G2. Algoritmos de Travessia de Grafos

(por vezes também designados por algoritmos de **pesquisa**)

PROJECTO CODEBOARD DE SUPORTE A ESTE MÓDULO:
https://codeboard.io/projects/10725

Caminhos em grafos orientados

Num grafo (V, E), um *caminho* do vértice v_0 para o vértice v_k é uma sequência de vértices

$$\langle v_0, v_1, \ldots v_k \rangle$$

tais que $v_i \in V$ para todo o $i \in \{0, \ldots, k\}$, e $(v_i, v_{i+1}) \in E$ para todo o $i \in \{0, \ldots, k-1\}$

Alternativamente, este caminho pode ser visto como uma sequência de arestas

$$(v_0, v_1), (v_1, v_2), \ldots (v_{k-1}, v_k)$$

O *comprimento* deste caminho é o número k de arestas nele contidas.

Um vértice v é **alcançável** a partir do vértice s se existe um caminho de s para v.
Num grafo orientado, isto não implica que s seja alcançável a partir de v.

Um **ciclo** é um caminho de comprimento ≥ 1 com início e fim no mesmo vértice.
(Note-se que existe sempre um caminho de comprimento 0 de um vértice para si próprio, que não se considera ser um ciclo)

Um grafo diz-se *acíclico* se não contém ciclos. Um **grafo orientado acíclico** é usualmente designado por **DAG**, de *Directed Acyclic Graph*.

Florestas e Árvores

Uma **árvore** (G, V) (ou no caso mais geral uma **floresta**) é um caso particular de um DAG: um grafo orientado que, além de acíclico, satisfaz a seguinte restrição:

> *O grau de entrada de todo o vértice $v \in V$ é 0 ou 1.*

Se $(u, v) \in E$ dizemos que o vértice u é o "pai" de v.

Os vértices com grau de entrada 0 são as *raízes* da floresta (vértices sem "pai"). Caso exista apenas uma raíz estamos em presença de uma árvore.

Travessia de Grafos

Um algoritmo de **travessia** de um grafo (V, E), a partir de um vértice inicial $s \in V$ dado, visita **todos os vértices alcançáveis** a partir de s, percorrendo caminhos seguindo uma determinada estratégia, **não passando mais do que uma vez por cada vértice.**

Os caminhos percorridos durante a travessia constituem um sub-grafo de (V, E) que é de facto uma árvore, designada por **árvore de travessia** do grafo. Diferentes estratégias produzirão árvores diferentes.

Note-se que não se trata aqui de um algoritmo para a resolução de um problema específico, mas antes de uma família de algoritmos. Para **resolver um problema específico** haverá que:

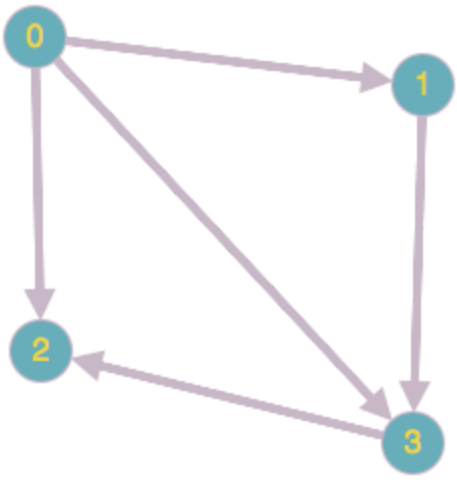
1. seleccionar a estratégia de travessia adequada para esse problema
2. adaptar o esquema geral da travessia para a resolução do problema em causa

Representação de árvores de travessia

Uma árvore de travessia pode ser representada por um simples *array* indexado pelos vértices do grafo, que associe a cada vértice v o seu pai na árvore, i.e. o vértice a partir do qual v foi alcançado durante a travessia. Utiliza-se por exemplo o valor -1 para identificar a raíz da árvore.

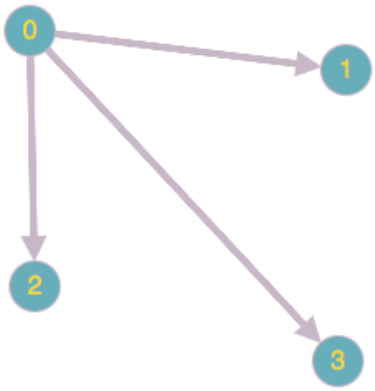
EXEMPLO

[O grafo seguinte foi desenhado em http://graphonline.ru/en/]

