# Estruturas discretas e Problemas sobre grafos

**Ana Neri (ana.i.neri@inesctec.pt)**

DI,
Universidade do Minho

24 de Abril 2023

# Conteúdo

# Part I

# ESTRUTURAS DISCRETAS

# Dicionários

Um dicionário (array associativo ou função finita) a um tipo de dados constituídos por uma coleção de pares *(chave, informação)* em que nenhuma chave aparece repetida.

As operações normalmente associadas a este tipo são:

- adição de um novo par;
- modificação da informação associada a uma chave;
- procura da informação associada a uma chave;
- remoção de um par.

# Tabelas de Hash

## Ligação a Multi-conjuntos

As funções finitas (de associação chave $\rightarrow$ valor) são implementadas via dicionários também conhecidos conhecidos como tabelas de Hash.
Um multi conjunto é um dicionário: no qual cada elemento do multi-conjunto (chave) está associada a sua multiplicidade (informação).
Na implementação apresentada na semana passada usamos os índices do array como as chaves do dicionário. Esta alternativa só é viável se verificarmos algumas restrições.

### Questão

Quais são as restrições necessárias?

# Tabela de Hash

Se não conseguimos verificar as restrições então devemos dividir a correspondência que queremos armazenar em duas correspondências:

- **função de hash** que converte cada chave num índice de um array.
- um array onde se armazena a informação necessária.

Assim, para aceder/procurar/modificar um par *(k,i)* começamos por usar a função de hash à chave *k* para determinar o índice do array, com índice, podemos aceder/modificar à informação nessa posição.

Por outro lado, ainda temos 2 problemas:

- Temos de determinar se uma chave pertence ou não ao conjunto.
- Como o tamanho do array é tipicamente muito menor do que o domínio da correspondência a função de conversão entre chaves e índices é necessariamente não injetiva. Ou seja, chaves diferentes são convertidas no mesmo índice.

Por isso cada componente do array deve armazenar:

- uma marca que sinaliza se a componente do array está ser usada.
- O par *(chave, informação)*

# Tabela de Hash

Se quisermos armazenar dois pares $(k_1, i_1)$ e $(k_2, i_2)$ para os quais a função de hash aplicada a $k_1$ e $k_2$ coincide, dizemos que ocorre uma **colisão**.
Para resolver este problema usa-se:

- **Closed Addressing ou Chaining** - Guarda em cada posição do array todos os pares *(chave, information)* que colidem nessa posição.
- **Open Addressing** - Mantém todos os pares *(chave, informação)* num único array. A posição em que um dado par deverá ser armazenado/consultado é calculada usando a função de hash. No caso de acontecer uma colisão, é calculada uma nova posição para armazenar esse par (este calculo chama-se **probing** e pode ter várias soluções).

`dict` do Python usa Open Addressing para resolver o problema das colisões.

# Árvores de procura balanceadas

O bom comportamento de uma tabela de hash ocorre se conseguirmos uma tabela bem dimencionada.
Se existirem muitas perturbações à estrutura, isto é com updates das chaves que implicam rehashing (uma operação bastante pesada), ou se existir uma elevado número de colisões teremos comportamentos lineares em vez de constantes.

Se esperarmos ordem nas chaves então temos de repensar a estrutura, para algo que junte estruturas hierárquicas com tabelas de hash, nesse caso **árvores binárias** são uma alternativa.
Linguagens de programação como o JAVA incluem TreeMaps onde há uma estrutura tipo tabela de hash, mas com ordem nas chaves (tipicamente via árvores binária).

# Part II

# Problemas sobre grafos

# Introdução

Grafos são estruturas matemáticas para modelar relações entre elementos de um determinado conjunto.
Um grafo $G = (V, E)$ tem:

- um conjunto $V$ finito de vértices (ou nodos) que são os elementos que queremos relacionar;
- e um conjunto $E \subseteq V^2$ de arestas (ou arcos) que traduzem a relação entre vários elementos. Cada aresta tem 2 vértices, as extremidades da aresta.

**Questão**

Que exemplos de aplicação de grafos em informática conhece?

# Introdução

Se a ordem das extremidades da aresta não for relevante temos um **grafo não orientado**, e se tal não acontecer temos um **grafo orientado**, nessa caso as um aresta chama-se **origem** e a outra **destino**. As arestas podem ainda ter **peso** ou **custo**. Nestes casos temos um **grafo pesado**.

## Exercicio

Desenhe uma representação gráfica de um grafo orientado não pesado, em que os vértices são o conjunto $\{0, 1, 2, 3, 4, 5\}$.

# Implementação

Na implementação é costume separar-se a representação do conjunto das arestas e do conjunto dos vértices.
No conjunto das arestas devemos conseguir de forma eficiente:

- determinar se uma aresta existe (e se for o caso, qual o seu peso)
- percorrer o conjunto de vértices adjacentes de um dado vértice (isto é, percorrer as arestas com uma dada origem).

A forma mais eficiente de aceder à informação associada a um item é usando um array. Mas os arrays tipicamente usam como índices números inteiros não negativos e por isso é costume guardar a informação das arestas de um grafo como se os vértices desse grafo fossem índices de um array.
Na representação do conjunto de vértices, e se de facto esses vértices não forem os índices referidos atrás, deve ser armazenada a correspondência entre esses e os índices que lhe estão associados na representação das arestas.

... quadro

# Implementação
## Matriz Adjacente

A forma mais imediata de representar as arestas de um grafo consiste em usar uma matriz em que na posição $(i, j)$ se encontra a informação sobre a aresta com origem $i$ e destino $j$.

