**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Факультет прикладной математики – процессов управления

Лабораторная работа по курсу

«Алгоритмы и анализ сложности»

на тему «Эмпирический анализ алгоритма»

Агеев Андрей

18.Б12-пу группа ФИИТ

Содержание

1.Описание алгоритма Левита.

2.Математический анализ алгоритма.

3.Описание характеристик входных данных.

4.Описание способа генерации входных данных.

5.Программа, реализующая алгоритм Левита.

6.Вычислительный эксперимент.

7.Анализ полученных результатов.

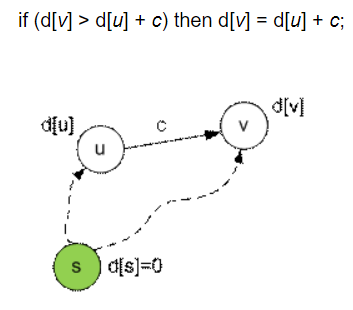
**1.Описание алгоритма**

Алгоритм Левита поиска кратчайшего расстояния в графе - алгоритм на графах, находит кратчайшее расстояние от одной из вершин графа до всех остальных. Алгоритм также работает для графов с ребрами отрицательного веса. Алгоритм Левита проигрывает алгоритму Дейкстры в том, что некоторые вершины приходится обрабатывать повторно, но выигрывает на более простых алгоритмах включения и исключения вершин и также на том, что он применим в случае отрицательных длин дуг. На практике для геометрических графов (графов, построенных на реальных расстояниях) метод Левита является наиболее быстрым.

**Описание работы алгоритма:**

Перед описанием алгоритма правильно будет ввести понятие релаксации ребра.

***Релаксацией*** ребра (*u*, *v*) называется уменьшение значения d[*v*] до d[*u*] + *c* (если второе значение меньше первого).Пример можно увидеть на рис.1.



*Рис.1.* Пример релаксации ребра

**Введем обозначения:**

N - число вершин графа

М - число ребер графа

E - множество ребер графа

wuv - вес ребра uv

s - заданная вершина

d - массив текущих длин кратчайших путей до вершин. d[i] - текущая длина кратчайшего пути до вершины i. Изначально, все элементы d, кроме s-го равны бесконечности; d[s] = 0.

М0 - вершины, расстояние до которых уже вычислено (возможно, не окончательно),

M1 — вершины, расстояние до которых вычисляется. Это множество в свою очередь делится на две [очереди](https://neerc.ifmo.ru/wiki/index.php?title=%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C):

* М’1 - основная очередь;
* М”1 - срочная очередь;

М2 - вершины, расстояние до которых не вычислено.

Изначально все вершины, кроме s помещаются в множество M2. Вершина s помещается в множество М1(в любую из очередей).

Шаг алгоритма:

1. Выбирается вершина u из М1. Если очередь М”1 не пуста, то вершина берется из нее, иначе из М’1. Для каждого ребра uv ∈ E возможны три случая:

* v ∈ М2 , то v переводится в конец очереди М’1 . При этом происходит релаксация ребра: d[v] = d[u] + wuv.
* v ∈ M1. то происходит релаксация ребра uv: d[v] = min(d[v],d[u]+wuv)
* v ∈ М0. Если при этом d[v] > d[u]+wuv, то происходит релаксация ребра d[v] = d[u]+wuv  и вершина v помещается в М”1, иначе ничего не происходит.

Алгоритм завершает работу когда множество M1 становится пустым.

**2.Математический анализ алгоритма.**

При неправильной реализации алгоритма, используя М’1 и М”1 не очередь, а дек(очередь с двумя концами) и добавляя вершины из М0 в начало дека, алгоритм в худшем случае будет работать за экспоненциальное время.

В худшем случае алгоритм Левита работает за O(N2\*M). Однако, на реальных графах алгоритм Левита работает быстрее. Среднее время работы алгоритма O(N\*M). В лучшем случае алгоритм работает за O(N\*logM).

**3.Описание характеристик входных данных**.

Исходными данными для алгоритма являются граф и заданная вершина для который будет происходить поиск кратчайших вершин до всех остальных вершин. Проверять будем на графах с числом вершин N от 10 до 100 с шагом 10 и числе ребер M от 15 до 150 с шагом 10.

