Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н. Э. Баумана»

(МГТУ им. Н.Э. Баумана)



ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ЛАБОРАТОРНАЯ РАБОТА №7

Отчет о лабораторной работе на тему:

«Муравьиный алгоритм»

Мирзоян Сергей

ИУ7-56

Москва 2019

**Оглавление**

Введение

1. Аналитическая часть 3

1.1 Постановка задачи 3

1.2 Описание алгоритма 3

2. Конструкторская часть 4

2.1 Разработка алгоритма 4

3. Технологическая часть 6

3.1 Требования к программному обеспечению 6

3.2 Средства реализации 6

3.3 Листинг кода 7

4. Экспериментальная часть 10

4.1 Постановка эксперимента 10

Заключение 10

**Введение**

Одной из самых известных задач комбинаторной оптимизации является задача коммивояжера. В классической постановке коммивояжеру нужно совершить тур, посетив каждый город ровно по одному разу и завершив путешествие в том же городе, из которого он выехал. Существует несколько методов решения этой задачи, одним из которых является муравьиный алгоритм.

Цель работы: изучить и реализовать муравьиный алгоритм.

Задачи:

1) реализовать муравьиный алгоритм;

2) реализовать ПО, включающее данный алгоритм, которое будет принимать на вход параметры алгоритма;

3) провести исследования зависимости получаемого результата от входных параметров;

4) описать и обосновать полученные результаты в отчете о выполненной лабораторной работе.

1. **Аналитическая часть**

В данном разделе приведено теоретическое описание реализованного алгоритма.

1.1 Постановка задачи

Реализовать муравьиный алгоритм для решения задачи коммивояжера.

1.2 Описание алгоритма

Муравьиные алгоритмы представляют собой вероятностную жадную эвристику, где вероятности устанавливаются, исходя из информации о качестве решения, полученной из предыдущих решений.

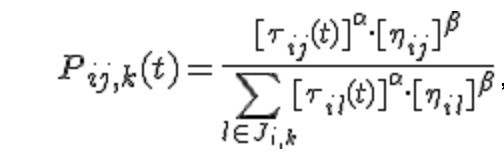
Идея муравьиного алгоритма - моделирование поведения муравьёв, связанного с их способностью быстро находить кратчайший путь от муравейника к источнику пищи и адаптироваться к изменяющимся условиям, находя новый кратчайший путь. При своём движении муравей метит путь феромоном, и эта информация используется другими муравьями для выбора пути. Это элементарное правило поведения и определяет способность муравьёв находить новый путь, если старый оказывается недоступным.

Любой муравьиный алгоритм, независимо от модификаций, представим в следующем виде

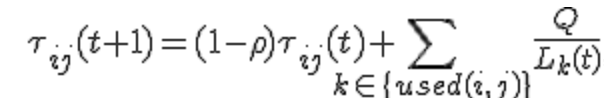
1. Пока (условия выхода не выполнены)

1. Создаём муравьёв
2. Ищем решения
3. Обновляем феромон
4. Дополнительные действия (опционально)

Теперь рассмотрим каждый шаг в цикле более подробно

1. Создаём муравьёв
   * Стартовая точка, куда помещается муравей, зависит от ограничений, накладываемых условиями задачи. Потому что для каждой задачи способ размещение муравьёв является определяющим. Либо все они помещаются в одну точку, либо в разные с повторениями, либо без повторений.
   * На этом же этапе задаётся начальный уровень феромона. Он инициализируется небольшим положительным числом для того, чтобы на начальном шаге вероятности перехода в следующую вершину не были нулевыми.
2. Ищем решения
   * Вероятность перехода из вершины i в вершину j определяется по следующей формуле
   * где τ ij(t) – уровень феромона, ηij – эвристическое расстояние, α, β – константные параметры. Необходим компромисс между этими величинами (чтоб смягчать жадность алгоритма и не застревать в локальных минимумах), который находится экспериментально.

