МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

им. Н.Э. Баумана

Кафедра «Системы обработки информации и управления»

ОТЧЕТ

**Лабораторная работа № 4**

по курсу«Метода поддержки принятия решений»

Тема: «Решение оптимизационных задач с помощью генетических алгоритмов»

ИСПОЛНИТЕЛЬ: Румянцев О.К.\_\_\_\_\_

ФИО

группа ИУ5-72 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись

"15" декабря 2019 г.

ПРЕПОДАВАТЕЛЬ: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ФИО

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

подпись

"\_\_"\_\_\_\_\_\_\_\_\_201\_ г.

Москва - 2019

1. 1. Цель и задачи лабораторной работы

1.1. Целью лабораторной работы является углубление и закрепление теоретических знаний, полученных на лекциях, приобретение практических навыков самостоятельной работы при решении оптимизационных задач больших размерностей с помощью генетических алгоритмов.

1.2. В процессе выполнения лабораторной работы по теме «Решение оптимизационных задач с помощью генетических алгоритмов» на примере задачи поиска кратчайшего пути для информационного пакета (сообщения) в компьютерной сети студенты решают следующие задачи (задания):

- описывают предметную область;

- определяют исходные данные задачи;

- формулируют задачу и исходные данные в терминах генетических алгоритмов;

- определяют последовательность работы генетического алгоритма;

- разрабатывают компьютерную программу;

- исследуют работу генетического алгоритма и полученное решение.

1. Задание к лабораторной работе

2.1. Сформулировать задачу и описать исходные данные в терминах генетических алгоритмов.

2.2. Разработать программу, которая осуществляет поиск кратчайшего пути для информационного пакета (сообщения) в компьютерной сети с помощью генетического алгоритма.

2.3. При проведения серии экспериментов (не меньше 10) по исследованию работы генетического алгоритма программа должна позволять пользователю задавать топологию сети (пропускные способности каналов связи), содержащей не менее 10 компьютеров (серверов), а также указывать компьютер-отправитель и компьютер-получатель. Должны отображаться все решения (хромосомы) одного поколения до и после применения каждого оператора (скрещивания, селекции, редукции и мутации). Переход к следующему поколению должен осуществляться: в автоматическом режиме в соответствии с заданным критерием; в ручном режиме.

1. Описание предметной области

В данной лабораторной работе производится поиск кратчайшего пути для информационного пакета (сообщения) в компьютерной сети между заданными отправителем и получателем. Компьютерная сеть является полносвязной (каждый компьютер связан со всеми остальными компьютерами в сети, в том числе с самим собой), у каждой связи задана пропускная способность.

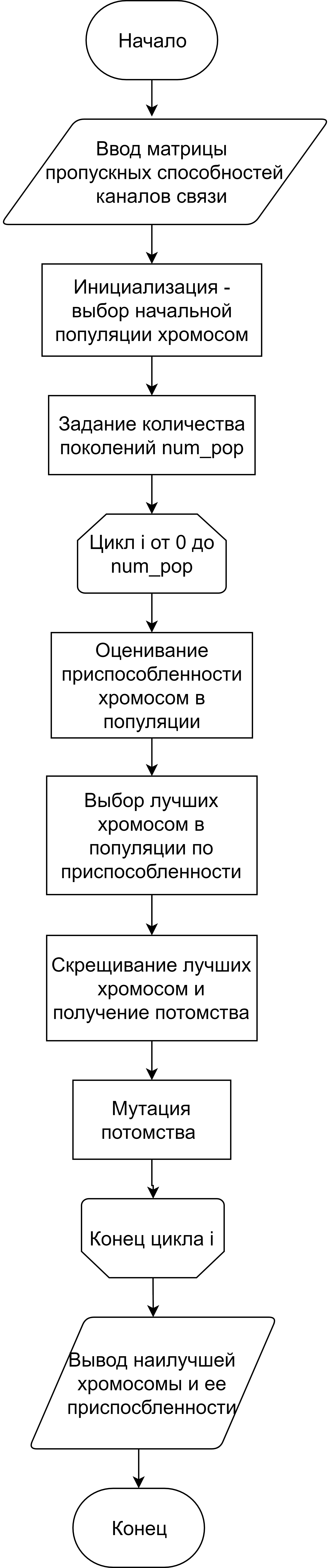
1. Формулировка задачи, описание исходных данных в терминах генетических алгоритмов и блок-схема генетического алгоритма

Задача: найти кратчайший по времени путь для информационного пакета с помощью генетического алгоритма.

