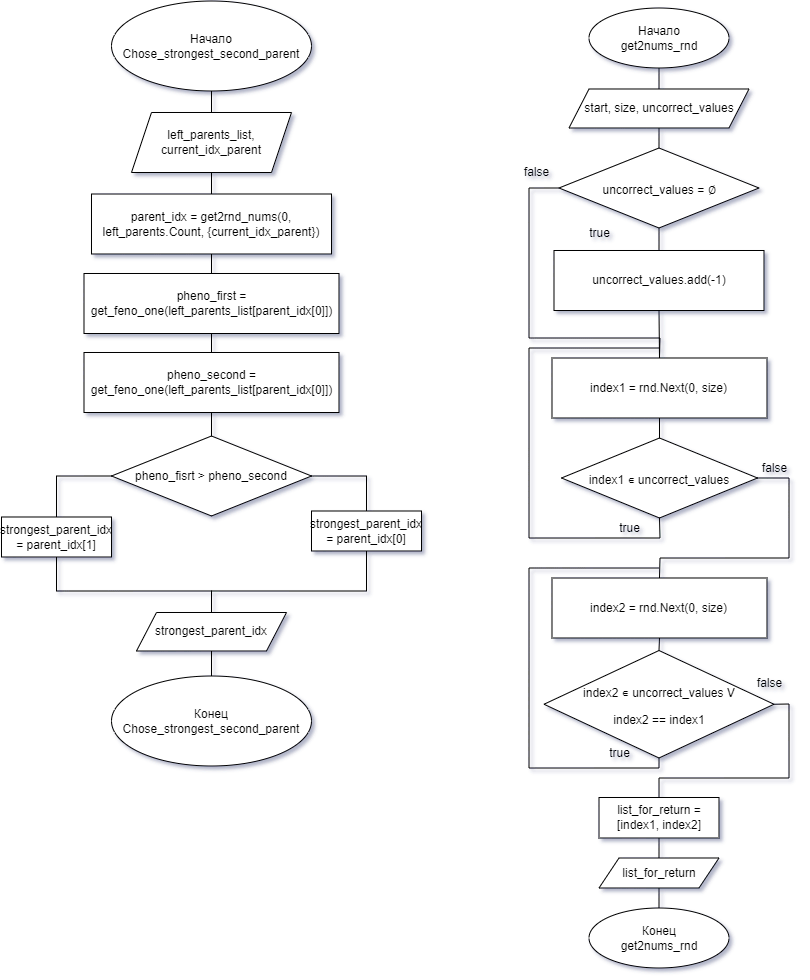
Рисунок 3 – Блок-схемы методов get\_feno\_one, generate\_new\_generation и show\_dead\_and\_aliveТакже в программе используются еще второстепенные методы Chose\_strongest\_second\_parent и get2nums\_rnd, блок схемы которых представлены на рисунке 4.

Рисунок 4 – Блоксхемы методов Chose\_strongest\_second\_parent и get2nums\_rnd.

