ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

**Факультет информатики, математики и компьютерных наук**

*Боровкова Дарья Александровна*

**КУРСОВАЯ РАБОТА**

Задача упаковки в контейнеры

|  |  |
| --- | --- |
|  | Научный руководитель  доцент  А.И.Николаев |

Нижний Новгород, 2017

Оглавление

[1. Вступление. 3](#_Toc484468734)

[2. Существующие подходы. 5](#_Toc484468735)

[3. Генетические алгоритмы. 6](#_Toc484468736)

[4. Генетический алгоритм для создания гипер-эвристики. 8](#_Toc484468737)

[4.1 Набор эвристик. 8](#_Toc484468738)

[4.2 Определение состояния задачи. 11](#_Toc484468739)

[4.3 Генетический алгоритм. 11](#_Toc484468740)

[4.3.1 Представление. 11](#_Toc484468741)

[4.3.2 Операторы. 12](#_Toc484468742)

[4.3.3 Фитнесс функция. 13](#_Toc484468743)

[4.3.4 Работа генетического алгоритма. 13](#_Toc484468744)

[5. Улучшение. 15](#_Toc484468745)

[6. Экспериментальные данные. 16](#_Toc484468746)

[7. Результаты. 17](#_Toc484468747)

[8. Вывод. 20](#_Toc484468748)

[9. Список литературы. 21](#_Toc484468749)

[10. Приложение 1. 22](#_Toc484468750)

# Вступление.

Задача упаковки в контейнеры заключается в упаковке предметов заранее определённой формы в конечное число контейнеров заданной формы таким способом, чтобы количество используемых контейнеров было наименьшим. С математической точки зрения для одномерного случая данную задачу можно сформулировать как: дан конечный набор чисел О (размер предметов) и константа С (размер контейнера). Вопрос: найти разбиение множества О на N подмножеств таких, чтобы сумма элементов в каждом подмножестве не превосходила константу С и N было минимальным.

Существует множество разновидностей задачи упаковки в контейнеры: двумерная упаковка, линейная упаковка, упаковка по весу, упаковка по стоимости и т.д. Интерес к решению проблем такого класса растёт, потому что похожие вопросы появляются и в других видах оптимизационных задач. Также задача упаковки в контейнеры применяется в разных областях: оптимальное заполнение контейнеров, загрузка грузовиков с ограничением по весу, создание резервных копий на съёмных носителях и во многих других.

Данная задача является NP-трудной, так как сводится к простому перебору всех возможных комбинаций предметов и поиска среди них лучшей. Как результат, использование точного алгоритма перебора возможно только для задач небольших размерностей.

Конечно, эта проблема не нова и уже существует много различных подходов к её решению. К сожалению, большинство из используемых алгоритмов дают далеко не оптимальные результаты. В статье [1] предложен новый подход к решению задач упаковки в контейнеры. Авторы используют генетический алгоритм и предлагают не усовершенствовать имеющиеся жадные эвристики, а пробовать их комбинировать, то есть подбирать эвристику, которая лучше подходит для данного этапа решения задачи.

Целью данной работы является улучшение алгоритма, предложенного Peter Ross. Была выбрана задача упаковки одномерных предметов, то есть предметов, имеющих только ценность. Для достижения цели нужно решить следующие задачи:

1. Рассмотреть существующие подходы к решению задач упаковки в контейнеры;
2. Изучить генетические алгоритмы;
3. Реализовать алгоритм предложенной в статье [1];
4. Усовершенствовать существующий алгоритм.

# Существующие подходы.

Самыми популярными алгоритмами решения задач упаковки в контейнеры являются жадные эвристики, которые дают приблизительное решение, но работают за приемлемое время.

Среди них самыми популярными алгоритмами являются Next-Fit-Decreasing, Largest-Fit-Decreasing и Best-Fit-Decreasing. Первым шагом во всех этих алгоритмах является сортировка по убыванию упаковываемых элементов.

Next-Fit-Decreasing – алгоритм, в котором предмет упаковывается в текущую корзину, если это возможно, если нет, то открывается новая корзина и предмет упаковывается в неё. Данный алгоритм обычно показывает плохой результат, но зато работает быстро, потому что ему нужен всего лишь один проход по всем элементам.

Суть Largest-Fit-Decreasing алгоритма состоит в том, что предметы помещаются в первую корзину, в которую подходят. Новая корзина открывается, когда предмет не помещается ни в одну из предыдущих.

Последний алгоритм упаковывает предмет не в первую подходящую корзину, как в предыдущем алгоритме, а в лучшую, то есть туда, где остаётся минимум места после упаковки текущего предмета.

