Министерство образования и науки Российской Федерации

Новосибирский Государственный Технический Университет

Кафедра ПМТ

Лабораторная работа №3

по дисциплине

«Дискретные математические модели»

Факультет: ПМИ

Группа: ПМИМ-71

Студенты: Кочнев А. В.

Преподаватель: Рояк М. Э.

Новосибирск

2018

**Цель работы:**

Реализация различных вариантов алгоритма Дейкстры и сравнение их временных оценок

**Алгоритм Дейкстры поиска кратчайшего пути в графе:**

Пусть дан взвешенный ориентированный граф ***G(V, E )*** без дуг отрицательного веса. Требуется найти кратчайшие пути от некоторой заданной вершины ***a*** графа ***G*** до всех остальных вершин этого графа.

Каждой вершине из ***V*** поставим в соответствие метку – минимальное известное расстояние от этой вершины до ***a*** . Алгоритм Дейкстры работает пошагово – на каждом шаге он «*посещает*» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены.

**Инициализация:**

Метка самой вершины ***a*** полагается равной 0, метки остальных вершин – бесконечности. Это отражает то, что расстояния от ***a*** до других вершин пока неизвестны. Все вершины графа помечаются как непосещённые.

**Шаг алгоритма:**

Если все вершины посещены, алгоритм завершается. В противном случае из ещё не посещённых вершин выбирается вершина ***u*** , имеющая минимальную метку. Вершины, в которые ведут рёбра из ***u***, назовём соседями этой вершины. Для каждого непосещённого соседа вершины ***u*** рассмотрим новую длину пути, равную сумме значений текущей метки ***u*** и длины ребра, соединяющего ***u*** с этим соседом. Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину u как посещённую и повторим шаг алгоритма

**Реализация с использование кучи:**

Очевидно, что асимптотические свойства алгоритма будут зависеть от

способа хранения множества посещенных вершин и известных уже расстояний. В простейшем случае для решения задачи можно воспользоваться массивом для хранения расстояний, тогда для графа имеющего ***N*** вершин потребуется выполнить на каждом шаге поиск минимального расстояния в ***N*** действий. Следовательно временная оценка алгоритма окажется ***O(N2)***.

В случае использования бинарной кучи, время выполнения поиска и модификации структуры хранящей информации о кратчайших путях можно сократить до ***log(N)***. Тогда общее время выполнения будет порядка ***O(Nlog(N) + Mlog(N))***.

Для реализации второго варианта алгоритма воспользуемся пакетом sortedcollections и структурой реализующей словарь отсортированный по значениям (меткам) ValueSortedDict, с заявленным временем доступа к элементу с минимальным значением ***log(N)*** и модификацией ***log(N)***.

**Генерация графа:**

Для проведения экспериментов воспользуемся случайно сгенерированными графами, с заданной вероятностью наличия ребра между двумя вершинами и случайным весом.

**Проведение эксперимента:**

Эксперимент проведем следующим образом:

1. Зададим некоторое количество повторов работы алгоритма N\_TIMES
2. Сгенерируем граф заданного объема
3. Найдем среднее время выполнения каждого из алгоритмов на сгенерированном графе

**Результаты:**

Зададим N\_TIMES = 10

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Volume | Avg simple | Avg heaped | Faster | Ratio |
| 10 | 0,0000464081764221191 | 0,000172102451324462 | Simple | 3,708451066015920 |
| 20 | 0,0001215696334838860 | 0,000376915931701660 | Simple | 3,100411845459910 |
| 40 | 0,0003103256225585940 | 0,000835919380187988 | Simple | 2,693684695759070 |
| 60 | 0,0005411148071289050 | 0,001503872871398920 | Simple | 2,779212195981660 |
| 120 | 0,0018078804016113200 | 0,003740787506103510 | Simple | 2,069156512106350 |
| 240 | 0,0066171288490295300 | 0,011043500900268500 | Simple | 1,668926380644400 |
| 480 | 0,0281997919082641000 | 0,042726099491119300 | Simple | 1,515121091322600 |
| 960 | 0,1251504063606260000 | 0,162328672409057000 | Simple | 1,297068680235050 |
| 1920 | 0,5936275959014890000 | 0,668072772026062000 | Simple | 1,125407202492870 |
| 4000 | 3,0994163513183500000 | 3,087443208694450000 | Heaped | 0,996136968620300 |
| 6000 | 7,3869280815124400000 | 7,026370203495020000 | Heaped | 0,951189740303577 |
| 8000 | 14,6595667481422000000 | 13,546906304359400000 | Heaped | 0,924100045867740 |

**Исходный код:**

Python3.6

main.py

**from** read\_graph **import** read\_graph  
**from** dejkstra\_simple **import** dejkstra\_simple  
**from** dejkstra **import** dejkstra  
**from** print\_pathes **import** print\_pathes  
**import** time  
  