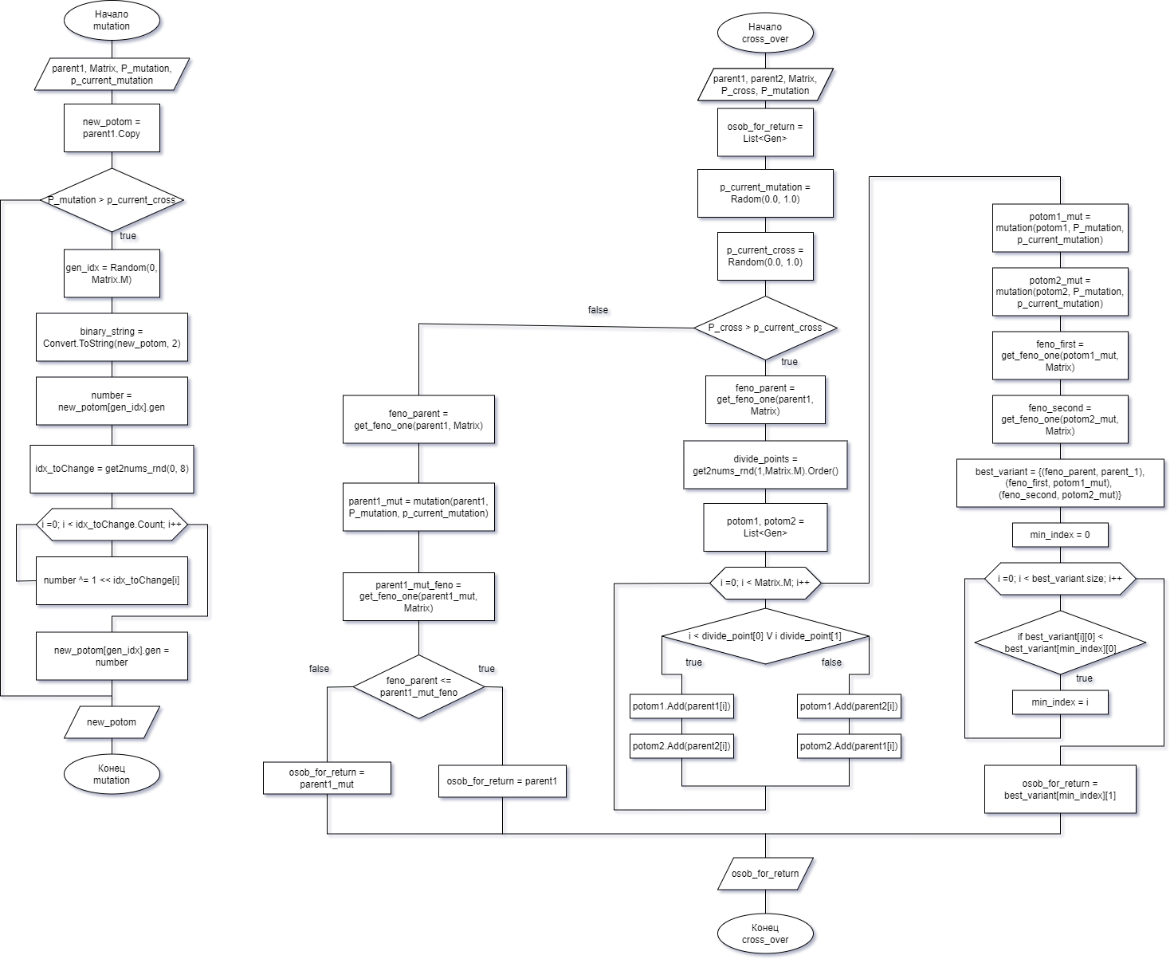
Главные действующие методы cross\_over и mutation представлены на рисунке 5.

Рисунок 5 – Блок-схема методов cross\_over и mutation

1. **Список литературы**
2. Нейронные сети, генетические алгоритмы и нечеткие системы – Рутковская Д., Пилиньский М., Рутковский Л
3. Генетические алгоритмы – Т.В. Панченко
4. Применение генетических алгоритмов к решению задач дискретной оптимизации – Д.И. Батищев, Е.А. Неймарк, Н.В. Старостин
5. Генетические алгоритмы. Учебное пособие – Л.А. Гладков, В.В. Курейчик, В.М. Курейчик
6. Генетические алгоритмы и оптимизация. Учебное пособие – А.В. Еремеев
7. **Листинг программы**

Листинг 1 – Класс Gen

public class Gen

{

public int task { get; init; }

public int gen { get; set; }

public Gen(int t, int g)

{

task = t;

gen = g;

}

}

Листинг 2 – Класс Matrix

public class Matrix

{

public int N { get; set; }

public int M { get; set; }

public int T1 { get; set; }

public int T2 { get; set; }

public List<List<int>> matrix {get; set;}

public Matrix(int n, int m, int t1 = 10, int t2 = 20)

{

N = n;

M = m;

T1 = t1;

T2 = t2;

generate\_matrix();

}

private void generate\_matrix()