Две последние эвристики являются одними из самых лучших в плане качества упаковки, но более трудоёмкими и затратными по времени. Производительность этих алгоритмов ниже, чем у первого, так как на каждом шагу приходится рассматривать все корзины.

В статье [1] используются 8 эвристик и генетический алгоритм, который определяет для состояний задачи подходящую эвристику.

# Генетические алгоритмы.

Генетические алгоритмы – семейство вычислительных моделей основанных на эволюции. Эти алгоритмы кодируют возможное решение отдельных проблем на простых хромосомах в качестве структур данных и применяют рекомбинационные операции к этим структурам, чтобы сохранять важную информацию.[2]

Как уже говорилось выше, генетические алгоритмы используют для работы эволюционные принципы, среди которых наследственность, изменчивость и естественный отбор. Такие алгоритмы работают с популяцией особей. Популяция состоит из набора хромосом, в каждой из которых закодировано возможное решение задачи. Изначально популяция формируется случайным образом. Для оценки качества закодированных решений используется фитнесс функция, которая необходима для вычисления приспособленности каждой особи. По результатам фитнесс функции выбираются наиболее приспособленные особи для скрещевания. Скрещивание выбранных особей осуществляется с помощью генетического оператора кроссовера. Кроссовер происходит путём обмена хромосомной информацией между родительскими особями и отвечает за создание потомства. Потомки формируют новую популяцию, причём часть из них мутирует. Это значит, что хромосомная информация случайно изменяется.

Эволюция популяции состоит из последовательности следующих этапов: «Оценивание популяции (фитнесс функция)» → «Отбор лучших особей» → «Кроссовер» → «Мутация».

Мутация

Формирование начальной популяции

Оценка популяции – подсчёт фитнесс функции

Отбор лучших особей (селекция)

Кроссовер

Результат

Мутация

Рис. 1

Генетические алгоритмы широко применимы, довольно просты и способны из случайных данных, сгенерированных для первого поколения, создать отличное решение. Но у таких алгоритмов есть существенный минус, результаты решения одной проблемы могут сильно разниться, от близкого к оптимальному и до самого худшего.

# Генетический алгоритм для создания гипер-эвристики.

Эта глава описывает реализацию генетического алгоритма для построения гипер-эвристики применительно к задаче упаковки в контейнеры. Основой для этого метода служит подход, описанный в статье [1].

## Набор эвристик.

Чтобы создать гипер-эвристику, нужно определиться с набором основных эвристик.

1. Largest-Fit-Decreasing была описана выше.

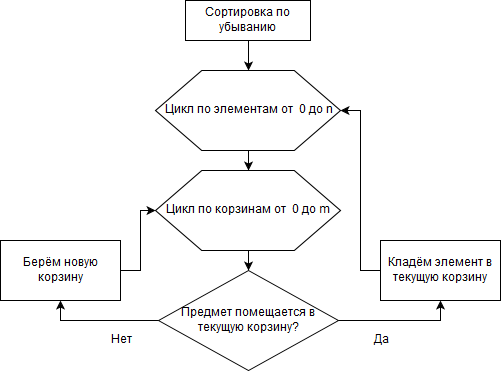


Рис. 2

1. Next-Fit-Decreasing также была описана выше.

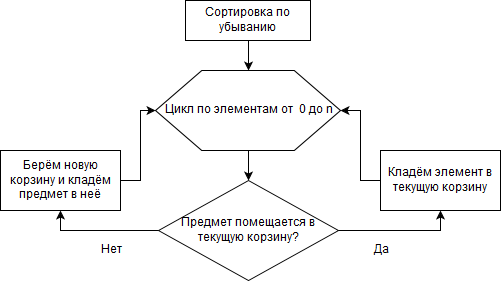


Рис. 3

1. Djang and Finch's алгоритм (DJD). Он состоит в том, что сначала предметы сортируются по убыванию. Затем предметы помещаются в корзину до тех пор, пока не будет заполнена на 1/3. Затем алгоритм ищет один, два или три предмета, которые полностью заполнят эту корзину. Если таких комбинаций нет, то алгоритм пытается найти комбинации, которые заполнят корзину не до конца, а на её вместимость за вычетом единицы. Если алгоритм опять терпит неудачу, то он пытается заполнить корзину до её вместимости за вычетом двух. Это продолжается пока алгоритм не подберёт подходящую комбинацию из одного, двух или трёх предметов или не обнаружит, что её нет. Затем открывается новая корзина и действия повторяются.

