1. Teoria de Grafos

1.1. Definições Básicas

A Teoria de Grafos é um ramo da matemática que estuda as relações entre os objetos de um determinado conjunto. Um **grafo** é uma estrutura composta por **vértices** (nós) e **arestas** (arcos), de tal maneira que diversos problemas podem ser formulados utilizando esta estrutura.

Formalmente falando, um grafo $G = (V, E)$ é definido da seguinte forma:

- V é o conjunto finito de elementos denominados vértices e
- E é o conjunto finito de pares **não ordenados** de vértices chamados de arestas. Na prática, uma aresta conecta dois vértices.

A Figura 1 apresenta um exemplo de grafo, em que $V = \{A, B, C, D\}$ e $E = \{(A, B), (A, C), (B, D)\}$ Como mencionado anteriormente, E é um conjunto não ordenado de vértices, o que significa que as arestas $e_1 = (A, B)$ e $e_2 = (B, A)$ representam a mesma relação e, por isso, apenas uma delas é incluída no conjunto E.

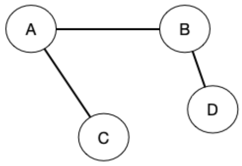


Figura 1: Exemplo de um grafo.

Algumas definições adicionais incluem: seja $e = (A, B)$ uma aresta tal que $e \in E$. Dizemos que A e B são os **extremos** desta aresta, e que A e B são vértices **adjacentes**. Ademais, temos que o **tamanho** do grafo G é dado por $|V| + |E|$, em que $|V|$ e $|E|$ denotam, respectivamente, o número de vértices e arestas. No exemplo da Figura 1, temos que $|V| = 4$ e $|E| = 3$.

Dizemos que um grafo é **simples** (Figura 1, por exemplo) quando o mesmo não possui **laços** ou **arestas múltiplas**. Um laço é uma aresta com extremos idênticos, ao passo que arestas múltiplas são definidas como sendo duas ou mais arestas com o mesmo par de extremos, conforme demonstrado na Figura 2.

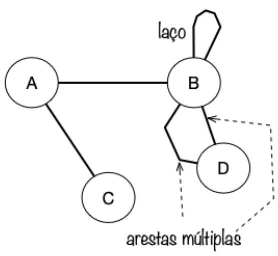
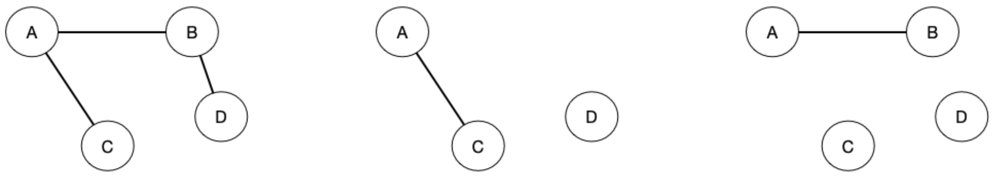


Figura 2: Exemplo de laços e arestas múltiplas em um grafo.

Dizemos que $H = (V', E')$ é um **subgrafo** de $G = (V, E)$ quando $V' \subseteq V$ e $E' \subseteq E$. Já o grafo $J = (V, E')$ é dito ser subgrafo **gerador** de G quando $E' \subseteq E$. A Figura 3 ilustra esses conceitos.



(a) (b) (c)

Figura 3: (a) grafo G, (b) subgrafo H e (c) subgrafo gerador J.

Já com relação ao **grau** de um vértice $v \in V$, denotado por $d(v)$, o mesmo diz respeito à quantidade de nós adjacentes à v, sendo que laços contam duas vezes. Na Figura 4, por exemplo, temos que $d(A) = 2$, $d(B) = 4$, $d(C) = 1$ e $d(D) = 1$.

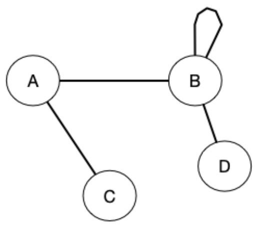


Figura 4: Exemplo de um grafo para contagem de graus de seus nós.