Исходные данные:

1. Матрица пропускных способностей каналов связи
2. Хромосома – путь от компьютера – отправителя к компьютеру-получателю, состоит из генов, не содержит гены (номера) отправителя и получателя
3. Ген – номер промежуточного компьютера в цепи от компьютера – отправителя к компьютеру-получателю
4. Популяция – список из возможных цепей передачи сообщения, т.е. хромосом
5. Функция приспособленности (fitness) – функция, вычисляющая время передачи сообщения
6. Функция селекции – функция, выбирающая наилучшие хромосомы из популяции по итогам работы fitness
7. Функция скрещивания – функция, формирующая потомков на основе выбранных функцией селекции предков
8. Функция мутации – функция, изменяющая случайным образом гены в потомстве

Блок-схема генетического алгоритма



1. Описание программы, ее ключевые особенности и новшества

Программа была реализована на Python 3.6 в виде Jupyter Notebook. Содержит имплементацию генетического алгоритма для поиска кратчайшего по времени пути для информационного пакета в компьютерной сети.

Особенность – реализация функций скрещивания и мутации.

Скрещивание реализовано путем случайного выбора части генов из одной хромосомы, части из другой. Обычно берется левая половина одной хромосомы и правая половина другой хромосомы.

Мутация реализована путем перемешивания случайной части генов в потомке. Обычно мутация представляет собой добавление к генам случайного числа в интервале (-1, 1).

Исходный код программы

**import** **numpy**

**def** cal\_pop\_fitness(equation\_inputs, pop, sender, receiver):

*# Calculating the fitness value of each solution in the current population.*

*# The fitness function calulates the sum of products between each input and its corresponding weight.*

*#fitness = numpy.sum(pop\*equation\_inputs, axis=1)*

fitness = []

**for** el **in** pop:

res = 0.

res += 1000/equation\_inputs[sender][el[0]]

**for** i **in** range(len(el)-1):

res += 1000/equation\_inputs[el[i]][el[i+1]]

res += 1000/equation\_inputs[el[-1]][receiver]

fitness.append(res)

**return** numpy.array(fitness)

**def** select\_mating\_pool(pop, fitness, num\_parents):

*# Selecting the best individuals in the current generation as parents for producing the offspring of the next generation.*

pop = numpy.array(pop)

parents = numpy.empty((num\_parents, len(pop[0])))

**for** parent\_num **in** range(num\_parents):

max\_fitness\_idx = numpy.where(fitness == numpy.min(fitness))

max\_fitness\_idx = max\_fitness\_idx[0][0]

parents[parent\_num, :] = pop[max\_fitness\_idx, :]

fitness[max\_fitness\_idx] = 99999999999

**return** parents

**def** crossover(parents, offspring\_size):

offspring = numpy.empty(offspring\_size)

*# The point at which crossover takes place between two parents. Usually, it is at the center.*

**for** k **in** range(offspring\_size[0]):

crossover\_points = numpy.random.randint(2, size=offspring\_size[1])

crossover\_first = numpy.where(crossover\_points == 0)

crossover\_second = numpy.where(crossover\_points == 1)

*# Index of the first parent to mate.*

parent1\_idx = k%**parents**.shape[0]

*# Index of the second parent to mate.*

parent2\_idx = (k+1)%**parents**.shape[0]

*# The new offspring will have its first half of its genes taken from the first parent.*

offspring[k, crossover\_first] = parents[parent1\_idx, crossover\_first]

*# The new offspring will have its second half of its genes taken from the second parent.*

offspring[k, crossover\_second] = parents[parent2\_idx, crossover\_second]

**return** offspring

**def** mutation(offspring\_crossover, num\_mutations=1):

print("Mutation in progress")

*# Mutation changes a number of genes as defined by the num\_mutations argument. The changes are random.*

**for** idx **in** range(offspring\_crossover.shape[0]):

crossover\_points = numpy.random.randint(2, size=offspring\_crossover.shape[1])

crossover\_points = numpy.where(crossover\_points == 1)

vals = offspring\_crossover[idx, crossover\_points]

random.shuffle(vals[0])

offspring\_crossover[idx, crossover\_points] = vals[0, :]

**return** offspring\_crossover

**def** input\_weight(num\_comps):

equation\_inputs = numpy.empty((num\_comps,num\_comps))

**for** i **in** range(num\_comps):

**for** j **in** range(i, num\_comps):

**if** i == j:

equation\_inputs[i][j] = 9999999

**continue**

equation\_inputs[i][j] = equation\_inputs[j][i] = numpy.random.uniform(0.000001, 1000, 1)

print(equation\_inputs)

**return** equation\_inputs

**import** **random**

**def** create\_population(sender, receiver, num\_comps, pop\_size):

num\_hubs = [i **for** i **in** range(0, num\_comps) **if** i!=sender **and** i!=receiver]

len\_hubs = len(num\_hubs)

population = []