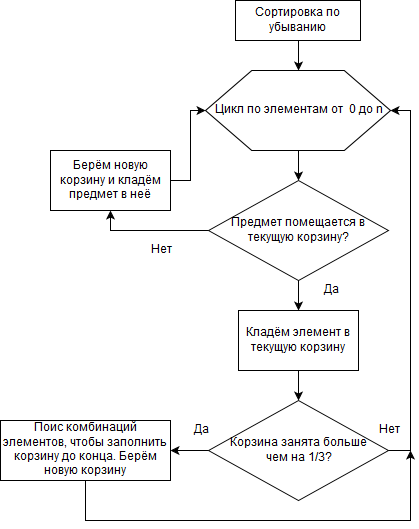


Рис. 4

1. Djang and Finch, more tuples (DJT). Этот алгоритм модификация предыдущего. Здесь ищутся комбинации до пяти элементов, а не до трёх.

Также каждый из четырёх алгоритмов были соединены с фильтром. Его смысл заключается в том, что прежде, чем открыть новую корзину, нужно заполнить текущую предметами, которые туда ещё помещаются.

1. Largest-Fit-Decreasing + Filter.
2. Next-Fit-Decreasing + Filter.
3. DJD + Filter.
4. DJT + Filter.

Итого получается 8 эвриcтик, которые буду дальше использоваться.

## Определение состояния задачи.

Состояние задачи определяется пятью числами. Первые четыре делят неупакованные предметы на категории: огромные - huge (c/2 < s, где c – размер корзины, s – ценность предмета), большие - large (c/3 < s <= c/2), средние - medium (c/4 < s <= c/3) и маленькие - small (s <= c/4). Пятое число соответствует количеству предметов, которые нужно упаковать - remaining.

## Генетический алгоритм.

Чтобы реализовать генетический алгоритм для какой-то конкретной задачи, нужно определить специфичные факторы реализации. Среди них представление задачи в виде хромосом, операторы (кроссоверы и мутации), вычисление фитнесс функции.

### Представление.

Хромосомы состоят из блоков, каждый блок содержит 6 цифр: h, l, m, s, r, a. Первые пять показателей соответствуют параметрам, описанным в пункте 4.2. Число a соответствует одному из номеров алгоритмов, описанных выше, с которым ассоциируется состояние. Параметры h, l, m, s, r могут принимать значение от 0 до 1, параметр a от 1 до 8. Ниже приведён пример правил:

0.04381776799476611 0.3071077940462067 0.5566723183108393 0.2090000109624227 0.5895322462430618 6

0.027655155765654138 0.48032450080098 0.6824898133575289 0.07141361823487313 0.9579817305369669 7

0.35961575987485517 0.5655863809926466 0.4643678689430134 0.12090984871456734 0.059292037166342904 1

0.8474659924414678 0.007893759277902257 0.6690675219911826 0.49392253961733457 0.10314925092045146 6

Каждое состояние, описанное первыми пятью параметрами, соответствует точке в пятимерном пространстве. Блоки в хромосомах представляют собой точки. На каждом шагу решения применяется эвристика, ассоциирующаяся с блоком в хромосоме, котрый ближе всего лежит к текущему состоянию задачи.

### Операторы.

В описываемом генетическом алгоритме использется два вида скрещевания и три вида мутации.

Первый тип скрещивания – обычный кроссовер. Сначала выбирается случайный начальный блок, затем количество блоков. Хромосомы родители обмениваются выбранными блоками, результатом обмена являются две новые хромосомы – потомки.

Второй тип скрещивания работает на уровне блоков. Первый родитель имеет 90% вероятности, что первый ребёнок унаследует его блок, и 10%, что блок передастся второму ребёнку. Для второго родителя обратная ситуация. Оба кроссовера имеют одинаковую вероятность быть выбранными.

Три вида мутации: добавление блока, удаление блока и нормальная мутация. Первый вид мутации – случайно генерирует новый блок и добавляет его в хромосому. Все шесть чисел выбираются с помощью нормального распределения. Первые пять лежат в интервале [0;1], шестое число принимает целые значения от 1 до 8. Второй вид мутации –удаляет случайно выбранный блок в хромосоме. Нормальная мутация – заменяет случайный блок другим случайно сгенерированным.

Ребёнок имеет 10% шанс на мутацию. Если происходит мутацияя, то выбирается один из её типов: добавление и удаление блоков с 25% вероятность, нормальная мутация с 50%.