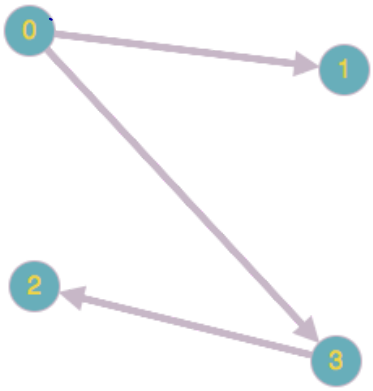


Grafo orientado G2

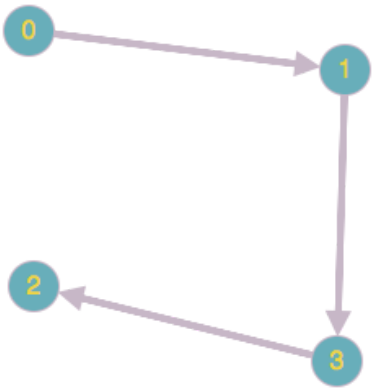
Existem várias travessias diferentes possíveis de G2 com início no vértice 0.
As árvores de travessia seguintes correspondem a 3 delas:



pais = [-1, 0, 0, 0]



pais = [-1, 0, 3, 0]



pais = [-1, 0, 3, 1]

Estado de um vértice e estruturas de dados auxiliares

Durante a execução de uma travessia, cada vértice de um grafo pode encontrar-se num de três estados, a que associaremos um código de cores:

- ainda não alcançado pela travessia [**BRANCO**]
- já alcançado, mas alguns dos seus vértices adjacentes ainda não alcançados [**CINZENTO**]
- já processado (todos os seus adjacentes foram alcançados) [**PRETO**]

Observe-se que

> *Os vértices cinzentos constituem uma fronteira entre os que já foram completamente tratados pela travessia e os que não foram ainda alcançados.*

Para o controlo da travessia, um algoritmo deve guardar esta informação de estado (num *array* indexado pelos vértices do grafo).

Dependendo da aplicação, pode não ser importante distinguir entre os vértices CINZENTOS e PRETOS; neste caso a informação de estado será simplesmente Booleana (**não visitado** / **visitado**).

Travessia em Largura

Esta estratégia de travessia caracteriza-se pelo seguinte:

> *Todos os vértices à distância k de s são visitados antes de qualquer vértice à distância $k+1$ de s.*

Trata-se pois de uma travessia *por níveis* de distância desde a origem. Em qualquer ponto da execução, os vértices cinzentos poderão estar no máximo em dois níveis diferentes consecutivos.

> *Como se define a noção de **distância** entre dois vértices?*

Esta estratégia exige a utilização de uma *fila de espera* como estrutura de dados auxiliar de controlo, onde serão armazenados os vértices que num determinado

momento se encontram cinzentos. A fila de espera é inicializada inserindo-se o vértice em que se pretende começar a travessia.

Passo básico

1. É retirado (*dequeued*) um vértice cinzento u da fila de espera, e processado de acordo com o problema concreto que se pretende resolver;
2. É percorrida a sua lista de adjacências, e para cada vértice v adjacente a u:
   a. **caso esteja ainda branco**, v é inserido (*enqueued*) na fila de espera, passando a cinzento;
3. u é marcado com cor preta.

Nota: o passo "e processado de acordo com o problema concreto que se pretende resolver" pode ser executado em diferentes momentos: quando o nó entra para a fila de espera (fica cinzento); quando é retirado dessa fila (como ilustrado acima); ou finalmente quando passa a preto.

A execução terminará quando a fila de espera ficar vazia.

Código em Python

A seguinte função imprime as mudanças de estado dos vértices ao longo da execução, e constrói um vector 'parent' contendo a árvore de travessia.

```
for v in g:
    color[v] = 'WHITE'
    parent[v] = ''


# Breadth-first traversal of graph 'g' taking as source the ve
rtex 's'
# Prints color changes and constructs traversal tree in 'paren
t'.
def bf_visit(g, s):
    color[s] = 'GRAY'
    print s, 'GRAY'
    parent[s] = '(ROOT)' ou -1 ou raiz
    q.enqueue(s)
```

```
    while(not q.isEmpty()):
        u = q.dequeue()
        for v in g[u]:          # Em C: travessia de uma lista de adjacências!
            if color[v] == 'WHITE':
                color[v] = 'GRAY'
                print v, 'GRAY'
                parent[v] = u
                q.enqueue(v)
        color[u] = 'BLACK'
        print u, 'BLACK'
```