{

var rnd = new Random();

var matrix\_new = new List<List<int>>();

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

{

var str\_matrix = new List<int>();

var rnd\_number = rnd.Next(T1, T2);

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

{

str\_matrix.Add(rnd\_number);

}

matrix\_new.Add(str\_matrix);

}

matrix = matrix\_new;

}

public void print\_matrix()

{

foreach(var str in matrix)

{

Console.WriteLine(string.Join(",", str));

}

}

}

Листинг 3 – Класс Gen\_algorith

public class Gen\_algorith

{

Random rnd = new Random();

int N\_chr { get; init; }

int N\_lim { get; init; }

List<List<Gen>> Ch\_i { get; set; }

public List<int> F\_Ch\_i { get; set; }

double P\_cross { get; init; }

double P\_mutation { get; init; }

public List<int> Generation\_coef { get; init; }

int number\_o\_generation = 1;

Matrix matrix { get; init; }

string logsDir = "Logs";

string fitnessFilePath;

public Gen\_algorith(int n\_chr, int n\_lim, double p\_cross, double p\_mutation, Matrix mt, List<int> vectror\_coef)

{

N\_chr = n\_chr;

N\_lim = n\_lim;

P\_cross = p\_cross;

P\_mutation = p\_mutation;

matrix = mt;

F\_Ch\_i = new List<int>();

Generation\_coef = vectror\_coef;

if (!Directory.Exists(logsDir))

Directory.CreateDirectory(logsDir);

else

{

foreach (var file in Directory.GetFiles(logsDir))

{

File.Delete(file);

}

}

fitnessFilePath = Path.Combine(logsDir, "fitness\_log.txt");

}

private void generate\_begin()

{

var ch\_i = new List<List<Gen>>();

for (int i = 0; i < N\_chr; i++)

{

var osob = new List<Gen>();

for (int j = 0; j < matrix.M; j++)

{

osob.Add(new Gen(matrix.matrix[j][rnd.Next(0, matrix.N)], rnd.Next(0, 256)));

}

ch\_i.Add(osob);

}

Ch\_i = ch\_i;

F\_Ch\_i = get\_vector\_survive(Ch\_i);

}

private List<int> get\_vector\_survive(List<List<Gen>> skibidi)

{

int counter = 1;

List<int> new\_F\_ch\_i = new List<int>();

foreach (var specimen in skibidi)

{

Console.WriteLine("особь #" + counter);

print\_counting\_geno\_one(specimen);

new\_F\_ch\_i.Add(get\_feno\_one(specimen));

counter++;

}

return new\_F\_ch\_i;

}

private void print\_counting\_geno\_one(List<Gen> osob)

{

Console.WriteLine("Генотип");

for (int i = 0; i < matrix.M; i++)

{

Console.Write(osob[i].task.ToString().PadLeft(3).PadRight(5));

}

Console.WriteLine();

for (int i = 0; i < matrix.M; i++)

{

Console.Write("|".ToString().PadLeft(3).PadRight(5));

}

Console.WriteLine();

for (int i = 0; i < matrix.M; i++)

{

Console.Write(osob[i].gen.ToString().PadLeft(3).PadRight(5));

}

Console.WriteLine();

}

private int get\_feno\_one(List<Gen> osob, bool need\_to\_print = true)

{

int interval\_len = 255 / matrix.N;

List<List<int>> feno = new List<List<int>>();

for (int i = 0; i < matrix.N; i++)

{

feno.Add(new List<int>());

}

foreach (var el in osob)

{

int intervalIndex = el.gen / interval\_len;

if (intervalIndex >= matrix.N)

{

feno[matrix.N - 1].Add(el.task);

}

else

{

feno[intervalIndex].Add(el.task);

}

}

if (need\_to\_print)

{

print\_counting\_feno(feno);

Console.WriteLine();

}

return feno.Select(column => column.Sum()).ToList().Max();

}

private void print\_counting\_feno(List<List<int>> feno)