**4.Описание способа генерации входных данных.**

Генерируем граф с заданным числом вершин num\_of\_nodes и ребер num\_of\_edge.

def graph\_generator(num\_of\_nodes = 5,num\_of\_edge = 6):

if num\_of\_edge < num\_of\_nodes:

return Exception("Graph is not connected")

else:

nodes = []

# генерируем названия узлов

nodes = list(range(0,num\_of\_nodes))

iter = 0

# создаем пустой граф

graph = DiGraph()

graph.add\_nodes\_from(nodes)

while iter < num\_of\_edge:

start = random.choice(nodes)

end = random.choice(nodes)

if ((start,end) in graph) or (start == end):

continue

else:

weight = random.choice(range(1,10))

graph.add\_edge(start,end)

graph[start][end]['weight'] = weight

iter = iter + 1

return graph

**5.Программный код, реализующий алгоритм Левита.**

Алгоритм реализован на языке программирования Python.

from networkx.algorithms.smallworld import random\_reference

import numpy as np

from networkx import DiGraph

import random

from queue import Queue

def graph\_generator(num\_of\_nodes = 5,num\_of\_edge = 6):

if num\_of\_edge < num\_of\_nodes:

return Exception("Graph is not connected")

else:

nodes = []

# генерируем названия узлов

nodes = list(range(0,num\_of\_nodes))

iter = 0

# создаем пустой граф

graph = DiGraph()

graph.add\_nodes\_from(nodes)

while iter < num\_of\_edge:

start = random.choice(nodes)

end = random.choice(nodes)

if ((start,end) in graph) or (start == end):

continue

else:

weight = random.choice(range(1,10))

graph.add\_edge(start,end)

graph[start][end]['weight'] = weight

iter = iter + 1

return graph

def Levit(graph,s = 0):

# инициализируем нужные нам множества и очереди

M = set()

M1\_main = Queue()

M1\_urgent = Queue()

M1 = set()

M2 = set()

# помещаем заданную вершину в срочную очередь

M1\_urgent.put(s)

M1.add(s)

# помещаем все остальные вершины в множество M2

for node in graph.nodes:

if node != s:

M2.add(node)

# создаем список кратчайших путей от заданной вершины до остальных

d = [float('inf')] \* (len(graph.nodes))

d[s] = 0

list\_of\_edges = list(graph.edges)

#while M1\_main.empty() or M1\_urgent.empty():

while len(M1) != 0:

if not M1\_urgent.empty():

u = M1\_urgent.get()

M1.remove(u)

else:

u = M1\_main.get()

M1.remove(u)

# Получаем вершины, соединенные с текущим ребром

le = list(filter(lambda x: x[0] == u,list\_of\_edges))

if not le:

M.add(u)

if len(M2) != 0:

next\_node = list(M2)[0]

M1\_main.put(next\_node)

M1.add(next\_node)

M2.remove(next\_node)

else:

break

k = []

for i in range(len(le)):

k.append(le[i][1])

for i in range(len(le)):

for v in k:

if v in M2:

M2.remove(v)

M1\_main.put(v)

M1.add(v)

d[v] = d[u] + graph[u][v]['weight']

elif v in M1:

d[v] = min(d[v],d[u] + graph[u][v]['weight'])

elif v in M:

if d[v] > d[u] + graph[u][v]['weight']:

d[v] = d[u] + graph[u][v]['weight']

M1\_urgent.put(v)

M1.add(v)

M.remove(v)

M.add(u)

for i in range(len(d)):

if d[i] == float('inf'):

d[i] = 0

return d

if \_\_name\_\_ == "\_\_main\_\_":

g = graph\_generator()

print('Edges:',g.edges)

p = Levit(g,0)

print("Result sizes of ways:",p)

print("Weights of edges:")

for i in g.edges:

print(i,g[i[0]][i[1]]['weight'])

**6.Вычислительный эксперимент.**

Для наглядности эксперимента диапазон данных возьмем число вершин от 100 до 1000 и число ребер от 200 до 2000. Результат вычислительного эксперимента представлен на Табл.1, где N - число вершин, M - число ребер.