- num grafo não pesado podemos usar matrizes booleanas.
- num grafo pesado caso exista uma aresta temos de saber o seu peso, tipicamente podemos dar um peso 0 quando a aresta não existe.
- num grafo não orientado a aresta com origem em i e destino em j é indistinguível da aresta com origem em j e destino em i, logo usamos uma matriz simétrica.

O custo em termos de memória de uma representação em matriz é proporcional $V^2$ e por isso independente do nº de arestas do grafo.
Tipicamente, o nº de arestas é muito inferior ao limite superior, dado que a maioria dos vértices está relacionada com uma percentagem muito pequena dos vértices.
Quando a maior parte dos elementos das matrizes são nulos é comum usar técnicas conhecidas como armazenamento de matrizes esparsas.

# Implementação
## Listas de Adjacência

Uma forma de armazenar uma matriz esparsa é armazenando apenas os elementos não nulos, agrupados por linhas, por exemplo numa lista ligada. A esta representação chama-se listas de adjacência.
Por outras palavras, por cada linha da matriz *(vértice do grafo)* armazenamos os elementos não nulos *(as arestas que têm esse vértice como origem)* indicando a coluna *(o destino da aresta)* e o valor *(peso)*.
Se o grafo for não pesado a definição não precisa de incluir o peso.
No caso de grafos orientados é costume armazenar duas arestas (direcionadas) por cada aresta do grafo.

# Implementação

## Vetor de Adjacência

Se precisarmos de compactar ainda mais representação anterior podemos ainda guardar todas as arestas do grafo num único vetor, mantendo contíguas as arestas que têm a origem em comum.

Para ter acesso aos vértices ligados a um determinado vértice (vértices adjacentes) precisamos apenas de guardar os índices referentes ao primeiro e último vértice adjacente. Na verdade chega guardar o índice onde se encontra o primeiro sucessor dado que o último pode ser obtido a partir do índice onde começam os sucessores do vértice seguinte.

# Consultas

Para comparar as formas de aceder a informação nas várias implementações do grafo vamos considerar as seguintes funções de consulta:

- Peso de uma aresta
- Grau de entrada e saída

# Consultas

## Peso de uma aresta

Dado um grafo e dois vértices o objetivo é determinar caso exista o peso da aresta que os liga.
Esta função deve devolver verdadeiro ou falso, e em caso afirmativo deve colocar num dado endereço o peso da aresta.
No caso da matriz de adjacência esta função executará em tempo constante:

```
1 def edge_weight(self, src, dest):
2     return self.matrix[src][dest]
```

# Consultas

## Peso de uma aresta

No caso da lista de adjacência a aresta deve ser procurada na listas dos vértices adjacentes primeiro:

```
def edge_weight(self, src, dest):
    for edge in self.adj[src]:
        if edge.dest == dest:
            return edge.cost
    return False
```

### Questão

Qual é o pior caso e qual a complexidade assimptótica do tempo de execução?
Qual é o melhor caso e qual a complexidade assimptótica do tempo de execução?

# Consultas

## Peso de uma aresta

No caso dos vetores de adjacência a diferença está na forma como os vértices adjacentes são percorridos. A complexidade da função é igual à última.

```
def edge_weight(self, src, dest):
    for k in range(self.vertices[src], self.vertices[src + 1]):
        if self.edges[k].dest == dest:
            return self.edges[k].cost
    return False
```

# Consultas

## Grau de entrada e saída

Dado um grafo e um vértice desse grafo, o grau de entrada desse vértice define-se como o número de arestas que têm esse vértice como destino. O grau de saída de vértice corresponde ao número de arestas que têm esse vértice como origem.
Na implementação de matrizes de adjacência estes dois problemas têm uma resolução idêntica:

```
1  def indegree(self, v):
2      r = 0
3      for i in range(NV):
4          if self.matrix[i][v] != NE:
5              r += 1
6      return r
7  def outdegree(self, v):
8      r = 0
9      for i in range(NV):
10         if self.matrix[v][i] != NE:
11             r += 1
12     return r
```

**Questão**

Qual é a complexidade média destas funções?

# Consultas

## Grau de entrada e saída

Nas outras representações o acesso aos sucessores de um vértice é bastante mais eficiente do que o acesso aos antecessores. Logo o calculo do grau de entrada é bastante menos eficiente.
No caso das listas de adjacência o cálculo do grau de saída é o calculo de comprimento de uma lista ligada.

```
1 def outdegree(self, v):
2     r = 0
3     for it in self.adj[v]:
4         r += 1
5     return r
```

### Questão

Normalmente a complexidade desta função tem um comportamento melhor do que o apresentado para a definição baseada em matrizes. Explique porquê?

# Consultas

## Graude entrada e saída

No caso do grau de entrada, e como não temos as arestas com um destino comum organizadas de qualquer forma, teremos que percorrer todas as arestas e por isso a função terá complexidade linear no tamanho do grafo ($\Theta(V + E)$).

```
def indegree(self, v):
    r = 0
    for i in range(NV):
        for it in self.adj[i]:
            if it.dest == v:
                r += 1
    return r
```

# Consultas

## Grau de entrada e saída

No caso dos vetores de adjacência o cálculo do grau de saída é bastante eficiente.

```
def outdegree(self, v):
    return self.vertices[v + 1] - self.vertices[v]
```

No caso do grau de entrada a complexidade é muito próxima do que apresentou na listas de adjacência.

```
def indegree(self, v):
    r = 0
    for i in range(len(self.edges)):
        if self.edges[i].dest == v:
            r += 1
    return r
```

Este exemplo de consulta da informação de um grafo evidencia a principal diferença nos diferentes algoritmos. A diferença nas implementações reside na forma como percorremos os vértices adjacentes a um dado vértice.

# Próximos episódios

- Algoritmos sobre grafos continuação