não podemos prever o nº de adjacentes que v tem

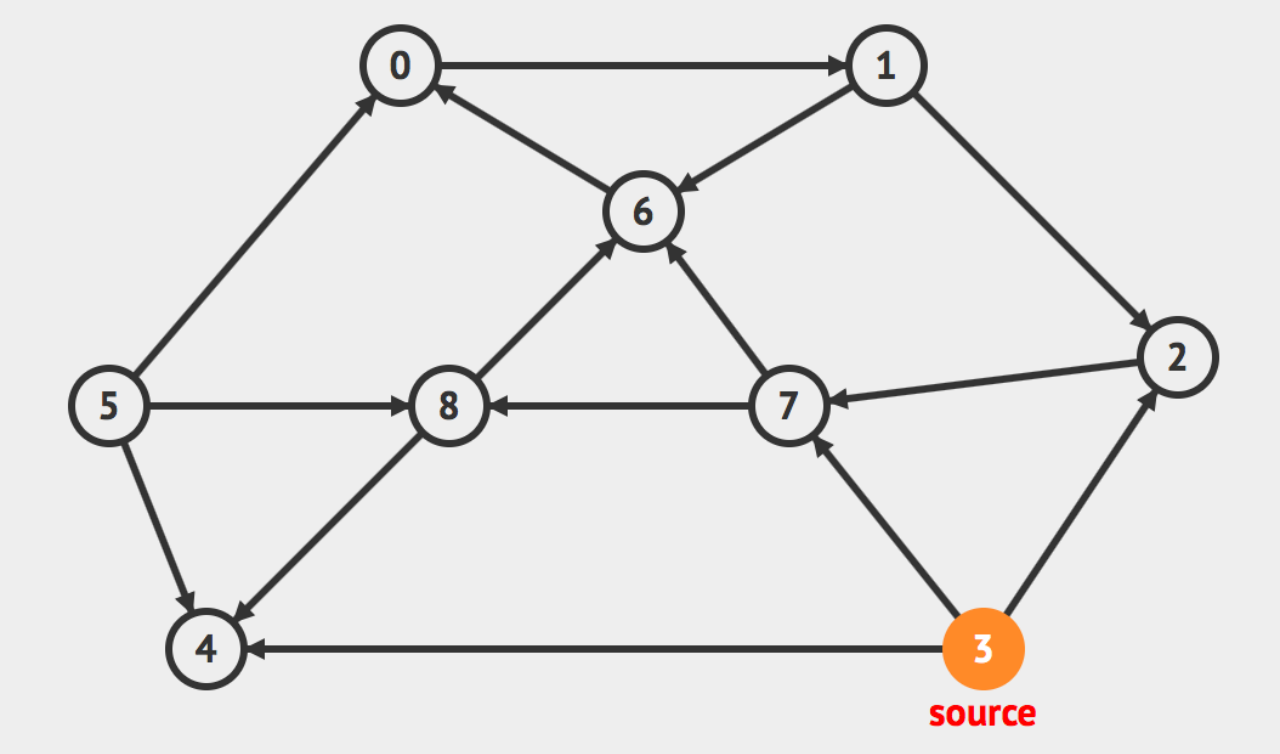
Observe-se que

> *A construção da árvore de travessia não é obrigatória: poderá ser ou não necessária, de acordo com o objectivo específico da travessia que se pretende implementar.*