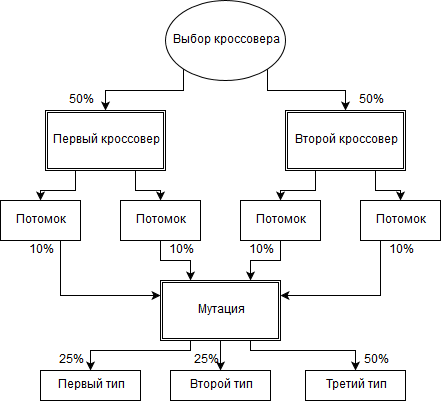


Рис. 5

### Фитнесс функция.

Фитнесс функция должна определять насколько хорошо хромосома решает задачи. В данном алгоритме фитнесс функция равна количеству корзин делённому на количество предметов, которые надо упаковать. Но так как хромосома должна хорошо решать не одну конкретную задачу, а быть довольно универсальной, имеет смысл брать среднее значение фитнесс функций за всё время жизни хромосомы.

### Работа генетического алгоритма.

На каждой итерации генетического алгоритма вся популяция делится на пары (родителей): первый из них – имеющий лучшие результаты, второй – выбирается случайным образом. Каждая пара создаёт по два потомка, потомки решают по четыре задачи, задачи назначаются случайно. Это позволяет определить начальные фитнесс функции. Затем два лучших ребёнка заменяют двух худших хромосом в популяции, вся популяция решает новую проблему и фитнесс функции пересчитываются. Этот процесс повторяется фиксированное число раз.

# Улучшение.

В ходе реализации и исследования генетического алгоритма описанного выше, было обнаружено, что существует некоторая зависимость в последовательности алгоритмов. Поэтому было внесено изменение в существующий алгоритм.

Помимо описанных в пункте 4.3.1 блоков в хромосому был добавлен ещё один блок, содержащий номер предыдущей запущенной эвристики. Пример новых правил приведён ниже:

0.7862245701392309 0.027068069542625084 0.5268779582474801 0.3363396143911175 0.2405005804890955 2 5

0.010817906297280855 0.08536020473708406 0.03358616507132661 0.4158758296858609 0.7216864705761187 4 5

0.8894735354569402 0.4774382509568219 0.9306689854113505 0.97478788025348 0.7471189197415535 1 4

0.24943527309044788 0.7438041716448902 0.5840478310615975 0.9495811613735354 0.8951749420248734 2 4

# Экспериментальные данные.

Генетические алгоритмы описанные выше были запущены по 20 раз, чтобы создать набор лучших хромосом. Размер популяции был 40, количество блоков в хромосоме 16, количество итераций 500 и 1000.

В случае 500 итераций было решено 20000 проблем, а в случае 1000 итераций было решено 40000 проблем, плюс 40 детей на каждой итерации решали 160 проблем.

Для тренировки и тестирования были сгенерированы данные. Количество предметов варьировалось от 500 до 800, размер корзин от 100 до 300.

Лучшая хромосома по итогам тренировки попадала в итоговой набор хромосом, который тестировался на тренировочных и тестовых задачах. Было создано по 4 итоговых набора хромосом для начального генетического алгоритма и для генетического алгоритма с улучшением. Эти наборы состояли из 5 и 10 лучших хромосом и были собраны на 500 и 1000 итерациях генетических алгоритмов.

Также были собраны данные о работе отдельных эвристик на тех же данных, чтобы впоследствии сравнить их с данными полученными с помощью гипер-эвристики.

# Результаты.

Результаты тестирования приведены в следующих таблицах. Результаты показывают, насколько процентов решение жадной эвристики или лучшего набора хромосом хуже идеальной упаковки.

Первая таблица показывает сравнение результатов работы лучших хромосом полученных с помощью генетических алгоритмов. Результаты сравнивались на трёх тестовых наборах данных: [7], [8] и [9]. Первый набор данных содержит 756 файлов, количество предметов в файлах варьируется от 50 до 500, размер корзины от 100 до 150, а веса предметов от 1 до 100. Второй набор данных содержит 300 файлов, количество предметов в файлах лежит в пределе от 50 до 500, размер корзины 1000, а веса предметов C/3, C/5, C/7, C/9, где C – размер корзины. В третьем наборе данных 10 файлов, в них содержится по 200 предметов, размер корзины 100000, а веса предметов от 20000 до 35000.