De acordo com o teorema conhecido por **Handshake Lemma**, temos a seguinte afirmação:

$$\sum_{v \in V} d(v) = 2\,|E|\,. \tag{1}$$

Vamos tentar provar por indução? A prova será realizada de acordo com o **número de vértices** no conjunto V. Conforme vimos em aulas anteriores, uma prova por indução possui os seguintes passos:

1. <u>Base</u>: seja $G = (V, E)$ um grafo com apenas um vértice v, ou seja, $|V| = 1$. Neste caso, existem duas possibilidades: (i) $d(v) = 0$ e (ii) $d(v) = 2$. Na primeira opção, temos que o número de arestas é igual à zero, ou seja, $|E| = 0$. Assim sendo, a Equação 1 é verdadeira, ou seja, $\sum_{v \in V} d(v) = 2\,|E| = 2.0 = 0$ Para o segundo caso, temos que G possui um laço no vértice v, ou seja, $d(v) = 2$. Neste caso, a Equação 1 também é verdadeira, ou seja, $\sum_{v \in V} d(v) = 2\,|E| = 2.1 = 2$
2. <u>Hipótese de Indução</u>: seja $G = (V, E)$ um grafo tal que a Equação 1 seja verdadeira.
3. <u>Passo de Indução</u>: seja $G' = (V', E')$ um grafo tal que $V' = V \cup \{v^\star\}$, ou seja, G' é formado pelos mesmos vértices de G acrescido do nó $v^\star$. Assim sendo, de acordo com o Handshake Lemma, temos que:

$$\sum_{v' \in V'} d(v') = 2\left|E'\right|. \tag{2}$$

A equação acima pode ser escrita da seguinte forma:

$$\sum_{v' \in V'} d(v') = \sum_{v \in V} d(v) + 2d(v^\star), \tag{3}$$

o que significa, na prática, que a somatória dos graus dos vértices de G' é igual à somatória dos graus dos vértices de G' acrescida do dobro do grau do novo vértice $v^\star$. Mas por que o dobro? A razão é que, ao adicionarmos v' no grafo G, temos que grau de cada vértice adjacente à $v^\star$ vai **aumentar** em uma unidade.

Pela hipótese de indução (Equação 2), podemos representar a Equação 3 como segue:

$$\sum_{v' \in V'} d(v') = \sum_{v \in V} d(v) + 2d(v^\star) = 2\,|E| + 2d(v^\star) = 2(|E| + d(v^\star)). \tag{4}$$

Temos que o número de arestas de G' é dado por $|E| + d(v^\star)$, ou seja, o número de arestas do grafo G' é dado pelo número de arestas de G acrescido da quantidade de vizinhos do novo nó $v^\star$. Desta forma, temos que $|E'| = |E| + d(v^\star)$. Finalmente, a Equação 4 pode ser representada da seguinte forma:

$$\sum_{v' \in V'} d(v^\star) = 2(|E| + d(v^\star)) = 2\,|E'|. \tag{5}$$

Um **caminho** π em um grafo G é uma sequência finita e não vazia de vértices **sem repetição**, como ilustra a Figura 5. Neste caso, temos o caminho em vermelho $\pi = \{A, B, D\}$. Já o **comprimento** c de um caminho é dado pela quantidade de vértices que o mesmo visita, ou seja, $c(\pi) = 3$.

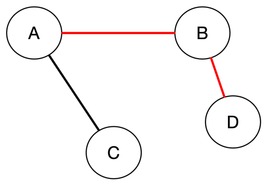


Figura 5: Exemplo de caminhos em grafos.

Dizemos que um grafo G é **conexo** se, para quaisquer dois vértices, existe um caminho conectando-os. Por outro lado, caso esta condição não possa ser satisfeita, dizemos que o grafo é **desconexo**, como ilustra a Figura 6.

(a) (b)

Figura 6: Exemplos de grafo: (a) conexo e (b) desconexo.

Finalmente, seja um grafo $G = (V, E)$ e dois nós $v_1, v_2 \in V$. Dizemos que G é **cíclico** quando existem, pelo menos, dois ou mais caminhos que possam v_1 e v_2. Caso contrário, G é dito ser **acíclico**. A Figura 6 ilustra essas situações, em que o grafo apresentado na Figura 6a é dito ser cíclico porque existem dois caminhos distintos (azul e vermelho) que conectam os nós B e C.

(a) (b)

Figura 6: Exemplos de grafo: (a) cíclico e (b) acíclico.

Os grafos também podem ser **ponderados** nas arestas, em que uma função $w : V \times V \rightarrow \Re$ é responsável por associar um valor à cada uma delas. A Figura 7 ilustra um exemplo de grafo ponderado.

