Trabalho realizado por:

Sebastião Manuel Inácio Rosalino, n.º 98437, turma CDA1

Trabalho 6 – EDA

Questão 1. Estudar a ordem de complexidade espacial da alínea 3.b) do Módulo 7 - Laboratório prático, indicando os cálculos e explicando a que corresponde cada parcela envolvida

Resposta à questão:

O exercício 3.b) do Módulo 7 - Laboratório prático solicita a "*implementação usando uma representação de listas de adjacências, com a construção da sequência de vértices e dos "contentores" associados a cada vértice como dicionários Python: a chave é o objeto vértice e o valor a lista de adjacências*".

Importa começar por explicitar os conceitos gerais que caracterizam a estrutura de dados correspondente a uma lista de adjacências.

Em termos gerais, uma **estrutura de lista de adjacências** agrupa as arestas de um grafo, armazenando-as em contentores (listas) menores e secundários que estão associados a cada vértice individual que, para este efeito, designamos por **(v)**.

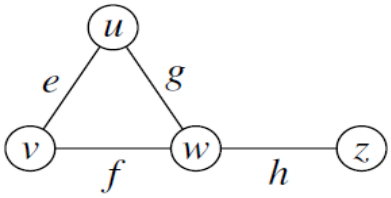
Para cada **vértice v**, é mantida uma **coleção I(v)**, chamada de **coleção de incidência de v**, cujas entradas são as arestas incidente para o **vértice v** (i.e. cada vértice aponta a sequência das arestas incidentes).

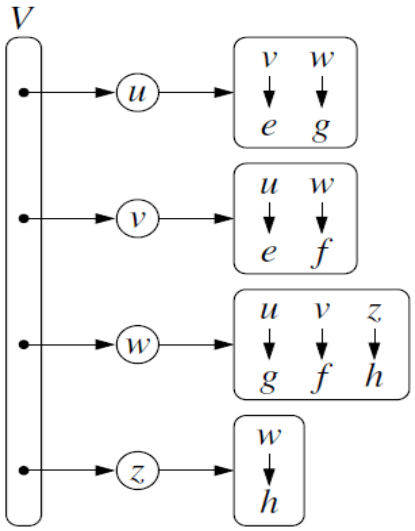
A coleção de **incidência I(v)** para um **vértice v** é uma lista, razão pela qual se chama a esta estrutura uma **representação gráfica de lista adjacente (grafo)**.

A **coleção V** é a lista primária de vértices. Cada vértice tem uma lista associada de arestas incidentes, que por sua vez tem uma ligação aos seus outros vértices adjacentes. No caso de um grafo orientado, existem 2 listas: a lista das arestas que entram e a lista das que saem de cada vértice **(v)**.

É requerido que a estrutura primária de uma lista adjacente mantenha a **coleção de V** de forma que seja possível localizar a **estrutura secundária I(v)** para um dado **vértice v** no tempo O (1). Isso pode ser feito usando uma lista posicional para representar **V**, com cada instância de vértice a ter uma referência direta à sua coleção **I(v)**.

Pode ver-se a representação do grafo com uma **estrutura de lista de adjacências** na figura seguinte:





Complexidade espacial

> A ordem de complexidade espacial de um programa ou algoritmo corresponde ao espaço de memória que o mesmo necessita para ser executado até ao fim.
>
> **S(n)**: "memória" utilizada pelo algoritmo para execução da tarefa, medida em função do tamanho n do input.

Em termos de espaço utilizado, o grau de complexidade é **S(n): O (n + m)** para uma representação gráfica com "n" vértices e "m" arestas. A lista primária de vértices usa espaço **O (n)**. A soma dos comprimentos de todas as listas secundárias traduz uma ocupação de espaço **O (m)**.

A ocupação de espaço naquela representação é:

- **n objetos do tipo vértice**
- **2m objetos do tipo aresta**

Em termos de referências para objetos a complexidade espacial: **2n**

Complexidade temporal:

Operação / método (com implementação)	**Ordem de Complexidade** *Running Time*
def **vertex_count**(self): return len(self._outgoing)	O (1)
def **vertices**(self): return self._outgoing.keys()	O (n) em que n é número de vértices

def **edge_count**(self):
  total = 0
  for v in self._outgoing:
  edges = self._outgoing[v]
  total += len(edges)
  return total if self.is_directed() else total // 2
	O (1)

def **edges**(self):
  result = set()
  for list_edges in self._outgoing.values():
  for e in list_edges:
  result.add(e)
  return result
	O (m)
	em que m é número de arestas

def **get_edge**(self, u, v): return self. outgoing[u].get(v)	O (min (deg(u),deg(v))) Para localizar uma determinada aresta (get_edge), podemos pesquise em I(u) e I(v). Ao escolher o menor dos dois, obtemos o tempo de execução **O (min (deg (u), deg (v)))**, onde deg (u) e deg (v) são os graus dos vértices u e v.

def **degree**(self, v, outgoing=True):
  adj = self._outgoing if outgoing else
  self._incoming
  return len(adj[v])
	O (1)

def **insert vertex**(self, x=None):
  v = self.Vertex(x)
  self. outgoing[v] = { }
  if self.is directed( ):
  self. incoming[v] = { }
  return v
	O (1)

def **insert edge**(self, u, v, x=None):
  e = self.Edge(u, v, x)
  self. outgoing[u][v] = e
  self. incoming[v][u] = e
	O (1)

def **remove_edge**(self, u, v):
  for e in self._outgoing[v]:
  x, y = e.endpoints()
  if u == x and v == y:
  self._outgoing[u].remove(e)
  self._incoming[u].remove(e)
	O (1)

def **remove_vertex**(self, v):
  for i in self.incident_edges(v):
  self.remove_edge(v, i)
  del self._outgoing[v]
  if self.is_directed():
  for i in self.incident_edges(v, False):
  self.remove_edge(i, v)
  del self._incoming[v]
  return v
	O (deg(v))
	onde deg (v) é o grau do vértice v

def **incident_edges**(self, v, outgoing=True):
  adj = self.outgoing if outgoing else
  self.incoming
  for edge in adj[v].values():
  yield edge
	O (deg(v))
	onde deg (v) é o grau do vértice v

Questão 2. Criar uma **representação gráfica do metro de Lisboa**, com diversas alíneas para implementação

A implementação em linguagem Python está disponível no ficheiro:

Trabalho_6_EDA_Sebastião_Rosalino_98437.py

Todo o código está devidamente documentado.

O programa oferece o seguinte menu:

```
"""O programa pode ser executado através de um menu com as seguintes opções:
----- Bem-Vindo ao menu do metro de Lisboa -----
0. Sair do menu
1. Consultar todas estações
2. Consultar todas as linhas
3. Consultar as ligações entre estações
4. Ver mapa do Metro de Lisboa
"""
```

A representação do metro de Lisboa produz a seguinte imagem (opção 4 do menu):