3. Обновляем феромон

* + Уровень феромона обновляется в соответствии с приведённой формулой

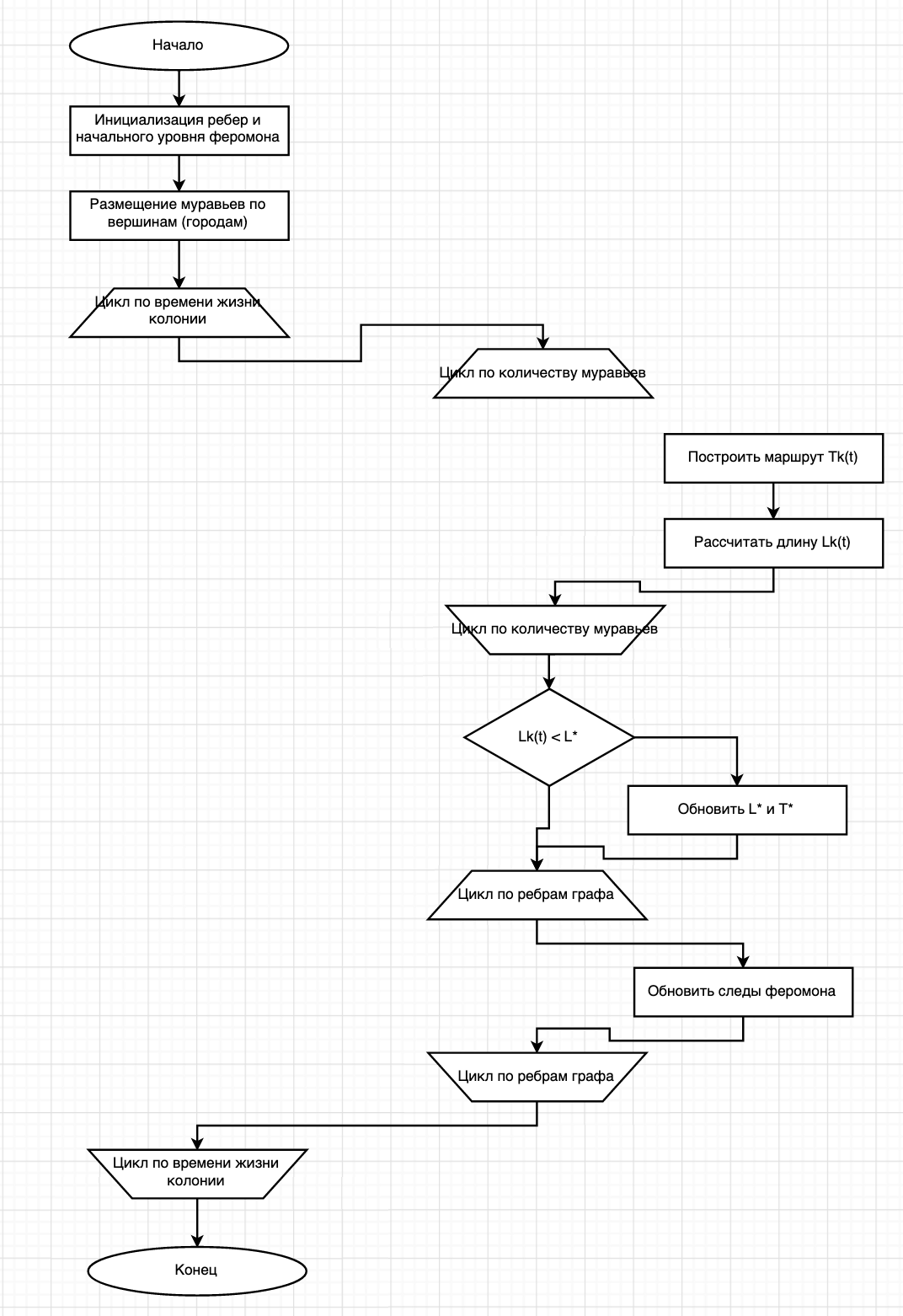
* где ρ – интенсивность испарения, Lk(t) – цена текущего решения для k-ого муравья, а Q – параметр, имеющий значение порядка цены оптимального решения, то есть Q/Lk(t) - феромон, откладываемый k-ым муравьём, использующим ребро (i,j).

4. Дополнительные действия

* Обычно здесь используется алгоритм локального поиска, однако он может также появиться и после поиска всех решений.

1. **Конструкторская часть**

В данном разделе приведена схема реализуемого в лабораторной работе алгоритма.

2.1 Разработка алгоритма

1. **Технологическая часть**

В данном разделе представлены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач, а также листинг кода программы.

3.1 Требования к программному обеспечению

Программа должна работать на всех устройствах, поддерживающих python версии > 3.

3.2 Средства реализации

Программа была реализована на языке Python под macOs. Python был выбран благодаря своей простоте, а также благодаря тому, что данный язык является крайне популярным и существует довольно много материалов, касающихся данной лабораторной работы, реализованных на данном языке.