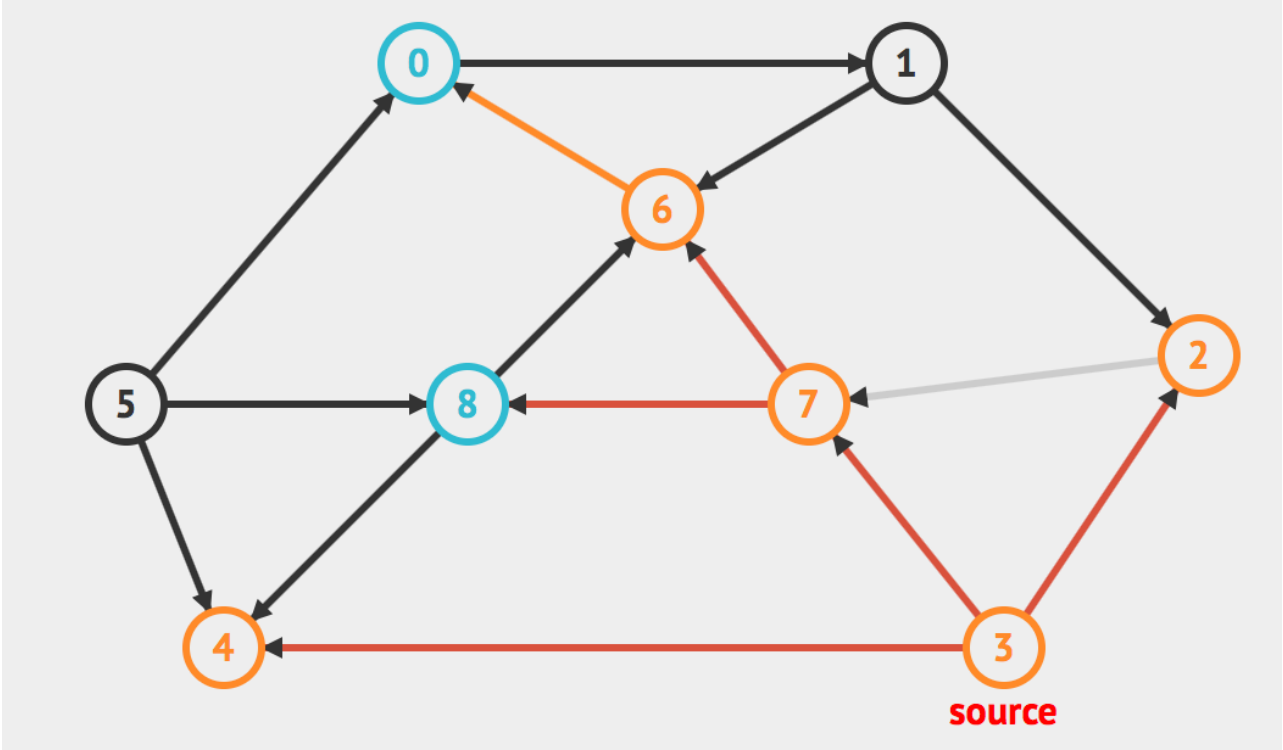
Exemplo

[As imagens foram obtidas a partir de uma animação em https://visualgo.net/en/dfsbfs]

No grafo orientado seguinte, será efectuada uma travessia em largura com início no vértice 3



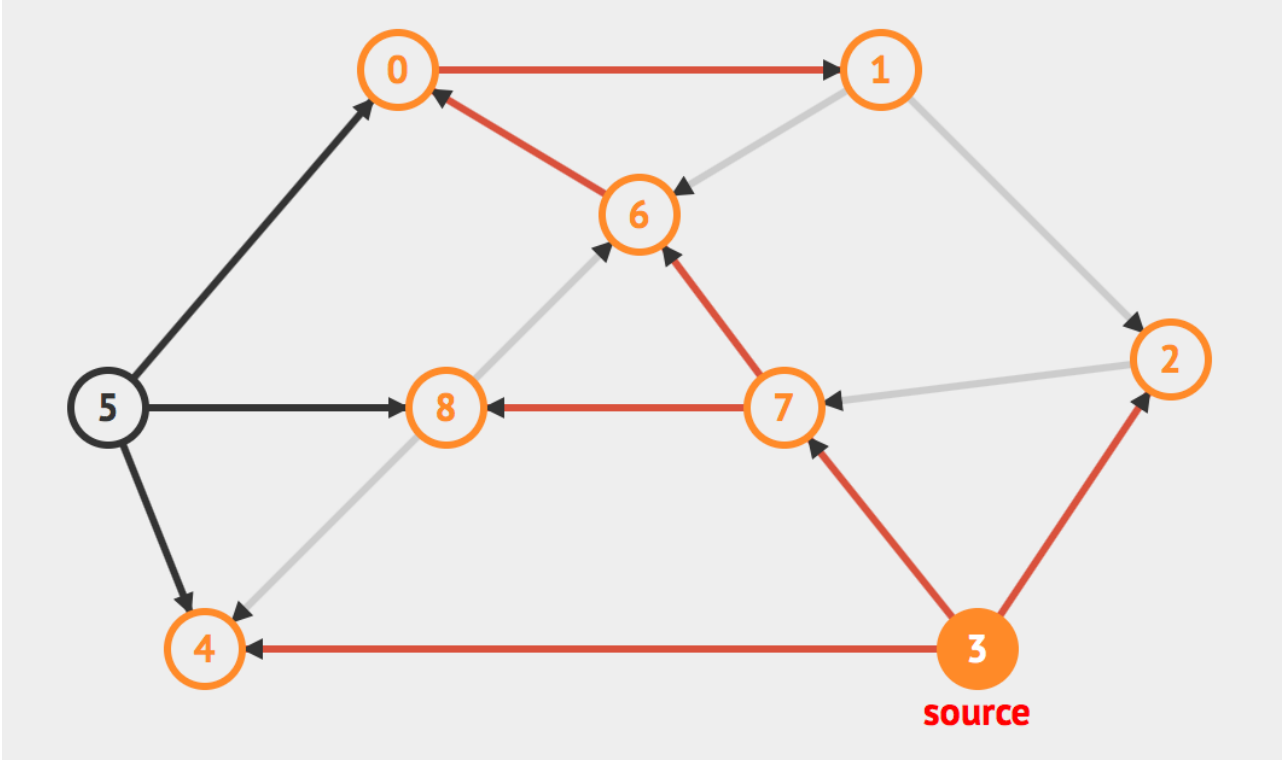
A meio da execução podemos observar que a fila de espera (vértices a azul) pode conter vértices que pertencem a dois níveis diferentes da árvore de travessia, i.e., a distância a que estão da origem não é a mesma (diferindo numa unidade).