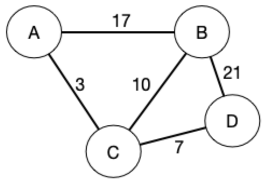


Figura 7: Exemplo de grafo ponderado, em que $w(A, B) = 17$.

Os tipos de grafos que vimos até agora são chamados de **grafos não direcionados** ou **grafos não orientados**. No entanto, existem grafos cujas arestas contemplam pares **ordenados** de vértices, sendo chamados de **grafos direcionados** ou **grafos orientados**. A Figura 8 apresenta um exemplo de grafo direcionado. Neste caso, nosso grafo $G = (V, E)$ é composto da seguinte maneira: $V = \{A, B, C, D\}$ e $E = \{(B, A), (A, C), (C, B), (B, D), (D, C)\}$ Assim sendo, temos que a aresta (B, A) representa um arco que **sai** de A e **entra** em B. Note que o oposto não é verdadeiro neste exemplo, ou seja, $(A, B) \notin E$. Podemos então dizer que B é vizinho de A, mas o contrário não é verdadeiro neste exemplo.

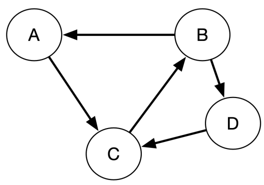


Figura 8: Exemplo de grafo direcionado.

Já com relação aos graus dos nós de um grafo direcionado, existem duas definições: o **grau de saída** d^+ e o **grau de entrada** d^-. O grau de saída $d^+(v)$ de um vértice $v \in V$ é dado pelo número de arestas que **saem** de v, ao passo que o grau de entrada $d^-(v)$ é dado pelo número de arestas que **entram** em v.

Considere o seguinte teorema: para todo grafo orientado $G = (V, E)$, temos que:

$$\sum_{v \in V} d^+(v) = \sum_{v \in V} d^-(v) = |E| \,. \tag{7}$$

A demonstração do teorema acima é bastante intuitiva. Seja o grafo orientado $G = (V, E)$ cujo teorema acima é válido. Seja, agora, um grafo orientado $G' = (V', E')$ de tal forma que $V' = V \cup \{v^\star\}$. O número de arestas de G' é dado pelo número de arestas de G acrescido pelo número de arestas que entram e saem de $v^\star$, ou seja, $|E'| = |E| + d^+(v^\star) + d^-(v^\star)$. Baseando-se no teorema, temos que:

$$\begin{aligned} |E'| &= \underbrace{|E| + d^+(v^\star)} + d^-(v^\star) \\ &= \sum_{v \in V'} d^+(v) \color{red}{- d^-(v^\star)} + d^-(v^\star) \\ &= \sum_{v \in V'} d^+(v). \end{aligned} \tag{8}$$

Na formulação acima, tivemos que adicionar o termo em vermelho pois a somatória dos graus de saída de G' já contabiliza esse montante.

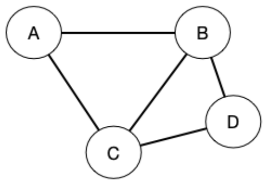
1.2. Representação

A complexidade de algoritmos baseados em grafos é dependente da maneira com a qual eles são representados. Basicamente, existem duas estruturas de dados principais utilizadas para representar grafos: (i) **matriz de adjacência** e (ii) **lista de adjcência**. O seu uso depende das operações que ditam a natureza do problema.

Dado um grafo qualquer $G = (V, E)$, a matriz de adjacência é uma matriz quadrada $A_{|V| \times |V|}$ cujas linhas e colunas são indexadas pelos vértices em G da seguinte forma:

$$A_{ij} = \begin{cases} 1 & \text{caso } (i, j) \in E \\ 0 & \text{caso contrário.} \end{cases} \tag{9}$$

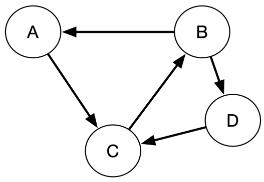
A Figura 9 ilustra um exemplo de um grafo não direcionado e sua respectiva matriz de adjacência. Note que a matriz A é simétrica quando G é não orientado.



	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	0	1
D	0	1	1	0

Figura 9: Exemplo de um grafo e sua respectiva matriz de adjacência.

Já a Figura 10 apresenta um grafo direcionado e sua respectiva matriz de adjacência. No caso, a mesma já não é mais simétrica.



