Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования

Санкт-Петербургский государственный университет
Факультет прикладной математики и процессов управления

Лабораторная работа

По дисциплине: Алгоритмы и анализ сложности

На тему: Эмпирический анализ алгоритма Левита поиска кратчайшего расстояния в графе

Выполнил:

Студент 3 курса группы 18.Б13-пу

бакалаврской программы

“Программирование и информационные технологии”

Жирнов Василий Владиславович

Санкт-Петербург

2020 г.

Оглавление

Описание алгоритма

Алгоритм Левита - алгоритм решения задачи о нахождении кратчайшего пути в графе. Как и другие алгоритмы из этой семьи, он широко применяется на практике во многих сферах - навигации, сетевой маршрутизации и других задачах, предметную область которых можно изобразить в виде взвешенного графа.

Основная цель алгоритма - снизить среднее время решения задачи, при этом сохраняя возможность работать с взвешенными графами, содержащими ребра отрицательного веса (но не содержащие отрицательных циклов).

Несмотря на то, что оценка алгоритма Левита в худшем случае больше, чем оценка других альтернатив этому алгоритму (например, алгоритм Беллмана-Форда), на практике алгоритм Левита часто оказывается значительно эффективнее. Цель данной работы - показать это эмпирическим путем.

Математический анализ алгоритма

Начнем с краткого описания принципа работы алгоритма для графа G с n вершинами, если начальная вершина i [1, 2]:

1. Создадим массив d длиной n, d[i] = 0, d[0, …, i-1, i+1, ...] = inf
2. Разделим вершины графа на три множества:
   a. M_0 - вершины, расстояние до которых уже вычислено;
   b. M_1 - вершины, расстояние до которых все еще вычисляется. Это множество содержит два подмножества:
      i. M_1' - основная очередь;
      ii. M_1'' - срочная очередь;
   c. M_2 - вершины, расстояние до которых еще не вычислено;
3. Поместим все вершины кроме i в множество M_2, а вершину i в множество M_1';
4. На каждом шаге алгоритма выберем вершину u из M_1, причем срочная очередь имеет приоритет.
   a. Для каждого ребра (u, v) возможны три случая:
      i. $v \in M_2$ - ставим v в очередь M_1'', производим релаксацию ребра uv;
      ii. $v \in M_1$ - производим релаксацию ребра uv;
      iii. $v \in M_0$ - если $d_v > d_u + W_{uv}$ (не была произведена релаксация), то производим релаксацию и помещаем v в срочную очередь M_1'', иначе продолжаем цикл;
   b. Помещаем вершину u в множество M_0;
5. Если M_1 - пустое - заканчиваем работу.

Сложность алгоритма в худшем случае - $O(n^2m)$ [3]. Но на практике алгоритм Левита обычно выполняется быстрее [2]. Покажем это эмпирическим путем.

Описание характеристик входных данных

Исходные данные для алгоритма - репрезентация взвешенного графа и начальная вершина. Граф может содержать ребра с отрицательными весами, но не может содержать отрицательных циклов. Имплементация алгоритма будет использовать в качестве репрезентации графа матрицу смежности.

- Исследуемый диапазон входных данных:
    - число вершин - n от 10 до 100 с шагом 10;
    - число ребер - $n * (n - 1)$;
- Единица измерения трудоемкости - время выполнения программы в секундах. Измерения будут проводиться с помощью модуля *time* языка программирования Python. Вычисления для каждого набора входных данных будут проводиться 10 раз, а время выполнения будет усредняться.

Способ генерации входных данных

Исходные данные для алгоритма генерации входных данных - количество вершин *vertex_count*, ребер *edge_count* и промежуток значений *threshold*, которые может принимать вес того или иного ребра.

Алгоритм возвращает матрицу смежности графа G с *vertex_count* вершинами и **ровно** *edge_count* случайно сгенерированных ребер с весом в промежутке [1, *threshold*]. По причине того, что ребра и веса генерируются случайно, использование отрицательных весов невозможно, так как алгоритм генерации не может гарантировать отсутствие отрицательных циклов в графе.