Observe-se que, sendo a travessia feita em *largura:*

- 7 é visitado a partir de 3 e não de 2
- 8 é visitado antes de 0 e 1, uma vez que entra para a fila de espera logo a seguir a 6

Terminada a travessia temos a árvore final:



Para sabermos qual o caminho mais curto até 6, vamos à árvore de travessia e vemos que o pai de 6 é o 7, o pai do 7 é o 3 e o 3 não tem pai e é a raiz, logo o caminho mais curto até 6 é 3 → 7 → 6

Análise do tempo de execução

- Uma vez que o passo básico envolve percorrer todos os vértices adjacentes a u, **assumiremos que o grafo é representado por listas de adjacências** — é a representação mais natural para a aplicação deste algoritmo

- Para o **pior caso**, assumimos que *todos os vértices do grafo são alcançados a partir de* s

- Cada vértice é *enqueued* e *dequeued* exactamente uma vez. Isto é garantido pelo facto de os vértices nunca serem pintados de branco depois da inicialização

- A lista de adjacência de cada vértice é percorrida exactamente uma vez (quando o vértice é *dequeued)*, e o comprimento *total* das listas é $\Theta(|E|)$. Logo, o tempo total tomado pela travessia das listas de adjacência é $\Theta(|E|)$

- Não podemos no entanto concluir precipitadamente que o algoritmo executa em tempo $\Theta(|E|)$, uma vez que as operações de inicialização, e também o conjunto de operações de acesso à fila de espera, são executadas em tempo $\Theta(|V|)$

- Sendo assim, temos $T_p(V, E) = \Theta(|V| + |E|)$

Tempo linear no tamanho da representação por listas de adjacências

- No **melhor caso** o vértice s não tem adjacentes, e $T_m(V, E) = \Theta(|V|)$, devido ao tempo de inicialização do vector de cores

OBSERVAÇÃO

Na análise de algoritmos sobre grafos simples (i.e. que **não são multi-grafos**) leva-se por vezes a identificação do pior caso mais longe: tendo em conta que o valor máximo de $|E|$ é dado por $|V|^2$, quando o grafo é *completo*, podemos dizer que $T_p(V, E) = \mathrm{O}(|V|^2)$.

Caminho Mais Curto e Distância entre dois vértices

Uma consequência da escolha desta estratégia é o seguinte resultado, que pode ser provado formalmente como parte da correcção do algoritmo:

> *A árvore de travessia construída contém todos os **caminhos mais curtos** com origem em s e destino em cada um dos vértices alcançáveis a partir de s.*

Ou seja, para encontrar o caminho mais curto (i.e. o caminho com o menor comprimento) de s até um vértice d, basta:

1. efectuar uma travessia em largura com origem em s, e
2. identificar depois o caminho de s para d na árvore de travessia construída.

O comprimento do caminho mais curto entre s e d designa-se por *distância* entre os dois vértices. Uma travessia em largura pode ser usada para calcular distâncias.

No entanto, é importante perceber que o cálculo de distâncias / caminhos mais curtos é apenas uma aplicação possível de um algoritmo de travessia em largura.

Travessia em Largura Completa de um Grafo

Dependendo da aplicação, poderá ser importante alcançar *todos os vértices* de um grafo, sendo para isso necessário iniciar mais do que uma travessia. Basta percorrer todos os vértices do grafo, iniciando uma travessia em cada vértice que não tenha ainda sido alcançado pelas travessias anteriores:

```
# Travessia em largura completa, produzindo uma floresta de tr
avessia
def bfs(g):
    for u in g:
        if color[u] == 'WHITE':
            bf_visit(g, u)
```

Este algoritmo executa agora necessariamente em tempo $T(V, E) = \Theta(|V| + |E|)$ em *todos os casos!*

Travessia em Profundidade

Esta estratégia de travessia caracteriza-se pelo seguinte:

> *Todos os vértices adjacentes a um vértice v são visitados imediatamente a seguir a v*

Enquanto na travessia em largura os adjacentes a um vértice que acaba de ser visitado ficam em espera numa fila, e só serão por isso tratados depois de todos os que estão entretanto pendentes, na travessia em profundidade esses adjacentes serão os primeiros a ser visitados.

A árvore de travessia é pois construída em profundidade, ramo a ramo, e não por níveis como anteriormente. Por esta razão, uma travessia em profundidade **não pode ser utilizada**:

- para calcular distâncias entre vértices, nem
- para calcular caminhos mais curtos entre vértices.

