Teoria dos Grafos

COS242 - 2022.2

TP2

Felippi Blanchard
Rodrigo Lucas

Representação do grafo com pesos

Lista de Adjacência

Representação do TP1 (sem peso):

Complexidade de O(2m)

Antes

Classe que constrói a lista usando AdjNode e Graph

```python
class AdjascencyList:
    def __init__(self, file_name):
        self.graph = Graph(file_name)
        self.list = [None] * self.graph.vertices_quantity
        self.build_representation()

    def build_representation(self):
        for edge in self.graph.edges:
            self.add_edge(edge[0], edge[1])

    def add_edge(self, s, d):
        node = AdjNode(d)
        node.next = self.list[s]
        self.list[s] = node

        node = AdjNode(s)
        node.next = self.list[d]
        self.list[d] = node
```

Objeto da lista de Adjacência

```python
class AdjNode:
    def __init__(self, value):
        self.vertex = value
        self.cost = 0
        self.next = None
```

Classe que constrói um array de arestas a partir da entrada do Grafo

```python
class Graph:
    def __init__(self, file_name):
        self.file_name = file_name
        self.readed_graph = self.read_graph_file()
        self.vertices_quantity = self.vertices_qty() + 1
        self.edges = self.edges_array()

    def read_graph_file(self): ...

    def vertices_qty(self): ...

    def edges_array(self):
        edges = []

        for line in self.readed_graph[1:]:
            line = line.split(' ')
            edges.append([int(line[0]), int(line[1])])
        return edges
```

Representação do grafo com pesos

Lista de Adjacência

Depois

Representando diretamente a partir do arquivo:

Reduzimos em O(m) a complexidade do algoritmo para a representação do Grafo.

Objeto da lista de Adjacência

```
class AdjNode:
    def __init__(self, value):
        self.vertex = value
        self.cost = 0
        self.next = None
```

Classe que constrói a lista usando AdjNode

```
class AdjascencyListV2:
    def __init__(self, file_name, weighted=False):
        self.file_name = file_name
        self.weighted = weighted
        self.haveNegativeWeight = False

        f = open(self.file_name, "r")
        self.vertices_quantity = int(f.readline()) + 1

        self.list = [None] * self.vertices_quantity
        self.build_representation()
```

Representação do grafo com pesos

Lista de Adjacência

Verificação ao representar o grafo em Lista de Adjacência:
tem peso ? tem peso negativo ?

```
def build_representation(self):
    fist_line = True
    with open(self.file_name) as f:
        for line in f:
            if fist_line != True:
                edge = line.strip().split(' ')
                if self.weighted == True:
                    self.add_edge([int(edge[0]), float(edge[2])], [int(edge[1]), float(edge[2])])
                    if(float(edge[2])<0):
                        self.haveNegativeWeight = True
                else:
                    self.add_edge(int(edge[0]), int(edge[1]))

            else:
                fist_line = False
```

Dijkstra
sem heap

Não exige uma verificação se o grafo tem peso negativo, visto que a representação em **lista de adjacência já tem** essa informação

```python
def verify_graph_have_negative_weight(self):
    if(self.graph.have_negative_weight()):
        print("This graph have edges with negative weigth, so, we can't use Dijkstra.")
        return True
    return False
```

Verificação em todos os vértices

```python
def find_vertex_with_minimum_cost(self, vertices, distance):
    vertex_with_minimum_cost = vertices.pop()

    for vertex in vertices:
        if distance[vertex] < distance[vertex_with_minimum_cost]:
            vertex_with_minimum_cost = vertex

    return vertex_with_minimum_cost
```

Complexidade de $O(n^2)$

```python
def iterate(self):
    if(self.verify_graph_have_negative_weight()):
        return
    distance = [float('inf')] * self.graph.vertices_quantity
    V = self.graph.vertices()
    S = set()
    distance[self.s] = 0

    while(S != V):
        remain_vertices = V - S
        u = self.find_vertex_with_minimum_cost(remain_vertices, distance)
        S.add(u)
        for v in self.graph.node_neighbors(u):
            if distance[v[0]] > distance[u] + self.weight(v):
                distance[v[0]] = round( distance[u] + self.weight(v), 3)

    return distance
```

