# Министерство образования и науки Российской Федерации

# Федеральное государственное бюджетное образовательное учреждение высшего образования «Новосибирский государственный технический университет»



## Кафедра параллельных вычислительных технологий

## Домашнее задание № 4 по дисциплине «Методы проектирования и анализа алгоритмов»

**Алгоритмы на графах**

Вариант 12

|  |  |  |
| --- | --- | --- |
|  | Факультет: | ПМИ |
| Группа: | ПМ-53 |
| Студент: | Тябин Е.А. |
| Преподаватель: | Щукин Г.А. |

Новосибирск

2017

# Условие задачи

Реализовать указанный в задании алгоритм. Проверить правильность работы алгоритма с помощью юнит-тестов. Замерить время работы алгоритма на указанном большом графе.

Для своего алгоритма также найти:

* поиск минимального остовного дерева: вес остовного дерева

## Графы для замера времени

Требуемый тип графа:

* поиск минимального остовного дерева: неориентированный взвешенный граф

Если требуется взвешенный граф, сгенерировать веса для ребер/дуг случайным образом. Для превращения ориентированного графа в неориентированный заменить дуги на ребра (учесть симметрию ребер). Если граф несвязный и требуется связный граф (поиск минимального остовного дерева), сделать граф связным путем добавления дополнительных ребер/дуг.

# -Алгоритм решения

**Алг** search(key){

1)Полагаем список ребер искомого остовного дерева пустым (list=[])

2)Создание списка рёбер edges.

3)Сортировка множества рёбер по их весам в порядке возрастания по полю weight.

4)Формируем множество edge, элементами которого являются множества вершин, соответствующих компонентам исходного остовного леса. Каждая такая компонента состоит из единственной вершины.

5)Если множество edge содержит более одного элемента (т.е. остовный лес состоит из нескольких компонент) и список edges не пуст, переходим дальше, иначе на шаг 8)

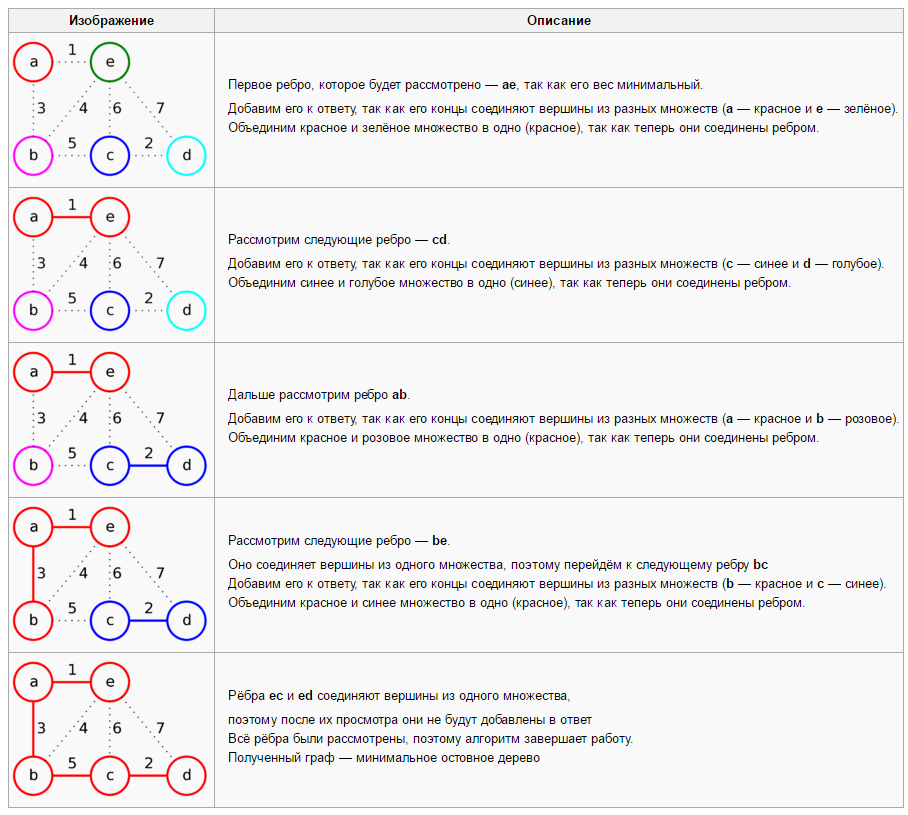
6)Извлекаем из списка edges ребро e. Если концы ребра(вершины) е принадлежат различным множествам вершин из edge, то переходим на следующий шаг, если иначе, то отбрасываем извлеченное ребро и возвращаемся на шаг 5.