{

Console.WriteLine("Фенотип");

for (int i = 1; i <= matrix.N; i++)

{

Console.Write(i.ToString().PadLeft(3).PadRight(5));

}

Console.WriteLine();

int maxRows = feno.Max(column => column.Count);

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

{

foreach (var column in feno)

{

if (i < column.Count)

{

string formattedNumber = column[i].ToString().PadLeft(3).PadRight(5);

Console.Write(formattedNumber);

}

else

{

Console.Write(" ");

}

}

Console.WriteLine();

}

for (int i = 0; i < matrix.N; i++)

{

Console.Write("\_\_\_\_\_");

}

Console.WriteLine();

foreach (var sum in feno.Select(column => column.Sum()).ToList())

{

string formattedsum = sum.ToString().PadLeft(3).PadRight(5);

Console.Write(formattedsum);

}

Console.WriteLine();

}

public void print\_F\_ch\_i(List<int> f\_ch\_i)

{

Console.WriteLine("Вектор приспособленности: " + string.Join(",", f\_ch\_i));

}

private void RedirectConsoleOutput(string fileName)

{

var fileStream = new FileStream(fileName, FileMode.Create, FileAccess.Write);

var streamWriter = new StreamWriter(fileStream, Encoding.UTF8) { AutoFlush = true };

Console.SetOut(streamWriter);

}

private void RestoreConsoleOutput()

{

var standardOutput = new StreamWriter(Console.OpenStandardOutput(), Encoding.UTF8) { AutoFlush = true };

Console.SetOut(standardOutput);

Console.OutputEncoding = Encoding.UTF8;

}

private List<List<Gen>> generate\_new\_gen(int num\_o\_el\_list)

{

var rem = show\_dead\_and\_alive(Ch\_i);

List<List<Gen>> new\_Ch\_i = new List<List<Gen>>();

foreach(var i in rem)

{

new\_Ch\_i.Add(Ch\_i[i]);

}

Ch\_i = new\_Ch\_i;

print\_F\_ch\_i(get\_vector\_survive(Ch\_i));

List<List<Gen>> new\_generation = new List<List<Gen>>(Ch\_i);

while (new\_generation.Count < N\_chr \* Generation\_coef[num\_o\_el\_list])

{

for (int i = 0; i < N\_chr; i++)

{

Console.WriteLine($"Генерируется {new\_generation.Count + i + 1}я особь");

var idx = Chose\_strongest\_second\_parent(Ch\_i, i);

new\_generation.Add(cross\_over(Ch\_i[i % Generation\_coef[num\_o\_el\_list]], Ch\_i[idx]));

}

}

return new\_generation

.Take(N\_chr \* Generation\_coef[num\_o\_el\_list])

.ToList(); ;

}

private List<int> show\_dead\_and\_alive(List<List<Gen>> new\_generation)

{

List<int> indices = Enumerable.Range(0, new\_generation.Count).ToList();

indices.Sort((i1, i2) => get\_feno\_one(new\_generation[i1], false).CompareTo(get\_feno\_one(new\_generation[i2], false)));

List<int> minIndices = indices.Take(N\_chr).ToList();

for (int i = 0; i < new\_generation.Count; i++)

{

Console.Write($"Особь #{i + 1} с приспособленностью {get\_feno\_one(new\_generation[i], false)}");

if (minIndices.Contains(i)) {

Console.Write(" Остается\n");

}

else { Console.Write(" Отсеивается\n"); }

}

return minIndices;

}

private int Chose\_strongest\_second\_parent(List<List<Gen>> left\_parents\_list, int current\_idx\_parent)

{

Console.WriteLine("Производится отбор второго правого родителя\n");

var parent\_idx = get2nums\_rnd(0, left\_parents\_list.Count, new List<int>([current\_idx\_parent]));

Console.WriteLine("Первый кандидат c индексом " + parent\_idx[0]);

int pheno\_fisrt = get\_feno\_one(left\_parents\_list[parent\_idx[0]]);