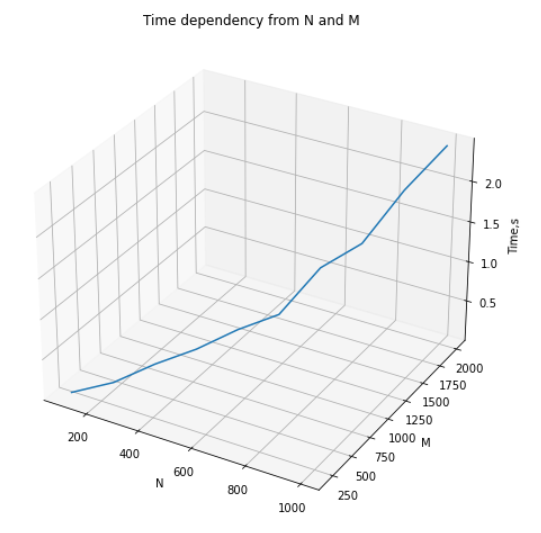
|  |  |
| --- | --- |
| Входные данные(N и M) | Время работы в секундах |
| N=100, M=200 | 0.03293275833129883 |
| N=200, M=400 | 0.13595271110534668 |
| N=300, M=600 | 0.27005767822265625 |
| N=400, M=800 | 0.5060627460479736 |
| N=500, M=1000 | 0.8429901599884033 |
| N=600, M=1200 | 1.4462835788726807 |
| N=700, M=14000 | 1.8170373439788818 |
| N=800, M=1600 | 1.8126604557037354 |
| N=900, M=1800 | 2.7149317264556885 |
| N=1000, M=2000 | 2.946183919906616 |

*Табл.1*. Результаты вычислительного эксперимента.

**7.Анализ полученных результатов.**

|  |  |  |
| --- | --- | --- |
| Входные данные | Время работы в секундах | Отношение времени работы T(2n) к T(n) |
| N=100, M=200 | 0.03293275833129883 | 4,130 |
| N=200, M=400 | 0.13595271110534668 | 3,722 |
| N=300, M=600 | 0.27005767822265625 | 5,355 |
| N=400, M=800 | 0.5060627460479736 | 3,582 |
| N=500, M=1000 | 0.8429901599884033 | 3,495 |

*Табл.2* Результаты работы алгоритма с двойным объемом данных



*Рис.2*. Зависимость времени от N-числа вершин и M-числа ребер

По результатам Табл.1 и Рис.2 можно заметить, что асимптотика этого алгоритма квадратичная, что подтверждает теоретическое предположение о его асимптотике O(N\*M). Также рассмотрим отношение измеренной трудоемкости при удвоении размера входных данных. Теоретически эта асимптотика должна стремиться к 4, т.к. сложность нашего алгоритма квадратичная. Из Табл. 2 можно заметить, что это отношение стремится к 4, подтверждая теоретическое предположение о асимптотике алгоритма.

**Характеристики вычислительной среды:**

Программа запускалась на машине с ОС Windows 10 и процессором Intel i7 1.80 GHz, использовалась VS Code от Microsoft и Jupyter Notebook. Объем ОЗУ: 16 ГБ.

**Список литературы:**

1. Левитин А. Алгоритмы: введение в разработку и анализ
2. Романовский, дискретная математика
3. [https://ru.wikipedia.org/wiki/Алгоритм\_Левита](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9B%D0%B5%D0%B2%D0%B8%D1%82%D0%B0)
4. [https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм\_Левита](https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9B%D0%B5%D0%B2%D0%B8%D1%82%D0%B0)
5. <https://studwood.ru/571185/informatika/analiz_suschestvuyuschih_algoritmov_poiska_kratchayshego_puti_vybor_optimalnogo_algoritma>
6. Доказательство сложности в худшем случае представлено в следующей статье: [https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм\_Левита#.D0.A1.D0.BB.D0.BE.D0.B6.D0.BD.D0.BE.D1.81.D1.82.D1.8C](https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9B%D0%B5%D0%B2%D0%B8%D1%82%D0%B0#.D0.A1.D0.BB.D0.BE.D0.B6.D0.BD.D0.BE.D1.81.D1.82.D1.8C)