Em termos de implementação, basta alterar a estrutura de dados auxiliar que armazena os vértices cinzentos (descobertos mas ainda não visitados), substituindo a fila de espera por uma **pilha,** mais uma vez inicializada inserindo-se o vértice em que se pretende começar a travessia. enqueue → push
dequeue → pop

Existe no entanto uma alternativa frequentemente utilizada, semelhante a um algoritmo típico de travessia de uma árvore, e que passa pela utilização de **recursividade**, substituindo-se assim a utilização explícita de uma pilha pela estrutura recursiva das chamadas de função.

Código em Python
A seguinte função imprime as mudanças de estado dos vértices ao longo da execução, e constrói um vector 'parent' contendo a árvore de travessia.

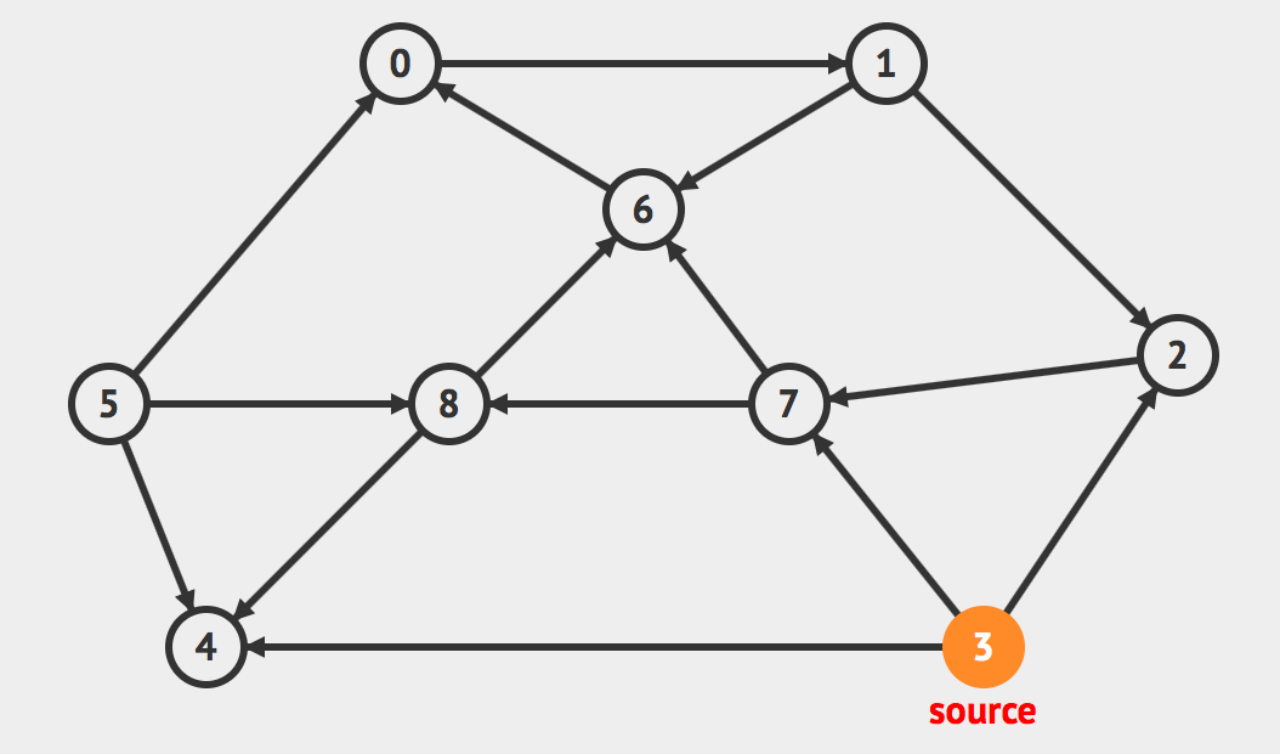
```
for v in g:
    color[v] = 'WHITE'
    parent[v] = ''

# Depth-first traversal of graph 'g' taking as source the vert
ex 's'
# Prints color changes and constructs traversal tree in 'paren
t'.
def df_visit(g, s):
    color[s] = 'GRAY'
    print s, 'GRAY'
    for v in g[s]:
        if color[v] == 'WHITE':
            parent[v] = s
            df_visit(g, v)
    color[s] = 'BLACK'
    print s, 'BLACK'
```

Tal como na travessia em largura, podemos facilmente implementar uma travessia completa iniciando uma travessia sucessivamente em todos os vértices do grafo, desde que não tenham ainda sido visitados por uma travessia anterior:

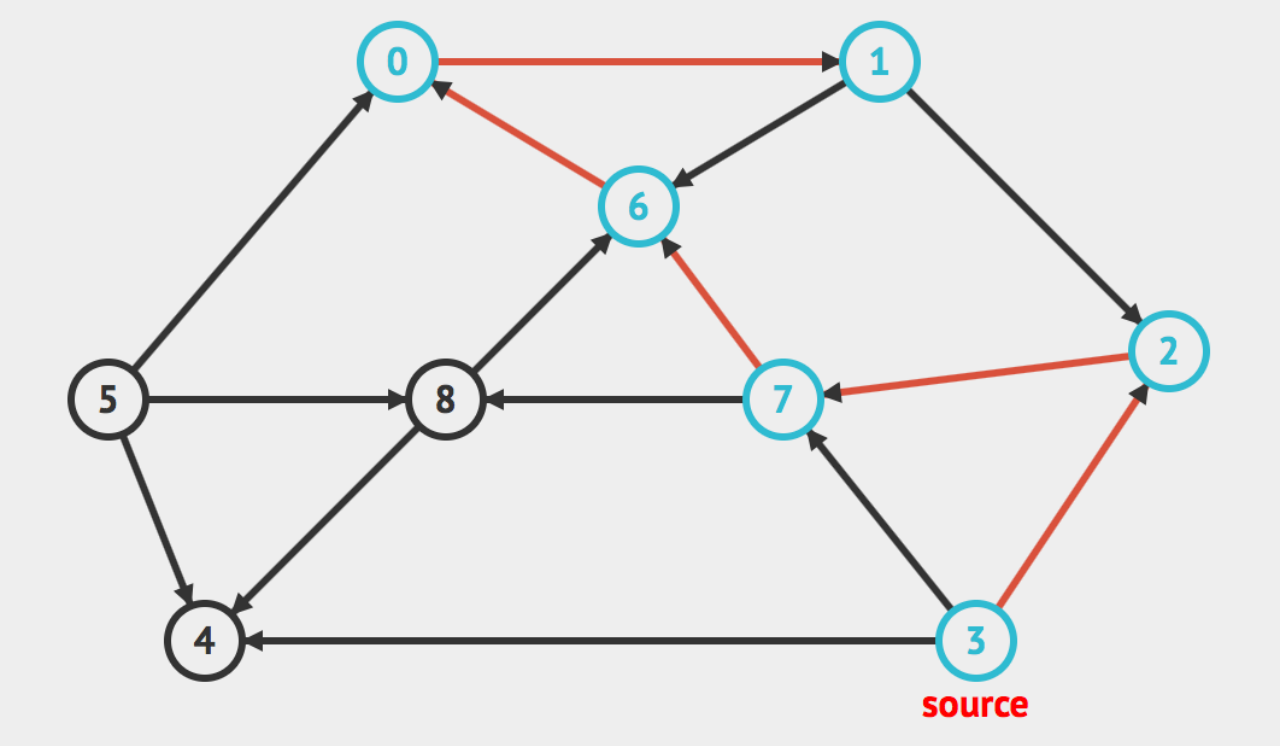
```
def dfs(g):
    for u in g:
        if color[u] == 'WHITE':
            df_visit(g, u)
```

Exemplo
[As imagens foram obtidas a partir de uma animação em https://visualgo.net/en/dfsbfs]

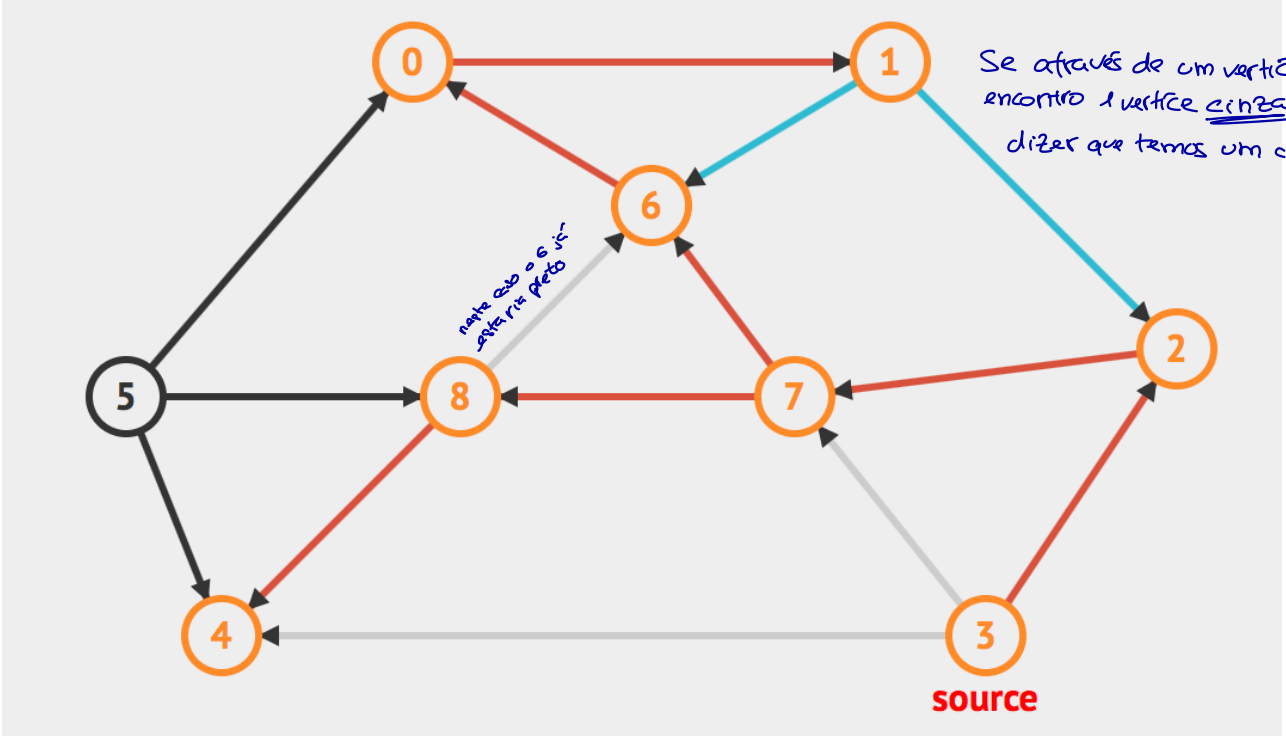
Novamente iniciaremos uma travessia no vértice 3 do seguinte grafo, mas desta vez em profundidade:



Na seguinte imagem intermédia é visível que os vértice 0 e 1 são visitados a partir de 6 antes de 8, que tal como 6 é adjacente a 7:



Na seguinte imagem final pode observar-se a vermelho a árvore de travessia em profundidade.



QUESTÃO: o que representarão as arestas a azul?

Análise do tempo de execução

- Mais uma vez, a análise do pior caso de `df_visit` coincide com a análise do caso único da travessia completa `dfs`

- Em `dfs` , o ciclo `for` executa em tempo $\Theta(|V|)$, excluindo o tempo tomado pelas execuções de `df_visit`. O mesmo acontece com os ciclos de inicialização dos vectores.

- `df_visit` é invocada *exactamente uma vez* para cada vértice do grafo (a partir de `dfs` ou da própria `df_visit`), uma vez que é invocada apenas com vértices brancos e a primeira coisa que faz é pintá-los de cinzento (e nenhum vértice volta a ser pintado de branco).

- Em `df_visit(g, u)`, o número de iterações do ciclo `for` é dado pelo número de vértices adjacentes de `u` . No total das invocações de `df_visit`, este ciclo é

então executado $\Theta(|E|)$ vezes

- Sendo assim, temos mais uma vez $T(V, E) = \Theta(|V| + |E|)$

Tempo linear no tamanho da representação por listas de adjacências

EXERCÍCIOS

[retomando https://codeboard.io/projects/10725]

1. Com base nas estruturas de dados que definiu anteriormente, implemente em C o algoritmo de travessia em largura.
   a. Altere o algoritmo por forma a calcular as distâncias do vértice s a todos os vértices alcançáveis a partir dele, colocando-as num array.
   b. Escreva uma função que **imprime** o **caminho mais curto** entre quaisquer dois vértices a e b de um grafo, com base numa travessia em largura.
   c. Implemente igualmente em C o algoritmo de travessia completa.

1. Com base nas estruturas de dados que definiu anteriormente, implemente em C o algoritmo de travessia em profundidade.
   a. Defina uma função que calcula se um grafo é ou não cíclico, com base numa travessia em profundidade.
   b. Implemente igualmente em C o algoritmo de travessia completa em profundidade.

Código Python para teste dos algoritmos

```
class Queue:
    def __init__(self):
        self.items = []
```

```
    def isEmpty(self):
        return self.items == []
    def enqueue(self, item):
        self.items.insert(0,item)
    def dequeue(self):
        return self.items.pop()
    def size(self):
        return len(self.items)

g = {}
g[0] = [1]
g[1] = [2, 6]
g[2] = [7]
g[3] = [2, 4, 7]
g[4] = []
g[5] = [0, 4, 8]
g[6] = [0]
g[7] = [6, 8]
g[8] = [4, 6]


color = {}
parent = {}
q=Queue()

print "\nGraph:"
for v in g:
    print v, "->", ", ".join([str(u) for u in g[v]])

print "\nDFS:"
```

```
# bf_visit(g, 3)
# bfs(g)
# df_visit(g, 3)
dfs(g)
print "\nTraversal tree / forest:"
for j in sorted(parent.keys()):
    print j, "<-", parent[j]
```