Console.WriteLine("Второй кандидат с индексом " + parent\_idx[1]);

int pheno\_second = get\_feno\_one(left\_parents\_list[parent\_idx[1]]);

int strongest\_parent\_idx = pheno\_fisrt > pheno\_second ? parent\_idx[1] : parent\_idx[0];

Console.WriteLine("");

Console.WriteLine($"Победил в итоге родитель с уровнем приспособленности = {get\_feno\_one(left\_parents\_list[strongest\_parent\_idx], false)} под номером {strongest\_parent\_idx}");

return strongest\_parent\_idx;

}

public List<int> get2nums\_rnd(int start, int size, List<int> uncorrect\_values = null)

{

if (uncorrect\_values == null) {

uncorrect\_values = new List<int> { -1 };

}

int index1;

do {

index1 = rnd.Next(size);

} while (uncorrect\_values.Contains(index1));

int index2;

do

{

index2 = rnd.Next(size);

} while (index2 == index1 || uncorrect\_values.Contains(index2));

return new List<int>([index1, index2]);

}

private List<Gen> cross\_over(List<Gen> parent\_1, List<Gen> parent\_2)

{

double p\_current\_mutation = rnd.NextDouble();

double p\_current\_cross = rnd.NextDouble();

if (p\_current\_cross < P\_cross)

{

Console.WriteLine("Родитель 1");

print\_counting\_geno\_one(parent\_1);

int feno\_parent\_1 = get\_feno\_one(parent\_1, true);

Console.WriteLine("Родитель 2");

print\_counting\_geno\_one(parent\_2);

int feno\_parent\_2 = get\_feno\_one(parent\_2, true);

var divide\_points = get2nums\_rnd(1, matrix.M).Order().ToList();

Console.WriteLine($"Происходит скрещивание. Точки разделения: {divide\_points[0]}, {divide\_points[1]}");

List<Gen> potom1 = new List<Gen>();

List<Gen> potom2 = new List<Gen>();

for (int i = 0; i < matrix.M; i++)

{

if (i < divide\_points[0] || i > divide\_points[1])

{

potom1.Add(parent\_1[i]);

potom2.Add(parent\_2[i]);

}

else

{

potom1.Add(parent\_2[i]);

potom2.Add(parent\_1[i]);

}

}

Console.WriteLine("\nПотомок 1");

print\_counting\_geno\_one(potom1);

Console.WriteLine("\nПотомок 2");

print\_counting\_geno\_one(potom2);

Console.WriteLine("Потомок 1 до возможной мутации");

get\_feno\_one(potom1);

var potom1\_mut = mutation(potom1, p\_current\_mutation);

Console.WriteLine("Потомок 1 после возможной мутации");

print\_counting\_geno\_one(potom1\_mut);

int feno\_first = get\_feno\_one(potom1\_mut);

Console.WriteLine("Потомок 2 до возможной мутации");

int start\_feno\_second = get\_feno\_one(potom2);

var potom2\_mut = mutation(potom2, p\_current\_mutation);

Console.WriteLine("Потомок 2 после возможной мутации");

print\_counting\_geno\_one(potom2\_mut);

int feno\_second = get\_feno\_one(potom2\_mut);

List<(int, List<Gen>)> best\_variant =

[ (feno\_parent\_1, parent\_1),

(feno\_first, potom1\_mut),

(feno\_second, potom2\_mut)

];

Console.WriteLine($"Левый родитель - {feno\_parent\_1}, Правый родитель - {feno\_parent\_2}, Потомок 1 - {feno\_first}, Потомок 2 - {feno\_second}");

int minIndex = Enumerable.Range(0, best\_variant.Count)

.MinBy(i => best\_variant[i].Item1);

switch (minIndex)

{

case 0:

Console.WriteLine("В следующее поколение переходит левый родитель");

return parent\_1;

case 1:

{

Console.WriteLine("В следующее поколение переходит 1й потомок");

return potom1\_mut;

}

default:

{

Console.WriteLine("В следующее поколение переходит 2й потомок");

return potom2\_mut;