Код реализации алгоритма генерации данных на языке программирования Python:

```
from random import choice
import numpy as np

def generate_adjacency_matrix(vertice_amount, edge_amount, weight_threshold):
    adj_matrix = np.zeros((vertice_amount, vertice_amount), dtype=int)

    def generate_edge():
        vertice_1 = choice([i for i in range(vertice_amount)])
        vertice_2 = choice([i for i in range(vertice_amount) if i !=
vertice_1])
        if adj_matrix[vertice_1, vertice_2] != 0:
            generate_edge()
        else:
            weight = np.random.randint(1, weight_threshold)
            adj_matrix[vertice_1, vertice_2] = weight
            adj_matrix[vertice_2, vertice_1] = weight

    for _ in range(edge_amount):
        generate_edge()

    return adj_matrix
```

Имплементация алгоритма

Код реализации алгоритма на языке программирования Python:

```
def sssp_levit(graph, source = 0):
    weights = [math.inf for vertex in graph]
    weights[source] = 0

    def relax_edge(source, dest):
        weight = weights[source] + graph[source, dest]
        if weight < weights[dest]:
            weights[dest] = weight

    # set использует в качестве основы хэш-таблицу
- сложность O(1)
    calculated = set()
    calculating = set()
    to_calculate = set()

    calculating_normally = Queue()
    calculating_urgently = Queue()

    # первичный шаг
    calculating_normally.put(source)
    calculating.add(source)

    for vertex in [vertex for vertex in range(len(graph)) if vertex != source]:
        to_calculate.add(vertex)

    while len(calculating) != 0:
        current_vertex = None

        if not calculating_urgently.empty():
            current_vertex = calculating_urgently.get()
            calculating.remove(current_vertex)
        elif not calculating_normally.empty():
            current_vertex = calculating_normally.get()
            calculating.remove(current_vertex)

        adjacent_vertices = [vertex for vertex in range(len(graph)) if
graph[current_vertex, vertex] != 0]

        for adjacent_vertex in adjacent_vertices:
            if adjacent_vertex in to_calculate:
                relax_edge(current_vertex, adjacent_vertex)
```

```
                calculating_normally.put(adjacent_vertex)
                calculating.add(adjacent_vertex)
                to_calculate.remove(adjacent_vertex)
            elif adjacent_vertex in calculating:
                relax_edge(current_vertex, adjacent_vertex)
            elif adjacent_vertex in calculated and weights[adjacent_vertex] >
weights[current_vertex] + graph[current_vertex, adjacent_vertex]:
                calculating_urgently.put(adjacent_vertex)
                calculating.add(adjacent_vertex)
                relax_edge(current_vertex, adjacent_vertex)

        calculated.add(current_vertex)

    return weights
```

Вычислительный эксперимент

Число вершин (число ребер)	Затраченное время, сек. (g(n))
10 (90)	0.0004794597625732422
20 (380)	0.0007126808166503906
30 (870)	0.001270151138305664
40 (1560)	0.0026010513305664063
50 (2450)	0.0037358760833740234
60 (3540)	0.0048999786376953125
70 (4830)	0.007431125640869141
80 (6320)	0.01393423080444336
90 (8010)	0.018296337127685545
100 (9900)	0.0227569580078125

Рис. 1 График функции трудозатрат g(n)