Обозначения в таблице: GA – хромосомы полученные с помощью начального генетического алгоритма, GAI – с помощью улучшенного. 500и – количество итераций генетического алгоритма, 5х – число лучших хромосом в итоговом наборе.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | GA 500и, 5х | GA 500и, 10х | GA 1000и, 5х | GA 1000и, 10х | GAI 500и, 5х | GAI 500и, 10х | GAI 1000и, 5х | GAI 1000и, 10х |
| Test1 | 0.313 | 0.315 | 0.317 | 0.32 | 0.310 | 0.313 | 0.314 | 0.313 |
| Test2 | 3.51 | 3.54 | 3.51 | 3.56 | 3.47 | 3.42 | 3.51 | 3.47 |
| Test3 | 5.69 | 5.52 | 5.68 | 5.53 | 5.34 | 5.34 | 5.52 | 5.27 |

Таблица 1.

Вторая таблица показывает сравнение результатов отдельно взятых жадных эвристик и лучших наборов хромосом полученных с помощью обычного и улучшенного генетических алгоритмов.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | GA | GAI | LFD | NFD | DJD | DJT | LFD + Filter | NFD + Filter | DJD + Filter | DJT + Filter |
| Test1 | 0.313 | 0.310 | 0.313 | 70.88 | 29.25 | 29.25 | 0.313 | 3.22 | 0.313 | 0.313 |
| Test2 | 3.51 | 3.42 | 3.47 | 20.78 | 7.28 | 7.19 | 3.47 | 6.52 | 3.53 | 3.53 |
| Test3 | 5.52 | 5.27 | 5.34 | 14.6 | 6.74 | 6.52 | 5.34 | 7.68 | 5.52 | 5.34 |

Таблица 2.

Из таблицы видно, что лучшие хромосомы генетических алгоритмов работают также как и лучшая отдельно взятая эвристика. Получается, что гипер-эвристика не даёт преимущества.

# Вывод.

К сожалению, полученные результаты не соответствуют ожиданиям. При определённой конфигурации улучшенная гипер-эвристика работает чуть лучше, чем лучшая жадная эвристика. Это может быть вызвано тем, что количество данных для обучения и их разнообразие было не большим. Также причиной таких результатов может быть недостаточное количество итераций в генетических алгоритмах, хотя в статье [1] оптимальным числом указано 500 поколений. С другой стороны к плюсам можно отнести тот факт, что улучшенная гипер-эвристика работает, лучше, чем изначальная.

Несмотря на слабые результаты, данное исследование даёт надежду и почву для дальнейших рассуждений. Возможно при увеличении количества тестовых данных, итераций генетического алгоритма и увеличении числа лучших хромосом в итоговом наборе, результаты будут соответствовать ожиданиям.

# Список литературы.

1. *Peter Ross, Javier G. Mar´ın-Bl´azquez, Sonia Schulenburg, and Emma Hart*. Learning a Procedure That Can Solve Hard Bin-Packing Problems: A New GA-Based Approach to Hyper-heuristics, [Электронный ресурс]. Режим доступа:
2. *Darrell Whitley*. A Genetic Algorithm Tutorial. Statistics and Computing (4):65-85, 1994. [Электронный ресурс]. Режим доступа: <http://bipad.cmh.edu/ga_tutorial1994.pdf>.
3. *E.G. Coffman, M.R. Garey, and D.S. Johnson*. Approximation algorithms for bin packing: a survey. In D. Hochbaum, editor, Approximation algorithms for NP-hard problems, pages 46–93. PWS Publishing, Boston, 1996.
4. *Ackley, D.* A Connectionist Machine for Genetic Hillclimbing. Kluwer Academic Publishers, 1987.
5. *Philipp A. Djang and Paul R. Finch*. Solving One Dimensional Bin Packing Problems. 1998.
6. *D.S. Johnson*. Near-optimal bin-packing algorithms. PhD thesis, MIT Department of Mathematics, 1973.
7. https://www2.wiwi.uni-jena.de/Entscheidung/binpp/bin1dat.htm
8. https://www2.wiwi.uni-jena.de/Entscheidung/binpp/bin2dat.htm
9. https://www2.wiwi.uni-jena.de/Entscheidung/binpp/bin3dat.htm

# Приложение 1.

LargestFitDecreasing.java

//This is a java program to implement largest fit decreasing for 1D objects using N bins

public class LargestFitDecreasing {

public int startPacking(int[][] a, int[] bins, int size)

{

//bin packing

return binPacking(a, bins, size, a.length);

}

//a - array of elements, size - size of baskets, n - number of elements

public int binPacking(int[][] a, int[] bins, int size, int n)

{

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

for (int j = 0; j < bins.length; j++)

{

if (a[i][1] != 1)