3.3 Листинг кода

Ниже приведен листинг кода для муравьиного алгоритма

Листинг 1. Код муравья:

def \_\_init\_\_(self, tourSize):

self.trailSize = tourSize

self.trail = [0 for \_ in range(tourSize)]

self.visitedCities = [False for \_ in range(tourSize)]

def visitCity(self, currentIndex, city):

self.trail[currentIndex + 1] = city

self.visitedCities[city] = True

def isVisited(self, i):

return self.visitedCities[i]

def trailLength(self, graph):

length = graph[self.trail[self.trailSize - 1]][self.trail[0]]

for i in range(self.trailSize - 1):

length += graph[self.trail[i]][self.trail[i + 1]]

return length

def clear(self):

for i in range(self.trailSize):

self.visitedCities[i] = False

Листинг 2. Код колонии:

def \_\_init\_\_(self, alpha, beta, evaporation,

Q, graph, ants):

self.c = 1.0

self.alpha = alpha

self.beta = beta

self.evaporation = evaporation

self.Q = Q

self.antFactor = 1

self.randomFactor = 0.01

self.maxIterations = 1000

self.graph = graph

self.numberOfCities = len(graph)

self.numberOfAnts = int(float(self.numberOfCities) \* self.antFactor)

self.currentIndex = 0

self.bestTourOrder = []

self.bestTourLength = 0.0

self.ants = ants

self.trails = [[0.001 for i in range(self.numberOfCities)] for i in range(self.numberOfCities)]

self.probabilities = [0.0 for i in range(self.numberOfCities)]

def solveACO(self):

self.setupAnts()

self.clearTrails()

for i in range(self.maxIterations):

self.currentIndex = 0

for ant in ants:

ant.clear()

ant.visitCity(-1, ant.trail[0])

self.moveAnts()

self.updateTrails()

self.updateBest()

return [] + self.bestTourOrder, self.bestTourLength - float(self.numberOfCities)

def setupAnts(self):

for \_ in range(self.numberOfAnts):

for i in range(len(self.ants)):

self.ants[i].clear()

self.ants[i].visitCity(-1, random.randint(0, self.numberOfCities - 1))

self.currentIndex = 0

def moveAnts(self):

for \_ in range(self.currentIndex, self.numberOfCities - 1):

for i in range(len(ants)):

self.ants[i].visitCity(self.currentIndex, self.selectNextCity(self.ants[i]))

self.currentIndex += 1

def selectNextCity(self, ant):

self.calculateProbabilities(ant)

r = random.random() \* sum(self.probabilities)

total = 0.0

for i in range(self.numberOfCities):

total += self.probabilities[i]

if total >= r:

return i

print(self.probabilities)

print("No cities left\n")

def calculateProbabilities(self, ant):

i = ant.trail[self.currentIndex]

sum = 0

for l in range(self.numberOfCities):

sum += math.pow(self.trails[i][l], self.alpha) / math.pow(self.graph[i][l], self.beta)

for j in range(self.numberOfCities):

if ant.isVisited(j):

self.probabilities[j] = 0

else:

self.probabilities[j] = math.pow(self.trails[i][j], alpha) / math.pow(self.graph[i][j], self.beta) / sum

def updateTrails(self):

for i in range(self.numberOfCities):

for j in range(self.numberOfCities):

self.trails[i][j] \*= 1 - self.evaporation

for i in range(len(self.ants)):

contribution = self.Q / self.ants[i].trailLength(self.graph)

for j in range(self.numberOfCities - 1):

self.trails[self.ants[i].trail[j]][self.ants[i].trail[j + 1]] += contribution

self.trails[self.ants[i].trail[self.numberOfCities - 1]][self.ants[i].trail[0]] += contribution

def updateBest(self):

if not len(self.bestTourOrder):

for i in range(self.numberOfCities):

self.bestTourOrder.append(self.ants[0].trail[i])

self.bestTourLength = self.ants[0].trailLength(self.graph)

for i in range(len(self.ants)):

if self.ants[i].trailLength(self.graph) < self.bestTourLength:

self.bestTourLength = self.ants[i].trailLength(self.graph)

for j in range(self.numberOfCities):

self.bestTourOrder[j] = self.ants[i].trail[j]

def clearTrails(self):

for i in range(self.numberOfCities):

for j in range(self.numberOfCities):

self.trails[i][j] = self.c

1. **Экспериментальная часть**

В экспериментальном разделе представлены примеры работы разработанного программного обеспечения.

4.1. Примеры работы

input alpha: 0.8

input beta: 1

input evaporation: 0.1

input Q: 100

([4, 3, 0, 9, 1, 6, 2, 5, 7, 8], 89.0)

**Заключение**

Во время разработки программного обеспечения в соответствии с поставленными требованиями были получены практические навыки реализации муравьиного алгоритма на языке Python.