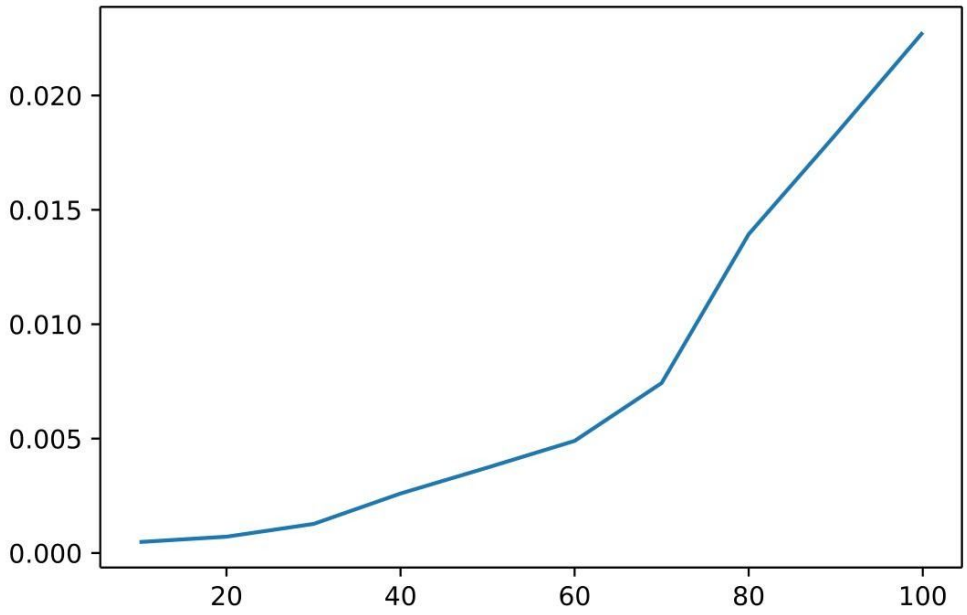


Рис. 2 График функции O(n^2*m) = O(n^2*n*(n-1)) = O(n^4)

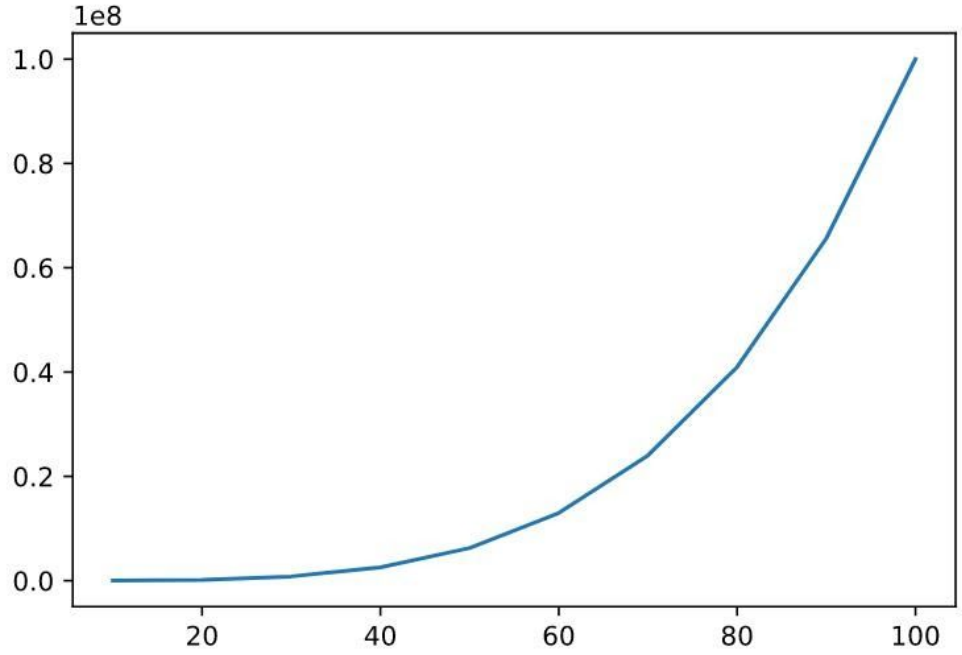
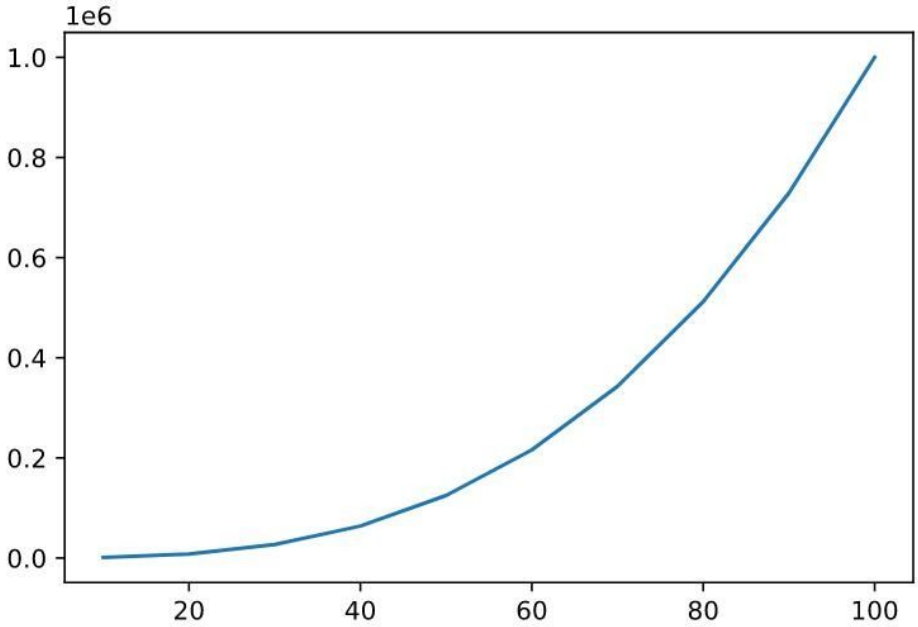