{

if (bins[j] - a[i][0] >= 0)

{

bins[j] -= a[i][0];

a[i][1] = 1;

return j;

}

if (a[i][0] > size)

{

System.exit(1);

}

}

}

return -1;

}

}

GeneticAlgorithmWithImprovement.java

import java.io.BufferedOutputStream;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.io.OutputStream;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.util.ArrayList;

import java.util.List;

public class GeneticAlgorithm {

public static final int POPULATION\_SIZE = 40;

public static final int NUMBER\_OF\_TASKS = 4;

public static final int NUMBER\_OF\_GA\_ITERATIONS = 1000;

public static final String FILE\_PATH = "C:\\data\_for\_binpacking\\improve\_results\\result";

public static final int NUMBER\_OF\_FILES = 750;

public static final String DATA\_PATH = "C:\\data\_for\_binpacking\\data\\";

public static int size;

List<Chromosome> population;

List<Chromosome> childrens;

public int[][] readFile(String fileName){

BufferedReader br = null;

FileReader fr = null;

int n = 0;

int[][] elements = null;

try {

fr = new FileReader(fileName);

br = new BufferedReader(fr);

String sCurrentLine;

br = new BufferedReader(new FileReader(fileName));

int i = 0;

String str = null;

while ((sCurrentLine = br.readLine()) != null) {

if (sCurrentLine.contains("Bin size:"))

{

str = sCurrentLine.substring(10, sCurrentLine.length());

size = Integer.parseInt(str);

}

else if (sCurrentLine.contains("Number of elements:"))

{

str = sCurrentLine.substring(20, sCurrentLine.length());

n = Integer.parseInt(str);

elements = new int[n][2];

}

else

{

elements[i][0] = Integer.parseInt(sCurrentLine);

elements[i][1] = 0;

i++;

}

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (br != null)

br.close();

if (fr != null)

fr.close();

} catch (IOException ex) {

ex.printStackTrace();

}

}

System.out.println("size = " + size);

System.out.println("n = " + n);

return elements;

}

public void writeFile(int num) throws IOException

{

String s = "";

for (int i = 0; i < population.size(); i++)

{

s = s + "population " + i + " fitness " + population.get(i).fitness + " years " + population.get(i).years + " length " + population.get(i).chromosome.size() + "\n";

}

s = s + "\n";

String path = FILE\_PATH + num + ".txt";

FileWriter writer = new FileWriter(path);

writer.write(s);

writer.flush();

writer.close();

}

public void go(int num) throws IOException{

initializePopulation();

childrens = new ArrayList<Chromosome>();

for (int i = 0; i < NUMBER\_OF\_GA\_ITERATIONS; i++)

{

int[] flag = new int[population.size()];

for (int j = 0; j < population.size(); j++)

{

flag[j] = 0;

}

for (int j = 0; j < population.size()/2; j++)

{

int tmp = (int)(Math.random() \* 2);

if (tmp == 0)

{

first\_crossover(findTheBest(population, flag), findRandom(population, flag));

}

else

{

second\_crossover(findTheBest(population, flag), findRandom(population, flag));

}

}

for (int j = 0; j < population.size(); j++)

{

flag[j] = 0;

}

for (int j = 0; j < childrens.size(); j++)

{

for (int k = 0; k < NUMBER\_OF\_TASKS; k++)

{

int tmp = (int)(Math.random() \* NUMBER\_OF\_FILES);

String fileName = DATA\_PATH + (tmp+1) + ".txt";

int[][]elements = readFile(fileName);

childrens.get(j).solveProblem(elements, size,0);

}

}

int[] flagss = new int[population.size()];

for (int j = 0; j < population.size(); j++)

{

flagss[j] = 0;

}

int[] flagsss = new int[childrens.size()];

for (int j = 0; j < childrens.size(); j++)

{

flagsss[j] = 0;

}

Chromosome theworst1 = findTheWorst(population, flagss);

Chromosome theworst2 = findTheWorst(population, flagss);

Chromosome thebest1 = findTheBest(childrens, flagsss);

Chromosome thebest2 = findTheBest(childrens, flagsss);

if (theworst1.fitness > thebest1.fitness && theworst2.fitness > thebest2.fitness

|| theworst1.fitness > thebest2.fitness && theworst2.fitness > thebest1.fitness)

{

population.remove(theworst1);

population.add(thebest1);

population.remove(theworst2);

population.add(thebest2);

}

else if (theworst1.fitness > thebest1.fitness)

{

population.remove(theworst1);

population.add(thebest1);