	A	B	C	D
A	0	0	1	0
B	1	0	0	1
C	0	1	0	0
D	0	0	1	0

Figura 10: Exemplo de um grafo direcionado e sua respectiva matriz de adjacência.

Com relação à lista de adjacência, a mesma é construída com base na seguinte premissa: para cada vértice j vizinho de i, adicionamos j na lista ligada $L[i]$, em que L é o apontador para o primeiro elemento desta lista. Desta forma, $L[i]$ armazena todos os vizinhos do vértice i. As Figuras 11 e 12 apresentam exemplos de uma lista de adjacência para um grafo não direcionado e outro direcionado, respectivamente.

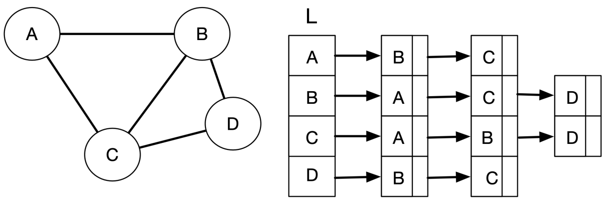


Figura 11: Exemplo de um grafo não direcionado e sua respectiva lista de adjacência.

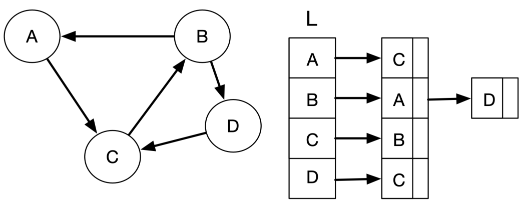


Figura 12: Exemplo de um grafo direcionado e sua respectiva lista de adjacência.

No entanto, qual das duas estruturas de dados devemos utilizar? A resposta à esta pergunta depende muito da **aplicação**. Vejamos alguns exemplos de operações:

- Verificar se existe uma determinada aresta (i, j):
  - Matriz de adjacência: $\theta(1)$ (a matriz já está alocada)
  - Lista de adjacência: $O(|V|)$ (tenho que percorrer toda a lista de adjacência do nó i)
- Complexidade de espaço:
  - Matriz de adjacência: $\theta(|V|^2)$ (geralmente mais adequada a **grafos densos**)
  - Lista de adjacência: $\theta(|V| + |E|)$ (geralmente mais adequada a **grafos esparsos**)

Dado um grafo com $|V|$ vértices, temos que o seu número máximo m de arestas é dado por:

$$m = \binom{|V|}{2} = \frac{|V|\,(|V| - 1)}{2} \in \theta(|V|^2). \tag{10}$$

Um grafo é dito denso se o seu número de arestas se **aproxima** do limitante m acima. Por outro lado, um grafo é dito ser esparso se o seu número de arestas é **muito menor** do que este limitante.

1.3. Aplicações

Uma das principais aplicações em grafos diz respeito à **algoritmos de busca**. Basicamente, temos duas abordagens principais: busca em **largura** e busca em **profundidade**. A seguir, veremos cada uma delas.

1.3.1 Busca em Largura

Seja o nosso grafo $G = (V, E)$. Dizemos que um vértice $v \in V$ é **alcançável** se, a partir de um outro vértice $s \in V$, existe um caminho que conecta s ao vértice v em G. Ademais, temos que a **distância** de s à v é dada pelo caminho mais **curto** se s à v. Caso o vértice v não seja alcançável a partir de s, dizemos então que a distância entre s e v é **infinita**.

A busca em largura é um algoritmo bastante conhecido que objetiva realizar uma varredura no grafo a partir de um nó s denominado **fonte**. O algoritmo percorre todos os vértices em G que são alcançáveis a partir deste nó fonte, construindo uma **árvore de busca em largura** com raiz em s. Cada caminho de s à v nesta árvore corresponde ao caminho mais curto de s à v.

O algoritmo funciona, basicamente, da seguinte forma:

1. Inicialmente, a árvore de busca em largura contém apenas o nó s.
2. Para cada vizinho v de s, o nó v e a aresta (s, v) são inseridos na árvore.
3. O passo acima é repetido para todos os vizinhos de s até que todos os nós alcançados por ele sejam inseridos na árvore.

O algoritmo de busca em largura é implementado utilizado-se uma **fila** Q. Além disso, o algoritmo atribui cores aos nós da seguinte forma:

- <u>branco</u>: significa que o nó ainda não foi visitado.
- <u>cinza</u>: significa que o nó foi visitado uma única vez.