7) Объединяем множества данных вершин и добавляем полученное объединение в множество edge. Добавляем ребро e в список list

8) Полученный список list и есть список рёбер минимального остовного дерева, упорядоченный по возрастанию весов.

Граф задается списком ребер, ребро задается тремя числами (начало, конец, вес), ребра хранятся в виде кортежей в списке edges. Вершины нумеруются, начиная с нуля, в списке edge хранится номер компоненты связанности, к которой отнесена данная вершина.

Иллюстрация работы алгоритма:



# Текст программы

**/\* Файл DZtimer.py \*/**

**import** timeit  
**from** graph **import** Graph  
**from** generate **import** main  
**from** util **import** powers\_of  
**def** make\_header(func\_names):  
 **return " "**.join([**"{0:<12}"**.format(s) **for** s **in** [**"N"**]+ func\_names])  
  
  
**def** make\_line(n, times):  
 **return " "**.join([**"{0:<12}"**.format(n)] + [**"{0:<12.5f}"**.format(t) **for** t **in** times])  
  
  
**def** time\_us(ns, generator):  
 generator()  
 print(make\_header([**'Krascal algorithm'**]))  
 G=Graph()  
 **for** n **in** ns:  
 G = G.input\_Graph(**"records\_1e{0}.txt"**.format(n))  
 times = []  
 timer = timeit.Timer(**lambda**: G.krascal\_alg())  
 times.append(timer.timeit(1))  
 print(make\_line(10 \*\* (n - 1), times))  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 time\_us(ns=powers\_of(1, 0, 5), generator=main)

**/\* Файл graph.py \*/**

**import** random  
  
**class** Graph(object):  
 *"""Graph class"""* **def** \_\_init\_\_(self, vertices=[], edges=[], weight={}):  
 self.vertices = set()  
 self.adjacent = {}  
 self.weight = {}  
 **for** v **in** vertices:  
 self.add\_vertex(v)  
 **for** v1, v2 **in** edges:  
 self.add\_edge(v1, v2)  
 **for** k, v **in** weight:  
 self.weight[k] = v  
  
 **def** add\_vertex(self, vertex):  
 *"""Add vertex to the graph."""* self.vertices.add(vertex)  
 **"""Add vertex to the graph."""  
 if** vertex **not in** self.adjacent:  
 self.adjacent[vertex] = []  
  
 **def** add\_edge(self, v1, v2, weight=1):  
 *"""Add edge to the graph between vertices v1 and v2."""* self.add\_vertex(v1)  
 self.add\_vertex(v2)  
 **if** v1>v2:  
 v1,v2=v2,v1  
 **if** v1 **in** self.weight:  
 **if** self.has\_edge(v1,v2):  
 self.weight[v1][v2]+=weight  
 **else**:  
 self.weight[v1][v2]=weight  
 **else**:  
 self.weight[v1] = {}  
 self.weight[v1][v2] = weight  
 **if** v2 **not in** self.weight:  
 self.weight[v2] = {}  
 self.weight[v1][v2] = weight  
 v1\_adj = self.adjacent[v1]  
 v2\_adj = self.adjacent[v2]  
 **if** v2 **not in** v1\_adj:  
 v1\_adj.append(v2)  
  