}

}

}

else

{

Console.WriteLine("Кроссинговер не получился.\n");

Console.WriteLine("Родитель 1");

print\_counting\_geno\_one(parent\_1);

int feno\_parent = get\_feno\_one(parent\_1, true);

var parent1\_mut = mutation(parent\_1, p\_current\_mutation);

Console.WriteLine("Родитель после возможной мутации");

print\_counting\_geno\_one(parent1\_mut);

var parent1\_mut\_feno = get\_feno\_one(parent1\_mut);

Console.WriteLine($"Левый родитель - {feno\_parent}, После мутации - {parent1\_mut\_feno}");

if (feno\_parent <= parent1\_mut\_feno)

{

Console.WriteLine("В следующее поколение переходит левый родитель");

return parent\_1;

}

else

{

Console.WriteLine("В следующее поколение переходит мутировавший родитель");

return parent1\_mut;

}

}

}

private List<Gen> mutation(List<Gen> potom, double p\_current\_mutation)

{

List<Gen> new\_potom = potom.Select(g => new Gen(g.task, g.gen)).ToList();

if (p\_current\_mutation < P\_mutation)

{

Console.WriteLine("Происходит мутация");

int gen\_idx = rnd.Next(0, matrix.M);

Console.WriteLine("Мутирует " + gen\_idx + "-й ген");

string binaryString = Convert.ToString(new\_potom[gen\_idx].gen, 2).PadLeft(8, '0');

Console.WriteLine(new\_potom[gen\_idx].gen + " = " + binaryString);

byte number = (byte)new\_potom[gen\_idx].gen;

var idx\_toChange = get2nums\_rnd(0, 8);

Console.WriteLine($"Индексы меняющихся битов: {idx\_toChange[0]} и {idx\_toChange[1]}");

foreach (var i in idx\_toChange)

{

number ^= (byte)(1 << i);

}

Console.WriteLine(number + " = " + Convert.ToString(number, 2).PadLeft(8, '0'));

new\_potom[gen\_idx].gen = number;

get\_feno\_one(potom, false);

}

else

{

Console.WriteLine("Мутация не происходит\n");

}

return new\_potom;

}

public void main\_algorithm()

{

string genLogPath = Path.Combine(logsDir, $"generation\_0.txt");

RedirectConsoleOutput(genLogPath);

Console.WriteLine("0 поколение");

generate\_begin();

File.AppendAllText(fitnessFilePath,

$"0) {string.Join(",", F\_Ch\_i)} = {F\_Ch\_i.Min()}\n", Encoding.UTF8);

print\_F\_ch\_i(F\_Ch\_i);

RestoreConsoleOutput();

int counter = 1;

while (counter < N\_lim)

{

string genLogPathCycle = Path.Combine(logsDir, $"generation\_{number\_o\_generation}.txt");

RedirectConsoleOutput(genLogPathCycle);

Console.WriteLine($"Генерируется поколение {number\_o\_generation}");

var new\_generation = generate\_new\_gen(number\_o\_generation % Generation\_coef.Count);

Console.WriteLine($"Сгенерировано поколение {number\_o\_generation}");

var new\_F\_ch\_i = get\_vector\_survive(new\_generation);

print\_F\_ch\_i(new\_F\_ch\_i);

File.AppendAllText(fitnessFilePath,

$"{number\_o\_generation}) {string.Join(",", new\_F\_ch\_i)} = {new\_F\_ch\_i.Min()}\n", Encoding.UTF8);

RestoreConsoleOutput();

if (F\_Ch\_i.Min() == new\_F\_ch\_i.Min())

{

counter++;

}

else

{

counter = 1;

}

Ch\_i = new\_generation;

F\_Ch\_i = new\_F\_ch\_i;

number\_o\_generation++;

}

Console.WriteLine("\nПравильный ответ:");

var idx\_min = F\_Ch\_i.IndexOf(F\_Ch\_i.Min());

print\_counting\_geno\_one(Ch\_i[idx\_min]);

get\_feno\_one(Ch\_i[idx\_min]);

}

}