- <u>azul</u>: significa que todos os vizinhos do nó já foram visitados.

Para explicar o funcionamento do algoritmo, tomemos como exemplo o grafo da Figura 12a. Neste caso, tomemos o vértice A como sendo o nó fonte, conforme apresentado na Figura 12b. Neste caso, A é então inserido na fila Q, recebe a cor cinza e o **custo** 0. Além disso, os demais nós recebem o custo ∞. A Figura 11c ilustra o nó A percorrendo seus dois vizinhos, isto é, os nós B e C. Neste exemplo, assumimos que a distância entre nós dois adjacentes é igual à 1. Assim sendo, por enquanto, o menor caminho entre A e B tem custo 1. O mesmo ocorre entre A e C. Em seguida, ambos nós C e B são inseridos na fila e o processo se repete para o próximo elemento (Figura 11d). A Figura 11d ilustra o nó C visitando os seus vizinhos A, B e D. No entanto, como A já saiu da fila, ou seja, possui a cor azul, então o mesmo não pode mais ser modificado. Além disso, o custo do nó D é atualizado para 2, ou seja, o custo que o nó C tinha + 1, e a cor do nó C é mudada para azul. Já a Figura 11g ilustra o nó B saindo da fila e analisando os seus vizinhos A, C e D. Note que nenhum custo será atualizado. A Figura 11h apresenta o nó D analisando seus vizinhos, mas não pode alterar o custo de nenhum deles. Finalmente, a Figura 11i apresenta a árvore de busca resultante, a qual é composta pelos nós e arestas azuis.