Рис. 3 График функции O(n*m) = O(n * n * (n-1)) = O(n^3)



По рисункам 1, 2, 3 можно увидеть, что g(n) ограничивается O(n^4) сверху и похожа по виду на график n^3. Проверим эту гипотезу, рассмотрев соотношение трудоемкостей при удвоении входных данных:

Число вершин (число ребер)	Время работы в секундах (нормальные входные данные)	Время работы в секундах (удвоенные входные данные)	Соотношение времени работы при удвоенном числе вершин к времени работы при нормальном числе вершин
20 (380)	0.000411748886108398	0.0034899578589041	8.475937583921842
40 (1560)	0.000481075485395850	0.0036032199859619	7.489926415595725
60 (3540)	0.001682615280151367	0.0153910827636718	9.147119335732706
80 (6320)	0.002433204650878906	0.0210222721099853	8.639746805675315
100 (9900)	0.003387403488159179	0.0278057098388671	8.208561494390405
120 (14280)	0.005572032928466797	0.0387843132019043	6.960531945846955
140 (19460)	0.009184455871582032	0.0548385620117187	5.970801403858534
160 (25440)	0.009552955627441406	0.0739532470703125	7.741399620644904
180 (32220)	0.02047724723815918	0.1516987323760986	7.408160413935390
200 (39800)	0.023785781860351563	0.2126335620880127	8.939523759883245

Смотря на соотношение трудоемкостей, можно увидеть, что в большой части случаев оно находится в окрестности 8, но не выходит за 16. Это соответствует предположению о том, что сложность алгоритма близка к кубической. Учитывая, что число ребер m = n * (n - 1), то оценку алгоритма можно записать как O(n^3) = O(nm), что быстрее, чем O(n^2 * m). Гипотеза о сложности алгоритма подтвердилась.

Характеристики среды исполнения

Среда исполнения: Python 3.9.0
Операционная система: Arch Linux x86_64, 5.9.10-arch1-1
Процессор: Intel i7-8565U 8 ядер @ 4.600GHz
Оперативная память: 16 GB

Источники

1. Романовский И.В. - Дискретный анализ: учебное пособие для студентов, специализирующихся по прикладной математике и информатике - 4-е изд., испр. И доп., ББК 22.176.Р69, стр. 223-233
2. http://e-maxx.ru/algo/levit_algorithm
3. https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Левита