}

else if (theworst1.fitness > thebest2.fitness)

{

population.remove(theworst1);

population.add(thebest2);

}

else if (theworst2.fitness > thebest2.fitness)

{

population.remove(theworst2);

population.add(thebest2);

}

else if (theworst2.fitness > thebest1.fitness)

{

population.remove(theworst2);

population.add(thebest1);

}

childrens.clear();

int x = (int)(Math.random() \* NUMBER\_OF\_FILES);

String fileName = DATA\_PATH + (x+1) + ".txt";

int[][]elements = readFile(fileName);

for (int j = 0; j < population.size(); j ++)

{

population.get(j).solveProblem(elements, size,0);

for (int k = 0; k < elements.length; k++)

{

elements[k][1] = 0;

}

}

}

int[] flags = new int[population.size()];

for (int j = 0; j < population.size(); j++)

{

flags[j] = 0;

}

writeFile(num);

}

public Chromosome findTheBest(List<Chromosome> popul, int[] flag)

{

Chromosome tmp = new Chromosome(0);

tmp.fitness = 1000000000;

int x = 0;

for (int i = 0; i < popul.size(); i++)

{

if ((flag[i] != 1) && (popul.get(i).fitness < tmp.fitness))

{

tmp = popul.get(i);

x = i;

}

if (popul.get(i).fitness == tmp.fitness)

{

int t = (int)(Math.random() \* 2);

if (t == 1)

{

tmp = popul.get(i);

x = i;

}

}

}

flag[x] = 1;

return popul.get(x);

}

public int findTheLastTheBest(List<Chromosome> popul, int[] flag)

{

Chromosome tmp = new Chromosome(0);

tmp.fitness = 1000000000;

int x = 0;

for (int i = 0; i < popul.size(); i++)

{

if ((flag[i] != 1) && (popul.get(i).fitness < tmp.fitness))

{

tmp = popul.get(i);

x = i;

}

if (popul.get(i).fitness == tmp.fitness)

{

int t = (int)(Math.random() \* 2);

if (t == 1)

{

tmp = popul.get(i);

x = i;

}

}

}

flag[x] = 1;

return x;

}

public Chromosome findTheWorst(List<Chromosome> popul, int[] flag)

{

Chromosome tmp = new Chromosome(0);

tmp.fitness = 0;

int x = 0;

for (int i = 0; i < popul.size(); i++)

{

if ((flag[i] != 1) && (popul.get(i).fitness > tmp.fitness))

{

tmp = popul.get(i);

x = i;

}

if (popul.get(i).fitness == tmp.fitness)

{

int t = (int)(Math.random() \* 2);

if (t == 1)

{

tmp = popul.get(i);

x = i;

}

}

}

flag[x] = 1;

return popul.get(x);

}

public Chromosome findRandom(List<Chromosome> popul, int[] flag)

{

int tmp = 0;

while (true)

{

tmp = (int)(Math.random() \* popul.size());

if (flag[tmp] != 1)

{

flag[tmp] = 1;

return popul.get(tmp);

}

}

}

public void initializePopulation(){

String fileName = "C:\\data\_for\_binpacking\\new\_data\\1.txt";

int[][]elements = readFile(fileName);

population = new ArrayList<Chromosome>();

for (int i = 0; i < POPULATION\_SIZE; i++)

{

Chromosome tmp = new Chromosome(1);

tmp.solveProblem(elements, size,0);

population.add(tmp);

for (int j = 0; j < elements.length; j++)

{

elements[j][1] = 0;

}

}

}

void first\_crossover(Chromosome parent1, Chromosome parent2) {

Chromosome children1 = new Chromosome(0);

Chromosome children2 = new Chromosome(0);

int div = (int)(Math.random() \* parent1.chromosome.size());

while (div >= parent1.chromosome.size()-1 || div >= parent2.chromosome.size()-1)

{

div = (int)(Math.random() \* parent1.chromosome.size());

}

int count = (int)(Math.random() \* parent1.chromosome.size());

while (count >= parent1.chromosome.size()-1 || count >= parent2.chromosome.size()-1)

{

count = (int)(Math.random() \* parent1.chromosome.size());

}

for (int i = 0; i < parent1.chromosome.size(); i++)

{

children1.chromosome.add(parent1.chromosome.get(i));

}

for (int i = 0; i < parent2.chromosome.size(); i++)

{

children2.chromosome.add(parent2.chromosome.get(i));

}

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

{

if ((div + i >= children1.chromosome.size()) && (div + i >= parent2.chromosome.size()))