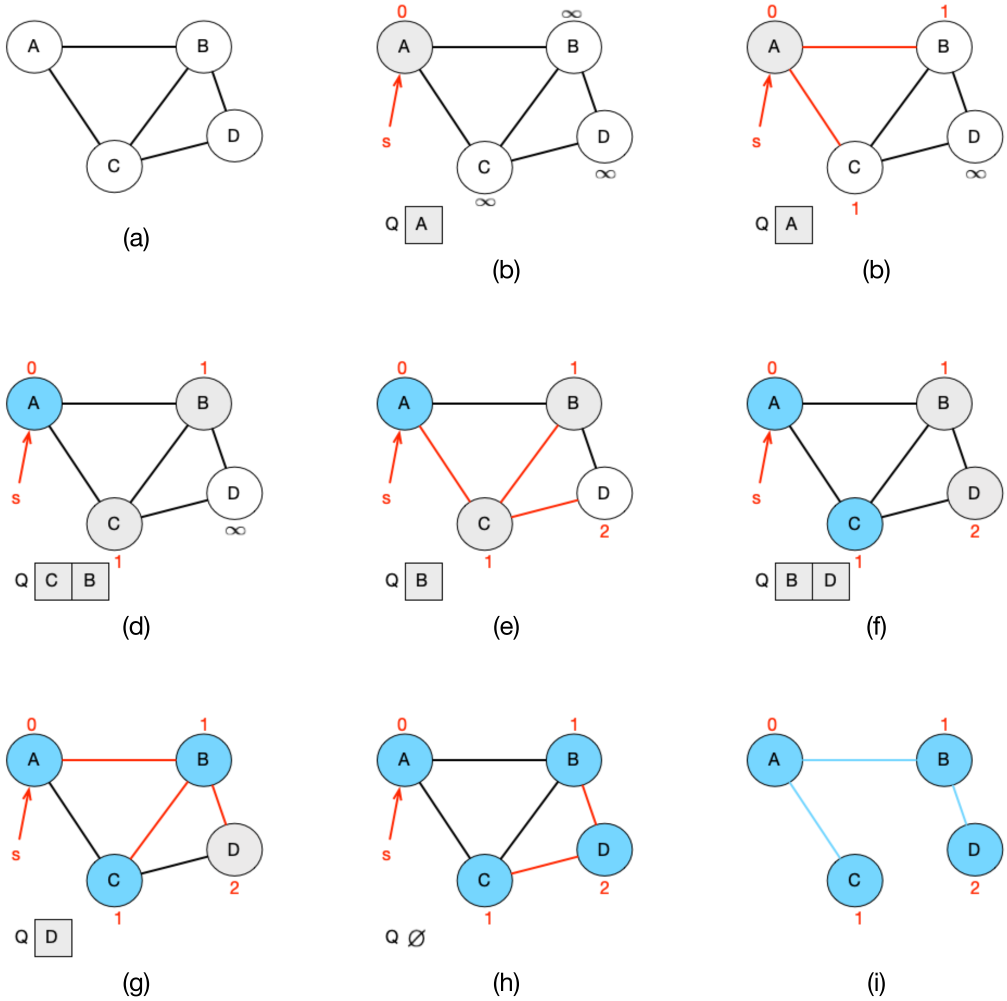


Figura 11: (a) grafo original, (b) busca começa pelo nó fonte A, (c) nó A analisando seus vizinhos, (d) nó A sai da fila, (e) nó C analisando seus vizinhos, (f) nó D entra na fila, (g) nó B analisando seus vizinhos, (h) nó D analisando seus vizinhos e (i) árvore de busca em largura resultante.

O algoritmo abaixo implementa a busca em largura utilizando o grafo da Figura 11a como exemplo, tendo A como sendo o nó fonte.

In [46]:

```
import numpy
import sys
import queue

# definindo cores
BRANCO = 0
CINZA = 1
AZUL = 2

# definindo vizinho vazio
NIL = -1

# definindo custo infinto
infty = sys.maxsize

number_nodes = 4 #define o número de nós (vértices)
cor = numpy.empty(number_nodes).astype('int') # cria vetor de cores
custo = numpy.empty(number_nodes).astype('int') # cria vetor de cus
caminho = numpy.empty(number_nodes).astype('int') # cria vetor de q

Q = queue.Queue(maxsize=number_nodes) #definindo a fila

# assumimos os seguintes identificadores para cada nó para fins de
# A = 0, B = 1, C = 2 e D = 4

# definindo o grafo G
G = {
        0: [1, 2], # B e C são vizinhos de A
        1: [0, 2, 3], # A, C e D são vizinhos de B
        2: [0, 1, 3], # A, B e D são vizinhos de C
        3: [1, 2] # B e C são vizinhos de D
    }

def Busca_Largura(graph, source):

    # inicializa todos os nós de G
    i = 0
    for node in graph:
        cor[i] = BRANCO
        custo[i] = infty
        caminho[i] = NIL

        i = i+1

    # inicializando o nó fonte
    cor[source] = CINZA
    custo[source] = 0
    caminho[source] = NIL

    Q.put(source) # inserindo nó fonte na fila

    while not Q.empty():

        v = Q.get()

        # para cada nó vizinho de v
```

```
        for neighbour in graph[v]:

            if cor[neighbour] == BRANCO:
                cor[neighbour] = CINZA
                custo[neighbour] = custo[v]+1
                caminho[neighbour] = v
                Q.put(neighbour) # insere vizinho na fila
        cor[v] = AZUL

    return caminho

arvore_busca_largura = Busca_Largura(G, 0)
print(arvore_busca_largura)
```

```
[-1  0  0  1]
```

O resultado acima nos mostra um vetor com 4 posições, em que A é o nó fonte (seu predecessor no caminho é -1), o nó B possui como predecessor o nó A, o mesmo ocorre com o nó C. Finalmente, o nó D possui o vértice B como predecessor, conforma apresentado na Figura 11i.

Agora, vamos analisar a complexidade do algoritmo acima. Temos que o laço de inicialização percorre todos os vértices, possuindo complexidade de $\theta(|V|)$. O laço do comando `while` percorre, exatamente, todos os nós, ou seja, também possui complexidade $\theta(|V|)$. Já o comando `if` é verdadeiro apenas uma vez para cada nó pois, quando o mesmo troca de cor, não consegue voltar à cor original branca. Assim sendo, a complexidade do laço `for` acaba sendo $O(|E|)$. Isto significa que cada vizinho será visitado uma única vez, mas nada garante que todas as arestas serão visitadas. As complexidades de inserção e remoção da fila são dadas por $\theta(1)$. Desta forma, a complexidade final da busca em largura é dada por $O(|V| + |E|)$. Note que estamos utilizando uma lista de adjacência para representar o grafo acima.

Este conteúdo foi baseado nas notas de aulas do Prof. Zanoni Dias (IC/Unicamp) disponíveis aqui (https://www.ic.unicamp.br/~zanoni/teaching/mo417/2011-2s/aulas/handout/10-grafos-buscas.pdf).