**for** i **in** range(pop\_size):

elem = [num\_hubs[numpy.random.randint(len\_hubs)] **for** j **in** range(len\_hubs)]

population.append(elem)

**return** numpy.array(population)

num\_parents\_mating = 4

sender = 0

receiver = 9

pop\_size = 10

num\_comps = 10

inputs = [[9999999, 1, 2, 3, 4, 5, 6, 7, 8, 9],

[1, 9999999, 2, 3, 4, 5, 6, 7, 8, 9],

[2, 1, 9999999, 3, 4, 5, 6, 7, 8, 9],

[3, 1, 2, 9999999, 4, 5, 6, 7, 8, 9],

[4, 1, 2, 3, 9999999, 5, 6, 7, 8, 9],

[5, 1, 2, 3, 4, 9999999, 6, 7, 8, 9],

[6, 1, 2, 3, 4, 5, 9999999, 7, 8, 9],

[7, 1, 2, 3, 4, 5, 6, 9999999, 8, 9],

[8, 1, 2, 3, 4, 5, 6, 7, 9999999, 9],

[9, 1, 2, 3, 4, 5, 6, 7, 8, 9999999]]

generated\_inputs = input\_weight(num\_comps)

print('Generated weights: ', generated\_inputs)

equation\_inputs = inputs

# Defining the population size.

# The population will have sol\_per\_pop chromosome where each chromosome has num\_weights genes.

#Creating the initial population.

new\_population = create\_population(sender, receiver, num\_comps, pop\_size)

print(new\_population)

best\_outputs = []

num\_generations = 30

for generation in range(num\_generations):

print("Generation : ", generation)

# Measuring the fitness of each chromosome in the population.

fitness = cal\_pop\_fitness(equation\_inputs, new\_population, sender, receiver)

print("Fitness")

print(fitness)

best\_outputs.append(numpy.min(fitness))

# The best result in the current iteration.

print("Best result : ", numpy.min(fitness))

# Selecting the best parents in the population for mating.

parents = select\_mating\_pool(new\_population, fitness, num\_parents\_mating)

print("Parents")

print(parents)

# Generating next generation using crossover.

offspring\_crossover = crossover(parents, offspring\_size=(pop\_size - parents.shape[0], parents.shape[1]))

print("Crossover")

print(offspring\_crossover)

# Adding some variations to the offspring using mutation.

offspring\_mutation = mutation(offspring\_crossover, num\_mutations=2)

print("Mutation")

print(offspring\_mutation)

# Creating the new population based on the parents and offspring.

new\_population[0:parents.shape[0], :] = parents

new\_population[parents.shape[0]:, :] = offspring\_mutation

# Getting the best solution after iterating finishing all generations.

#At first, the fitness is calculated for each solution in the final generation.

fitness = cal\_pop\_fitness(equation\_inputs, new\_population, sender, receiver)

# Then return the index of that solution corresponding to the best fitness.

best\_match\_idx = numpy.where(fitness == numpy.min(fitness))

print("Best solution : ", new\_population[best\_match\_idx, :])

print("Best solution fitness : ", fitness[best\_match\_idx])

import matplotlib.pyplot

matplotlib.pyplot.plot(best\_outputs)

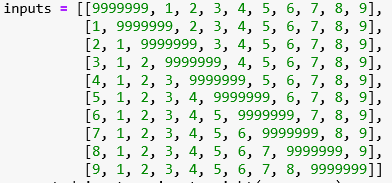
matplotlib.pyplot.xlabel("Iteration")

matplotlib.pyplot.ylabel("Fitness")