{

children1.chromosome.set((div + i) - children1.chromosome.size(), parent2.chromosome.get((div + i) - parent2.chromosome.size()));

}

else if (div + i >= children1.chromosome.size())

{

children1.chromosome.set((div + i) - children1.chromosome.size(), parent2.chromosome.get(div + i));

}

else if (div + i >= parent2.chromosome.size())

{

children1.chromosome.set((div + i), parent2.chromosome.get((div + i) - parent2.chromosome.size()));

}

else

{

children1.chromosome.set((div + i), parent2.chromosome.get(div + i));

}

}

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

{

if ((div + i >= children2.chromosome.size()) && (div + i >= parent1.chromosome.size()))

{

children2.chromosome.set((div + i) - children2.chromosome.size(), parent1.chromosome.get((div + i) - parent1.chromosome.size()));

}

else if (div + i >= children2.chromosome.size())

{

children2.chromosome.set((div + i) - children2.chromosome.size(), parent1.chromosome.get(div + i));

}

else if (div + i >= parent1.chromosome.size())

{

children2.chromosome.set((div + i), parent1.chromosome.get((div + i) - parent1.chromosome.size()));

}

else

{

children2.chromosome.set((div + i), parent1.chromosome.get(div + i));

}

}

int tmp = (int)(Math.random() \* 10);

if (tmp == 0)

{

tmp = (int)(Math.random() \* 20);

if (tmp < 5)

{

first\_mutate(children1);

}

else if (tmp < 10)

{

second\_mutate(children1);

}

else

{

third\_mutate(children1);

}

}

tmp = (int)(Math.random() \* 10);

if (tmp == 0)

{

tmp = (int)(Math.random() \* 20);

if (tmp < 5)

{

first\_mutate(children2);

}

else if (tmp < 10)

{

second\_mutate(children2);

}

else

{

third\_mutate(children2);

}

}

childrens.add(children1);

childrens.add(children2);

}

void second\_crossover(Chromosome parent1, Chromosome parent2) {

Chromosome children1 = new Chromosome(0);

Chromosome children2 = new Chromosome(0);

int tmp = 0;

if (parent1.chromosome.size() > parent2.chromosome.size())

{

tmp = parent1.chromosome.size();

}

else

{

tmp = parent2.chromosome.size();

}

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

{

int var = (int)(Math.random() \* 10);

if (var == 9)

{

if ((parent1.chromosome.size() > i) && (parent2.chromosome.size() > i))

{

children1.chromosome.add(parent2.chromosome.get(i));

children2.chromosome.add(parent1.chromosome.get(i));

}

else if (parent1.chromosome.size() > i)

{

children2.chromosome.add(parent1.chromosome.get(i));

}

else if (parent2.chromosome.size() > i)

{

children1.chromosome.add(parent2.chromosome.get(i));

}

}

else

{

if ((parent1.chromosome.size() > i) && (parent2.chromosome.size() > i))

{

children1.chromosome.add(parent1.chromosome.get(i));

children2.chromosome.add(parent2.chromosome.get(i));

}

else if (parent1.chromosome.size() > i)

{

children1.chromosome.add(parent1.chromosome.get(i));

}

else if (parent2.chromosome.size() > i)

{

children2.chromosome.add(parent2.chromosome.get(i));

}

}

}

tmp = (int)(Math.random() \* 10);

if (tmp == 0)

{

tmp = (int)(Math.random() \* 20);

if (tmp < 5)

{

first\_mutate(children1);

}

else if (tmp < 10)

{

second\_mutate(children1);

}

else

{

third\_mutate(children1);

}

}

tmp = (int)(Math.random() \* 10);

if (tmp == 0)

{

tmp = (int)(Math.random() \* 20);

if (tmp < 5)

{

first\_mutate(children2);

}

else if (tmp < 10)

{

second\_mutate(children2);

}

else

{

third\_mutate(children2);

}

}

childrens.add(children1);

childrens.add(children2);

}

public Chromosome first\_mutate(Chromosome parent)

{

Genome gen = new Genome(1);

parent.chromosome.add(gen);

return parent;

}

public Chromosome second\_mutate(Chromosome parent)

{

int tmp = (int)(Math.random() \* parent.chromosome.size());

parent.chromosome.remove(tmp);

return parent;

}

public Chromosome third\_mutate(Chromosome parent)

{

int tmp = (int)(Math.random() \* parent.chromosome.size());

parent.chromosome.remove(tmp);

Genome gen = new Genome(1);

parent.chromosome.add(gen);

return parent;

}

}