EXAMPLE\_STR = **'examples/1000\_0.001.txt'**N\_TIMES = 10  
  
input = read\_graph(EXAMPLE\_STR)  
acc\_simple = 0.0  
acc\_heaped = 0.0  
  
i = 0  
**while** i < N\_TIMES:  
 i = i + 1  
 start\_time\_s = time.time()  
 pathes\_s = dejkstra\_simple(input)  
 acc\_simple = acc\_simple + (time.time() - start\_time\_s)  
  
 start\_time = time.time()  
 pathes = dejkstra(input)  
 acc\_heaped = acc\_heaped + (time.time() - start\_time)  
  
print(**'Simple %s'** % (acc\_simple/N\_TIMES))  
print(**'Heaped %s'** % (acc\_heaped/N\_TIMES))  
print(pathes == pathes\_s)

dejkstra\_simple.py

**def** dejkstra\_simple(input):  
 (labels, vertices) = init\_dejkstra(input)  
 **while** bool(vertices):  
 v\_id = get\_min\_label(labels, vertices)  
 **if** (v\_id == -1): *# Если есть недостижимые вершины* **return** labels  
 edges = get\_edges(input, v\_id)  
 labels = calc\_labels(labels, edges, v\_id)  
 vertices = remove\_vertex(vertices, v\_id)  
 **return** labels  
  
**def** remove\_vertex(vertices, v\_id):  
 idx = vertices.index(v\_id)  
 **return** vertices[0:idx] + vertices[idx+1:]  
  
**def** calc\_labels(labels, edges, v\_id):  
 result = labels  
 self\_label = labels[v\_id]  
 **for** edge **in** edges:  
 target\_v = edge[**'id'**]  
 target\_w = edge[**'weight'**]  
 **if** labels[target\_v] > self\_label + target\_w:  
 result[target\_v] = self\_label + target\_w  
 **return** result  
  
**def** get\_edges(input, v\_id):  
 **return** input[**'graph'**][v\_id][**'edges'**]  
  
**def** get\_min\_label(labels, vertices):  
 v\_id = -1  
 v\_val = float(**'inf'**)  
 **for** c\_v\_id **in** vertices:  
 c\_v\_val = labels[c\_v\_id]  
 **if** c\_v\_val < v\_val:  
 v\_val = c\_v\_val  
 v\_id = c\_v\_id  
 **return** v\_id  
  
**def** init\_dejkstra(input):  
 labels = {}  
 vertices = []  
 graph = input[**'graph'**]  
 lead = input[**'lead'**]  
 **for** idx, vertex **in** graph.items():  
 label = 0 **if** idx == lead **else** float(**'inf'**)  
 labels[idx] = label  
 vertices.append(idx)  
 **return** (labels, vertices)

dejkstra.py

**from** sortedcollections **import** ValueSortedDict  
**from** sys **import** maxsize  
**from** random **import** random  
*# 1. Выставляем всем вершинам метки 0 для lead, для остальных inf  
# 2. Выбираем непокрытую вершину с наименьшей меткой  
# 3. Для всех ее соседей пересчитываем метки  
# 4. Как соседей нет считаем вершину покрытой  
# 5. Переходим на 2 если есть непокрытые вершины*  
*# Особенности реализации:  
# Пути хранятся в словаре labels => [:метка]: [:вершина]  
# Словарь с сортировкой*

*# WARNING global!!!*v\_dict = ValueSortedDict()  
  
**def** dejkstra(input):  
 labels = init\_dejkstra(input)  
 **while** bool(v\_dict.\_\_len\_\_()):  
 v\_id = pop\_min\_label()  
 **if** labels[v\_id] == float(**'inf'**): *# Если есть недостижимые вершины* **return** labels  
 edges = get\_edges(input, v\_id)  
 labels = calc\_labels(labels, edges, v\_id)  
 **return** labels  
  
**def** calc\_labels(labels, edges, v\_id):  
 result\_labels = labels  
  
 self\_label = labels[v\_id]  
 **for** edge **in** edges:  
 target\_v = edge[**'id'**]  
 target\_w = edge[**'weight'**]  
 **if** labels[target\_v] > self\_label + target\_w:  
 v\_dict.\_\_delitem\_\_(target\_v)  
 v\_dict.\_\_setitem\_\_(target\_v, self\_label + target\_w)  
 result\_labels[target\_v] = self\_label + target\_w  
  
  
 **return** result\_labels  
  
**def** get\_edges(input, v\_id):  
 **return** input[**'graph'**][v\_id][**'edges'**]  
  
  
**def** pop\_min\_label():  
 (id, \_v) = v\_dict.popitem(0)  
 **return** id  
  
**def** init\_dejkstra(input):  
 labels = {}  
 graph = input[**'graph'**]  
 lead = input[**'lead'**]  
  
 **for** idx, vertex **in** graph.items():  
 label = 0 **if** idx == lead **else** float(maxsize)  
 labels[idx] = label  
  
 v\_dict.update(  
 list(labels.items())  
 )  
 **return** labels

generate\_graph.py

**from** random **import** random  
  
GRAPH\_N = 1000  
LEAD\_VERTEX = 0  
EDGE\_PROB = 0.001

EXAMPLE\_STR = **'examples/%s\_%s.txt'** % (GRAPH\_N, EDGE\_PROB)  
  
spaced = **lambda** char, pos: str(char) **if** pos == 0 **else ' %s'** % char  
roundedX10 = **lambda** val: round(val \* 10)  
out\_file = open(EXAMPLE\_STR, **'w'**)  
out\_file.write(**'%s\n'** % LEAD\_VERTEX)  
  
**for** i **in** range(GRAPH\_N):  
 **for** j **in** range(GRAPH\_N):  
 **if** i == j:  
 out\_file.write(spaced(**'-'**, i))  
 **else**:  
 prob = random()  
 w = roundedX10(random())  
 **if** prob > EDGE\_PROB **and** w > 1:  
 out\_file.write(spaced(w, j))  
 **else**:  
 out\_file.write(spaced(**'-'**, j))  
 out\_file.write(**'\n'**)