Dijkstra
com heap

Foi utilizado a biblioteca **heapdict** para a criação da fila de prioridade e assim diminuir a complexidade do algoritmo de $O(n^2)$ para $O((m+n) \log n)$

Exemplo de output da heap (vértice, distância):

[(211, 2.12), (880, 2.22), (44, 2.38), (301, 2.16), (226, 2.17)]

Retorna o vértice de menor distância

Possui complexidade O(1) para obter este vértice, graças à estrutura da heap

```python
def iterate_heap(self):
    if(self.verify_graph_have_negative_weight()):
        return
    distance = [float('inf')] * self.graph.vertices_quantity

    distance[self.s] = 0

    priority_queue = heapdict.heapdict()
    priority_queue[self.s] = 0

    V = self.graph.vertices()
    S = set()

    while(S != V):
        u = self.find_vertex_with_minimum_cost_heap(priority_queue)
        S.add(u[0])

        for v in self.graph.node_neighbors(u[0]):
            if distance[v[0]] > distance[u[0]] + self.weight(v):
                priority_queue[v[0]] = round(distance[u[0]] + self.weight(v), 3)
                distance[v[0]] = round(distance[u[0]] + self.weight(v), 3)

    return distance

def find_vertex_with_minimum_cost_heap(self, queue):
    vertex_with_minimum_cost = queue.popitem()

    return vertex_with_minimum_cost
```

MST

Prim com heap

O((m+n) log n)

Criação de array de pais e custos, para ter acesso ao representar a MST em grafo

```python
def generate_MST(self):
    self.cost = [float('inf')] * self.graph.vertices_quantity
    self.cost[0] = 0
    self.S = set()
    priority_queue = heapdict.heapdict()
    priority_queue[self.root] = 0
    self.parents = [0] * self.graph.vertices_quantity
    self.parents[self.root] = -1
    self.cost[self.root] = 0
    while(len(self.S) < self.graph.vertices_quantity-1):
        u = self.find_vertex_with_minimum_cost_heap(priority_queue)
        self.S.add(u[0])
        for v in self.graph.node_neighbors(u[0]):
            if self.cost[v[0]] > self.weight(v):
                self.parents[v[0]] = u[0]
                priority_queue[v[0]] = self.weight(v)
                self.cost[v[0]] = self.weight(v)
```

Representação da MST

com heap

Sem segredos

```
def write_MST(self, file_name):
    f = open(file_name, "w")
    f.write("Custo total: " + str(self.total_cost_MST()) + "\n")
    f.write("Arvore:" + "\n")
    for vertex in self.S:
        if(self.parents[vertex] == -1):
            f.write("Vertice: "+ str(vertex) + "--- Pai: Raiz" + ", Peso: " + str(self.cost[vertex]) + "\n")
        else:
            f.write("Vertice: "+ str(vertex) + "--- Pai: " + str(self.parents[vertex]) + ", Peso: " + str(self.cost[vertex]) + "\n")
    f.close()

def total_cost_MST(self):
    total = 0.000
    for cost in self.cost:
        total += cost
    print(total)
    return total
```

Resultados

Grafo	Tempo médio das distâncias (seg)	Distância (10, 20)	Distância (10, 30)	Distância (10, 40)	Distância (10, 50)	Distância (10, 60)	Peso da MST
grafo_W_1_1	0.080	1.52	1.48	1.52	1.39	1.39	182.540
grafo_W_2_1	1.425	2.08	1.92	1.61	1.34	1.69	1859.41
grafo_W_3_1	341.6	1.98	2.08	2.14	1.82	2.23	18445.59
grafo_W_4_1	faltou tempo	-	-	-	-	-	-
grafo_W_5_1	-	-	-	-	-	-	-

Assim como o grafo 4, o grafo dos colaboradores não teve tempo suficiente de rodar

Discussão sobre os Resultados

Testamos variações sobre o uso de conjunto, mas a velocidade não alterou

```
while(S != V):
```

```
while(len(self.S) < self.graph.vertices_quantity-1):
```

Concluímos que nosso algoritmo está lento, mas não identificamos onde melhorar, esperamos ter um feedback para corrigir a velocidade