  
 **def** get\_vertices(self):  
 *"""Get all vertices of the graph."""* **return** list(self.vertices)  
  
 **def** get\_adjacent(self, vertex):  
 *"""Get list of adjacent vertices in the graph for the vertex"""* **if** vertex **in** self.adjacent:  
 **return** self.adjacent[vertex]  
 **return** []  
  
 **def** get\_edge\_weight(self, v1, v2):  
 **return** self.weight[v1][v2]  
  
 **def** get\_weight(self):  
 weight=0  
 **for** i **in** self.get\_vertices():  
 **for** j **in** self.get\_adjacent(i):  
 weight+=self.get\_edge\_weight(i, j)  
 **return** weight  
  
 **def** has\_edge(self, v1, v2):  
 *"""Returns True if there is an edge between vertices v1 and v2, False otherwise."""* **if** v1 **not in** self.vertices **or** v2 **not in** self.vertices:  
 **return False  
 return** v2 **in** self.adjacent[v1]  
  
 **def** input\_Graph(self, file):  
 graph = Graph()  
 **with** open(file) **as** graph\_file:  
 graph\_file = graph\_file.read()  
 **for** edge **in** graph\_file.split(**'\n'**):  
 vert = edge.split(**'\t'**)  
 graph.add\_edge(str(vert[0]), str(vert[1]), random.randint(1, 10))  
 **return** graph  
  
 **def** krascal\_alg(self):  
 list = []  
 edges = []  
 **for** i **in** self.get\_vertices():  
 **for** j **in** self.get\_adjacent(i):  
 edges.append([i, j, self.get\_edge\_weight(i, j)])  
 edges.sort(key=**lambda** x: x[2])  
 edge = {i: i **for** i **in** self.get\_vertices()}  
 **for** start, end, weight **in** edges:  
 **if** edge[start] != edge[end]:  
 list.append((start, end, weight))  
 a = edge[start]  
 b = edge[end]  
 **for** i **in** self.get\_vertices():  
 **if** edge[i] == b:  
 edge[i] = a  
 **return** list

**/\* Файл generate.py \*/**

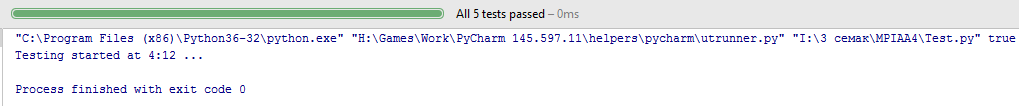
**import** random  
  
  
**def** read\_file(file):  
 **with** open(file, **"r"**) **as** f:  
 **return** f.read().splitlines()  
  
  
**def** write\_file(file, data):  
 **with** open(file, **"w"**) **as** f:  
 f.write(**"\n"**.join(data))  
  
**def** main():  
 records=[]  
 **for** p **in** range(1, 6):  
 num = 10\*\*p  
 output\_file = **"records\_1e{0}.txt"**.format(p)  
 print(**"Generating {0} records into {1}..."**.format(num, output\_file))  
 **for** i **in** range(num):  
 records.append(str(random.randint(1, 10\*\*p)) + **'\t'** + str(random.randint(1, 10\*\*p)))  
 write\_file(output\_file, records)  
 print(**"Done"**)  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

# Юнит-тесты

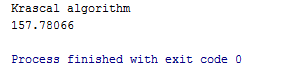
**/\* Файл Test.py \*/**

**from** DZ **import** krascal\_alg  
**from** graph **import** Graph  
**import** unittest  
  
  
**class** GraphTests(unittest.TestCase):  
 **def** setUp(self):  
 self.graph = Graph()  
  
 **def** test\_empty(self):  
 self.assertEqual(self.graph.get\_vertices(), [])  
  
 **def** test\_add\_edge(self):  
 self.graph.add\_edge(**"A"**, **"B"**)  
 self.assertEqual(len(self.graph.get\_vertices()), 2)  
 self.assertTrue(self.graph.has\_edge(**"A"**, **"B"**))  
 self.assertIn(**"B"**, self.graph.get\_adjacent(**"A"**))  
 self.assertEqual(self.graph.krascal\_alg(), [(**'A'**, **'B'**, 1)])  
  
 **def** test\_add\_edges(self):  
 self.graph.add\_edge(**"A"**, **"B"**)  
 self.graph.add\_edge(**"A"**, **"C"**)  
 self.graph.add\_vertex(**"D"**)  
 self.assertEqual(len(self.graph.get\_vertices()), 4)  
 self.assertIn(**"C"**, self.graph.get\_adjacent(**"A"**))  
 self.assertNotIn(**"D"**, self.graph.get\_adjacent(**"A"**))  
 self.assertFalse(self.graph.has\_edge(**"A"**, **"D"**))  
  
 **def** test\_krascal\_alg(self):  
 self.graph.add\_edge(**'0'**, **'1'**, 1)  
 self.graph.add\_edge(**'1'**, **'0'**, 2)  
 self.graph.add\_edge(**'1'**, **'3'**, 8)  
 self.graph.add\_edge(**'1'**, **'2'**, 6)  
 self.graph.add\_edge(**"3"**, **"2"**, 4)  
 self.graph.add\_edge(**"3"**, **"4"**, 5)  
 self.graph.add\_edge(**"5"**, **"6"**, 9)  
 self.graph.add\_edge(**"4"**, **"6"**, 8)  
 self.graph.add\_edge(**"3"**, **"5"**, 2)  
 self.graph.add\_edge(**"8"**, **"4"**, 1)  
 self.graph.add\_edge(**"7"**, **"4"**, 7)  
 self.assertEqual(self.graph.krascal\_alg(), [(**'4'**,**'8'**,1), (**'3'**,**'5'**,2), (**'0'**, **'1'**, 3), (**'2'**,**'3'**,4), (**'3'**,**'4'**,5), (**'1'**, **'2'**, 6), (**'4'**,**'7'**,7), (**'4'**,**'6'**,8)])  
  
 **def** test\_weight(self):  
 self.graph.add\_edge(**'0'**, **'1'**, 1)  
 self.graph.add\_edge(**'1'**, **'0'**, 2)  
 self.graph.add\_edge(**'1'**, **'3'**, 8)  
 self.graph.add\_edge(**'1'**, **'2'**, 6)  
 self.graph.add\_edge(**"3"**, **"2"**, 4)  
 self.graph.add\_edge(**"3"**, **"4"**, 5)  
 self.graph.add\_edge(**"5"**, **"6"**, 9)  
 self.graph.add\_edge(**"4"**, **"6"**, 8)  
 self.graph.add\_edge(**"3"**, **"5"**, 2)  
 self.graph.add\_edge(**"8"**, **"4"**, 1)  
 self.graph.add\_edge(**"7"**, **"4"**, 7)  
 self.assertEqual(self.graph.get\_weight(), 53)  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 unittest.main()

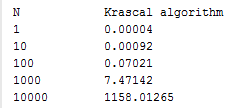
# Результаты замера времени работы алгоритма и тестов

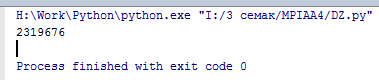


Время нахождения минимального оставного дерева заданного графа:



Сгенерированных графов, с N-ым количеством рёбер:



Вес минимального оставного дерева заданного графа: 

# Выводы

Проделав работу, мы успешно построили алгоритм для решения нашей задачи и реализовали его на языке программирования, после чего были сделаны юнит-тесты для проверки правильности работы алгоритма, а также сделаны замеры времени работы алгоритма. Поскольку общее время работы алгоритма Краскала составляет (где О(MlogN)-количество операций сортировки рёбер-списка edges, O(N)-количество операций объединения для работы алгоритма Краскала), то можно сделать вывод о том, что функция работает за линейно-логарифмическое время.