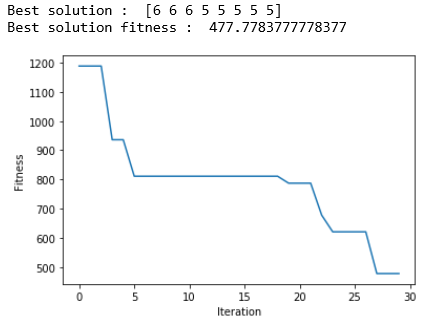
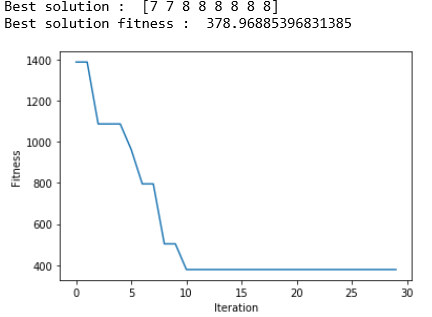
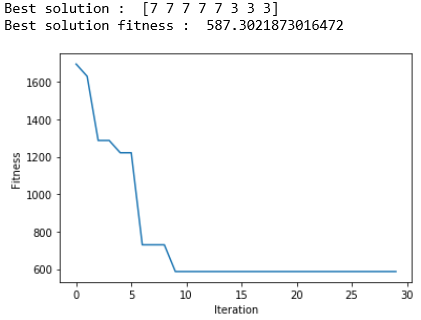
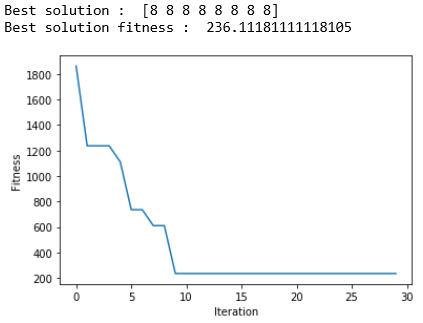
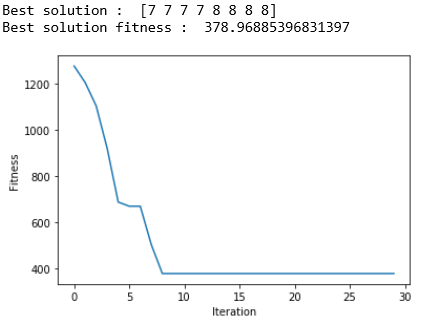
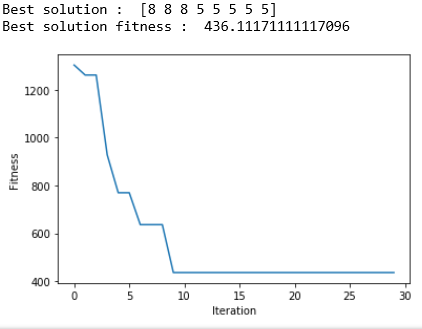
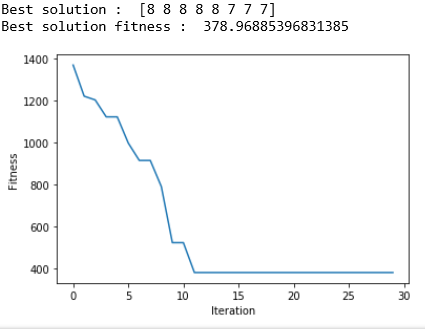
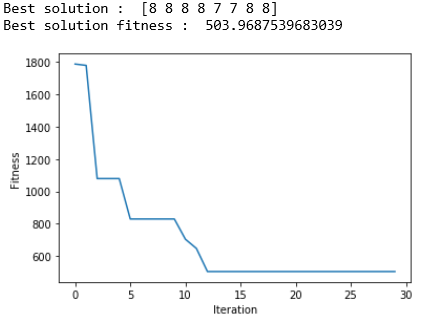
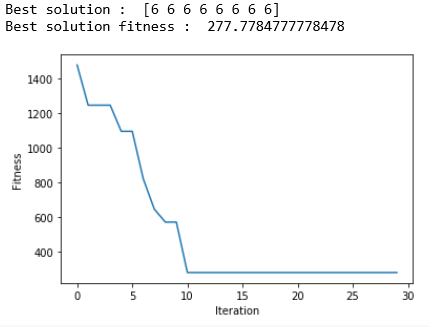
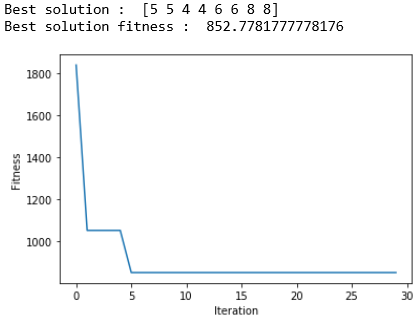
matplotlib.pyplot.show()

1. Протоколы проведенных экспериментов

Входные значения (пропускная способность)



Количество поколений = 30

1. 
2. 
3. 
4. 
5. 
6. 
7. 
8. 
9. 
10. 
11. Выводы

В результате выполнения данной ЛР был реализован генетический алгоритм поиска кратчайшего пути для информационного пакета (сообщения) в компьютерной сети между заданными отправителем и получателем. Трудности возникли в выборе алгоритма скрещивания и мутации, так как при работе обычных версий этих алгоритмов функция приспособленности не доходила до своего минимума, а останавливалась около 0,7. Введение новых алгоритмов мутации и скрещивания позволило решить это проблему.

1. Используемая литература.
2. В.И. Терехов Конспект лекций по курсу МППР
3. <https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%B5%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC>