UniAcademia

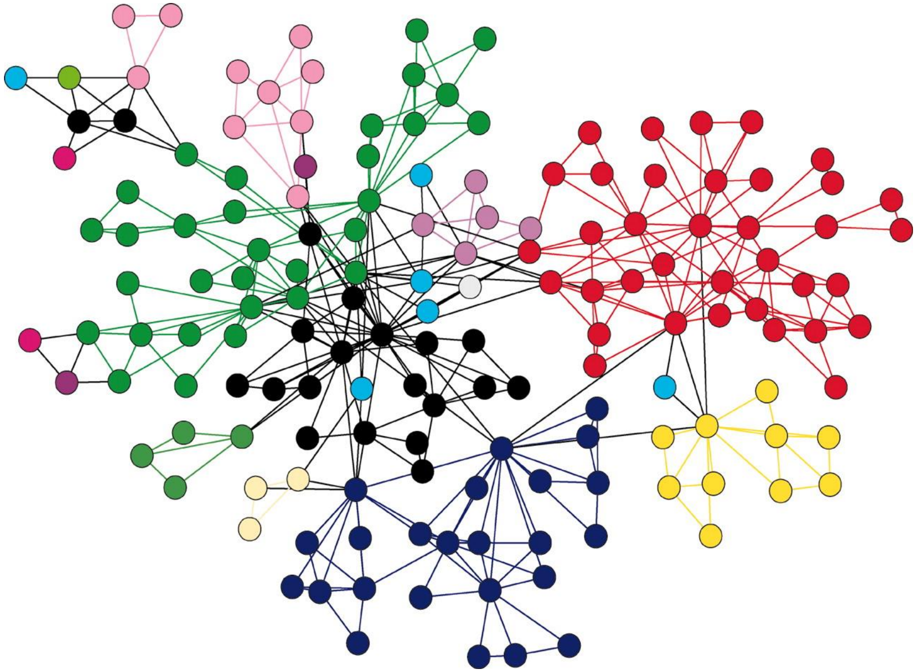
Bachharelado em Engenharia de Software

Bacharelado em Sistemas de Informação

Estrutura de Dados

Teoria dos Grafos

Prof.: Luiz Thadeu Grizendi



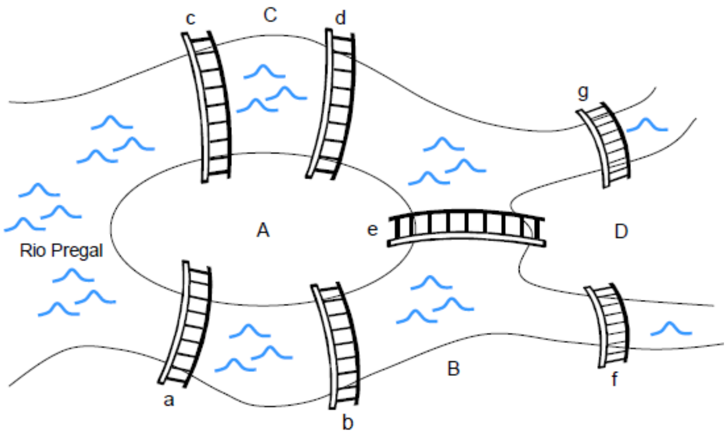
Objetivos

- Aprender o que é um grafo e como ele é usado.
- Implementar o tipo abstrato de dados grafo usando várias representações internas.
- Utilização dos grafos para resolver uma ampla variedade de problemas

Histórico

- A primeira evidência sobre **grafos** (*graphs*) remonta a 1736, quando Euler fez uso deles para solucionar o problema clássico das pontes de Koenigsberg

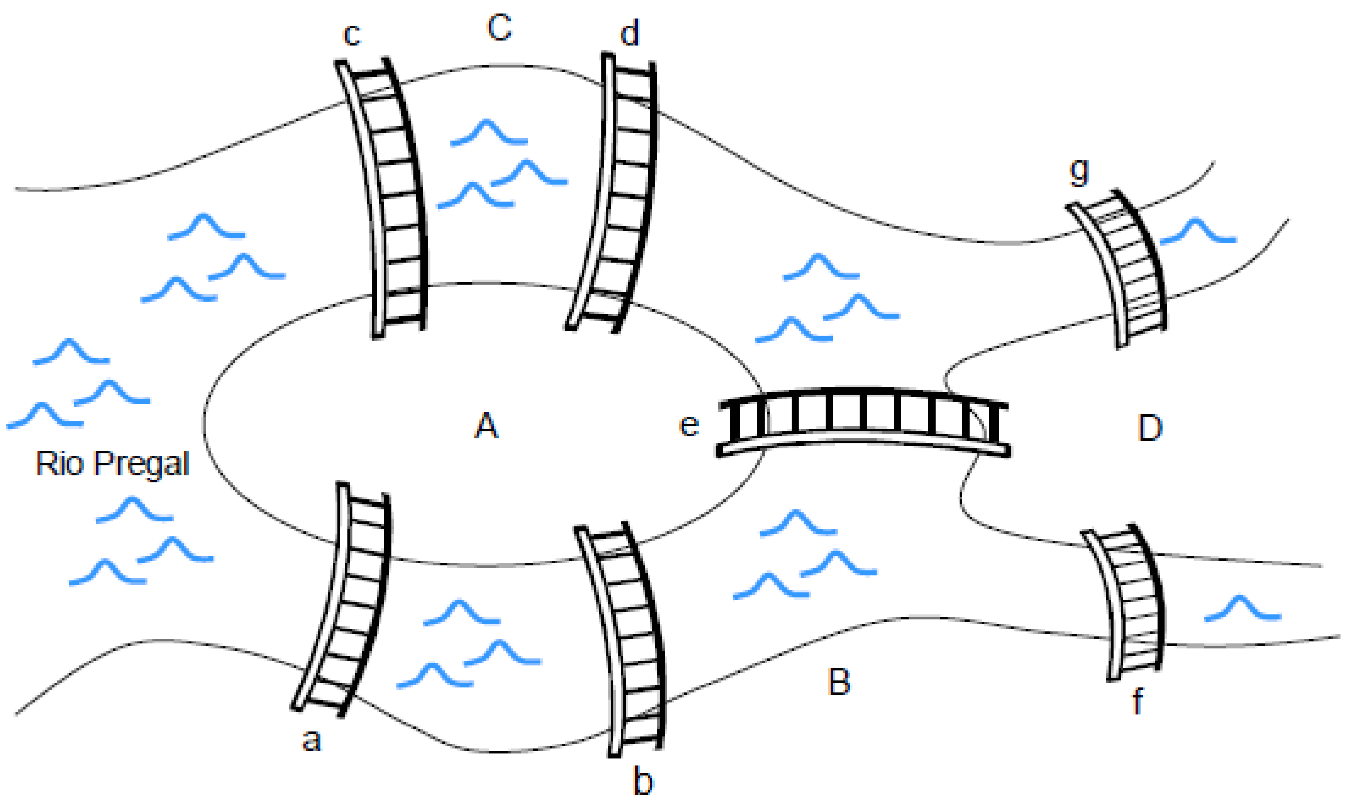
- Na cidade de Koenigsberg (na Prússia Oriental), o rio Pregal flui em torno da ilha de Kneiphof, dividindo-se em seguida em duas partes . Assim sendo, existem quatro áreas de terra que ladeiam o rio: as áreas de terra (A-D) estão interligadas por sete pontes (a-g)



- O problema das pontes de Koenigsberg consiste em se determinar se, ao partir de alguma área de terra, é possível atravessar todos os pontos exatamente uma vez, para, em seguida, retornar à área de terra inicial

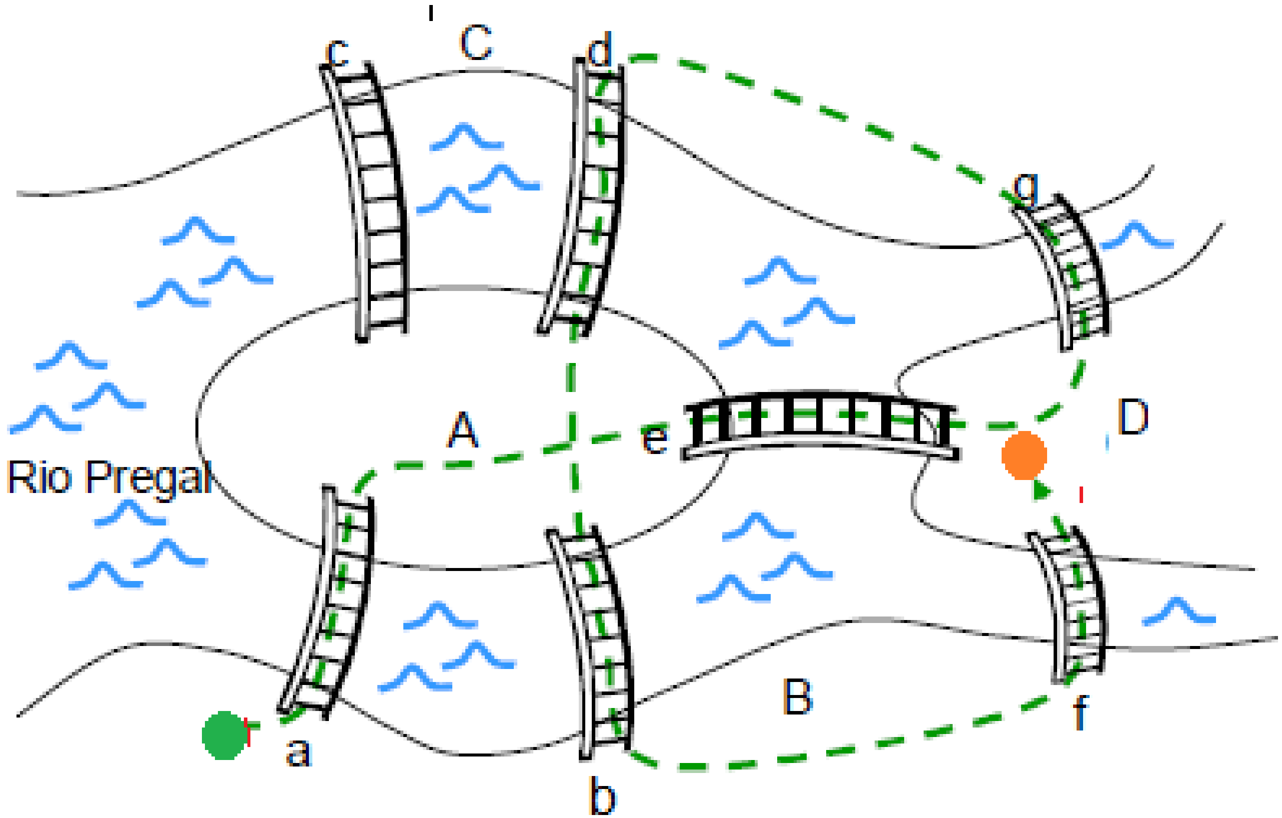
Histórico

- É possível caminhar sobre cada ponte exatamente uma única vez e retornar ao ponto de origem?



Histórico

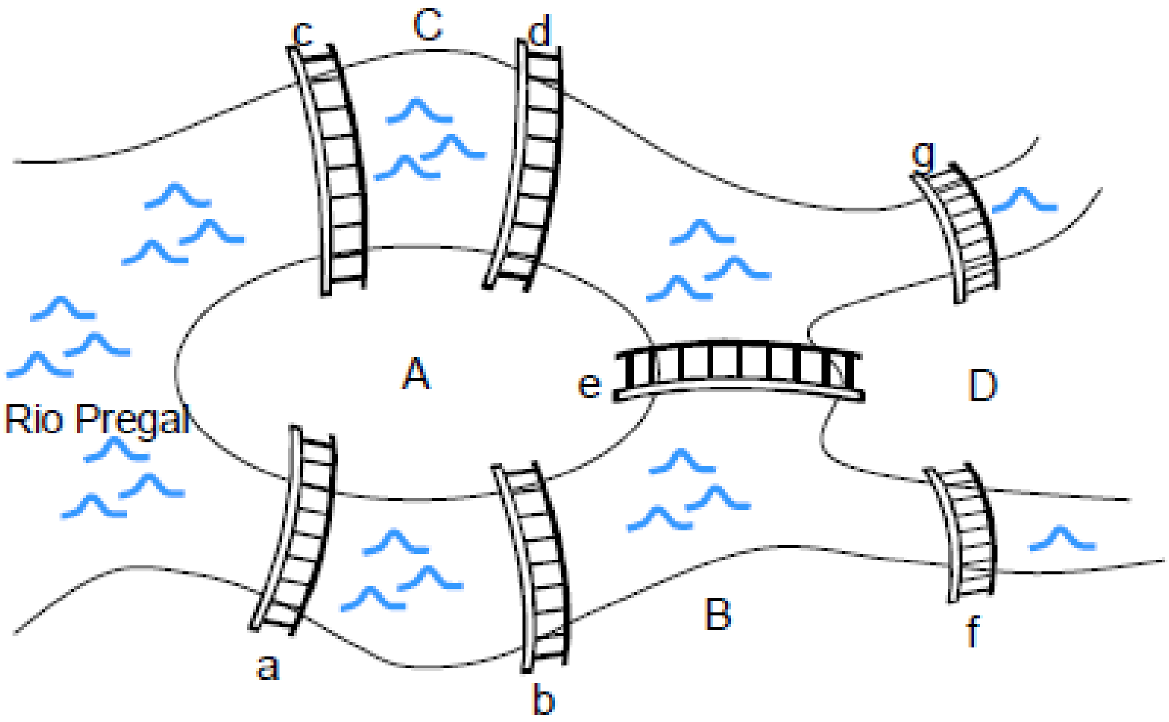
- Um caminho possível consistiria em iniciar na área de terra **B**, atravessar a ponte **a** para a ilha **A**; pegar a ponte **e** para chegar à área **D**, atravessar a ponte **g**, chegando a **C**; cruzar a ponte **d** até **A**; cruzar a ponte **b** até **B** e a ponte **f**, chegando a **D**

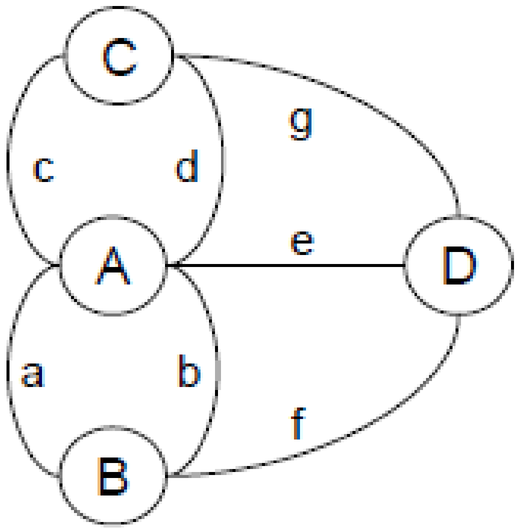


Observe que este caminho NÃO passa por todas as pontes!!!!

Histórico

- Euler provou que não é possível o povo de Koenigsberg atravessar cada ponte exatamente uma vez, retornando ao ponto inicial
- Ele resolveu o problema, representando as áreas de terra como vértices e as pontes como arestas de um grafo (na realidade, um multigrafo)





Introdução

- As estruturas de dados Vetor, Lista, Fila e Pilha, são estruturas de dados unidimensionais ou lineares, e existem aplicações que necessitamos de outras estruturas mais complexas, como por exemplo, estruturas do tipo hierárquica, isto é, um determinado nó possui, um ou mais nós hierarquicamente inferiores.

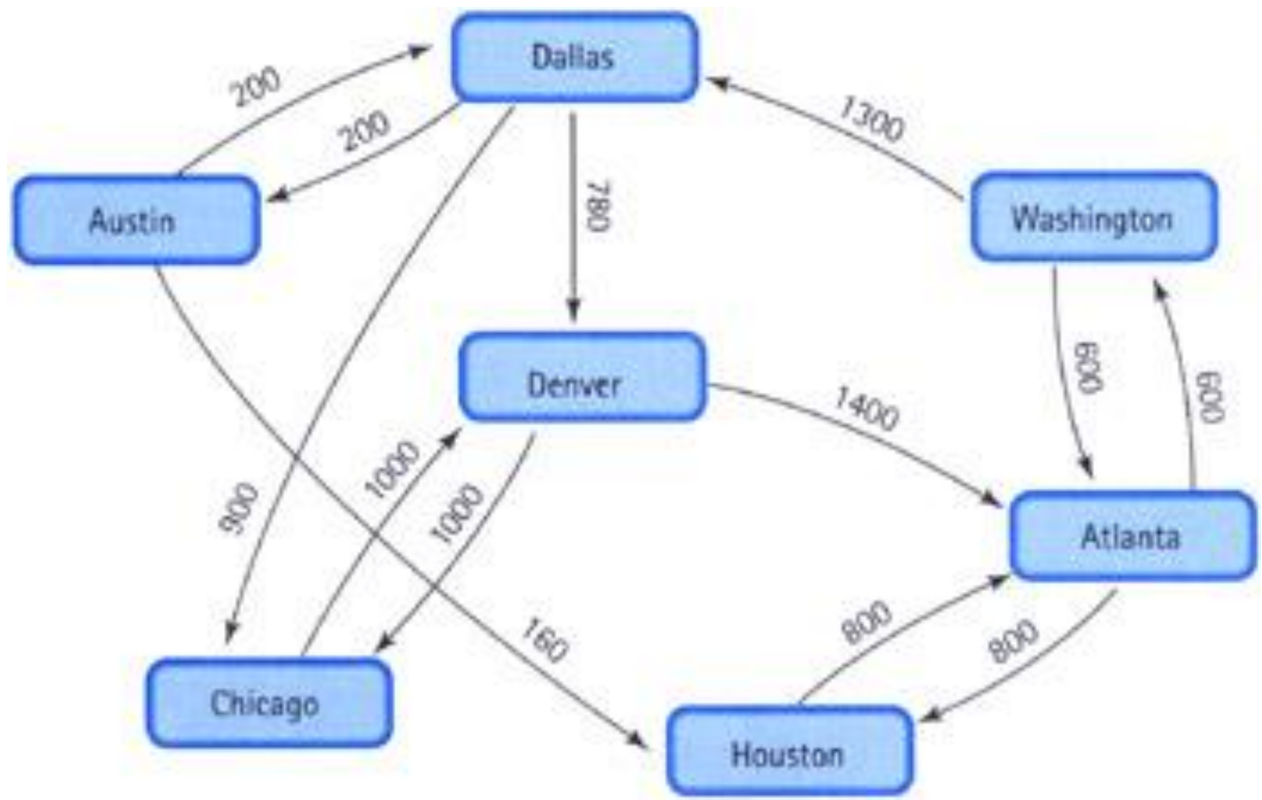
- Podemos destacar vários exemplos de aplicações de estruturas hierárquicas bem como: as pastas, subpastas e documentos de um computador, índice remissivo de um determinado livro, a árvore filogenética da vida etc.

- Os grafos são estruturas de dados mais geral que as árvores ou podemos pensar que as árvores são estruturas especiais de grafos.

- Os grafos podem ser usados para representar muitas coisas interessantes sobre o nosso mundo, incluindo sistemas rodoviários, voos aéreos de cidade em cidade, como a Internet está conectada ou mesmo a sequência de aulas que você deve fazer para concluir um curso de ciência da computação.

A figura a seguir ilustra um grafo com as distâncias entre algumas cidades americanas.

Repare que necessitamos conhecer os nós dos grafos e os elos de ligação entre esses nós.



Primeiras noções

Numa escola algumas turmas resolveram realizar um torneio de vôlei. Participam do torneio as turmas 6*A*, 6*B*, 7*A*, 7*B*, 8*A* e 8*B*. Alguns jogos foram realizados até agora:

6*A* jogou com 7*A*, 7*B*, 8*B*
6*B* jogou com 7*A*, 8*A*, 8*B*
7*A* jogou com 6*A*, 6*B*
7*B* jogou com 6*A*, 8*A*, 8*B*
8*A* jogou com 6*B*, 7*B*, 8*B*
8*B* jogou com 6*A*, 6*B*, 7*B*, 8*A*

Mas será que isto está correto ? Pode ter havido um erro na listagem?

Primeiras noções

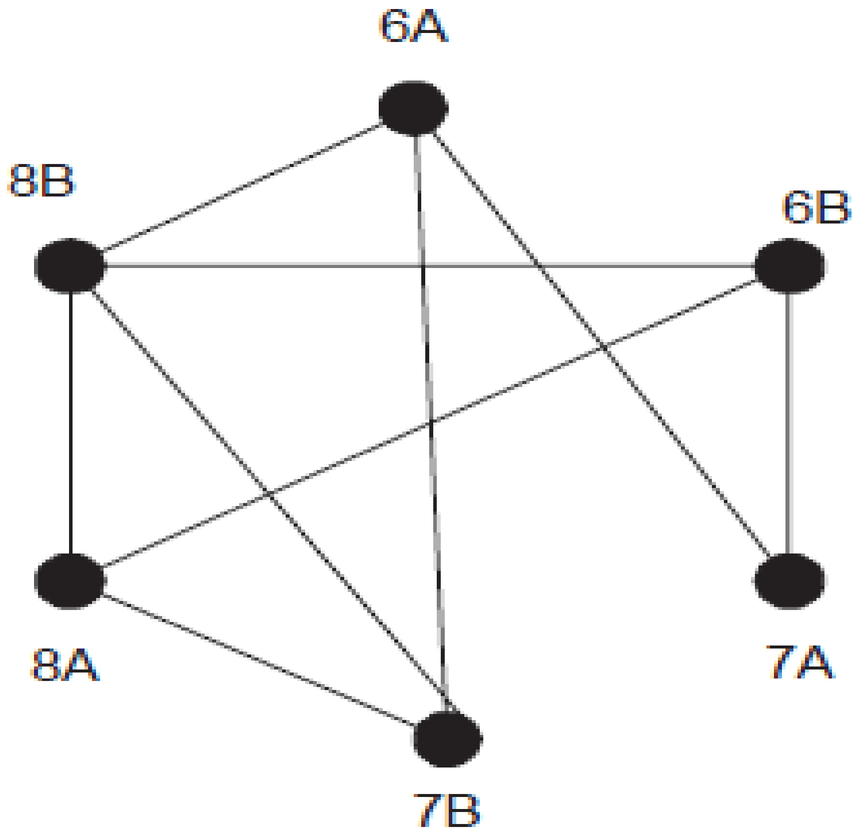
Uma maneira de representar a situação através de uma figura as turmas serão representadas por pontos e os jogos serão representados por linhas.

Não ´é difícil agora constatar a consistência das informações. A estrutura que acabamos de conhecer é um **grafo**.

Apresentamos duas formas de representar esta estrutura

- Por uma lista, dizendo quem se relaciona com quem.
- Por um desenho, isto é, uma representação gráfica do problema.

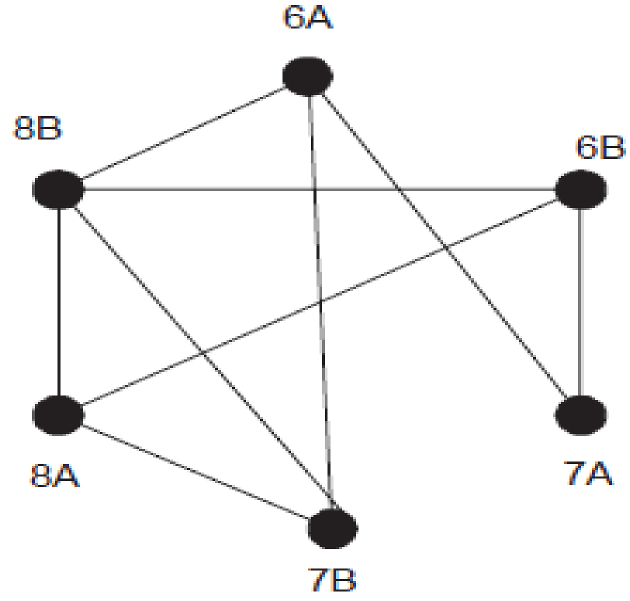
representação gráfica do problema.



Para que um grafo fique bem definido temos que ter dois conjuntos:

- O conjunto *V* , dos **vértices** - no nosso exemplo é o conjunto das turmas.
- O conjunto *A*, das **arestas** - no nosso exemplo, os jogos realizados.

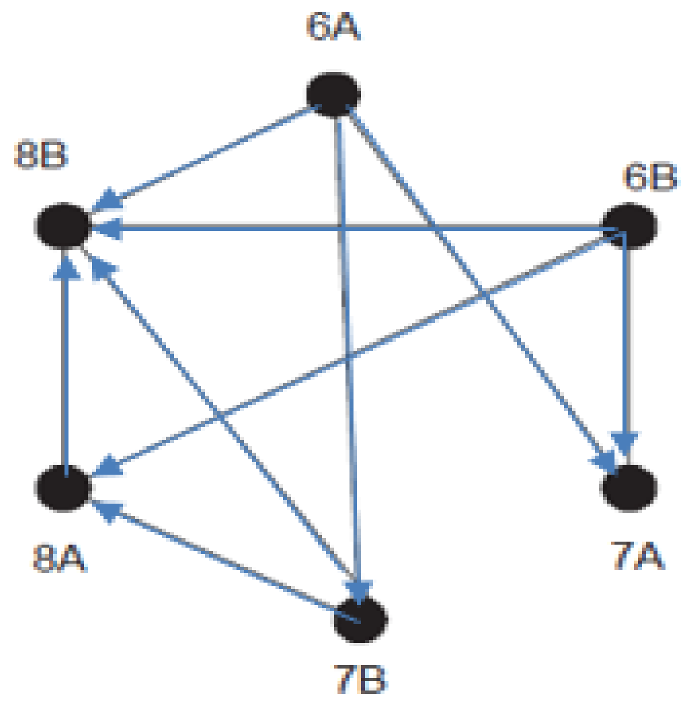
Quando existe uma aresta ligando dois vértices dizemos que os vértices são **adjacentes** e que a aresta é **incidente** aos vértices.



No nosso exemplo podemos representar o grafo de forma sucinta como:

V = *{* 6*A*; 6*B*; 7*A*; 7*B*; 8*A*; 8*B* *}*
A = *{* (6*A*; 7*A*); (6*A*; 7*B*); (6*A*; 8*B*); (6*B*; 7*A*); (6*B*; 8*A*); (6*B*; 8*B*); (7*B*; 8*A*);(7*B*; 8*B*); (8*A*; 8*B*)*}*

Mas e se o campeonato tivesse turno e returno?! Aí o nosso grafo teria que ser representado graficamente da seguinte forma



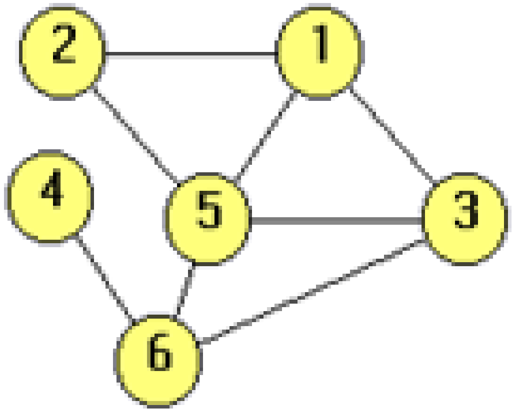
I

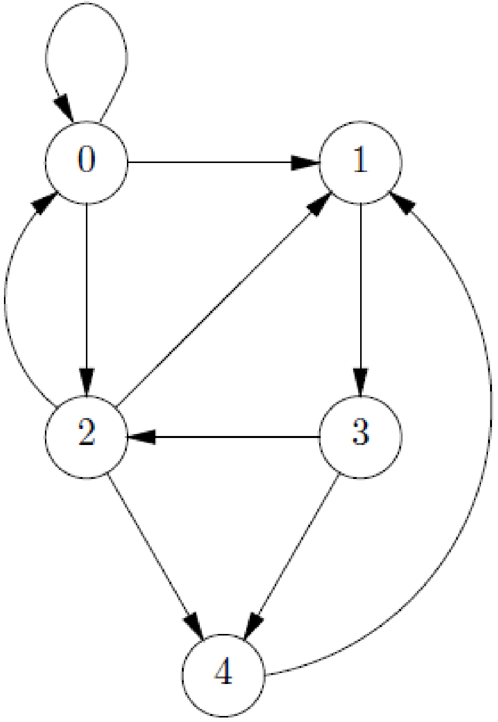
Observe que nesse caso, a direção da seta (arco ou aresta) é fundamental para entendermos qual vértice liga a outro vértice. Esse caso é conhecido como grafo dirigido ou dígrafo.

Grafos

- Grafo é uma estrutura matemática que consiste em um conjunto nós denominados de vértices e um conjunto de linhas que ligam estes nós.

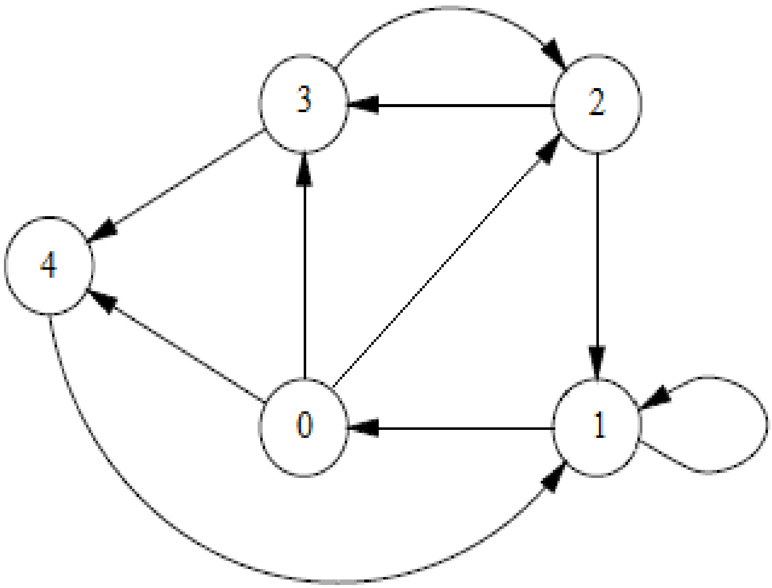
- Estas linhas são denominadas de arestas (edges) no caso de linhas não orientadas ou, de setas, denominadas de arcos (arcs), no caso de linhas orientadas.





Grafo Dirigido (Dígrafo)

- Um Grafo Dirigido ou Dígrafo(Directed Graph - diGraph) consiste de um conjunto de nós (vértices) V e um conjunto de arcos que são as setas que ligam dois nós quaisquer do conjunto V.
- É importante frisarmos que o conjunto de arcos { (u,v) } indica que u é o vértice origem e v o vértice extremidade do arco.



$$V = \{\, 0,1,2,3,4 \,\}$$

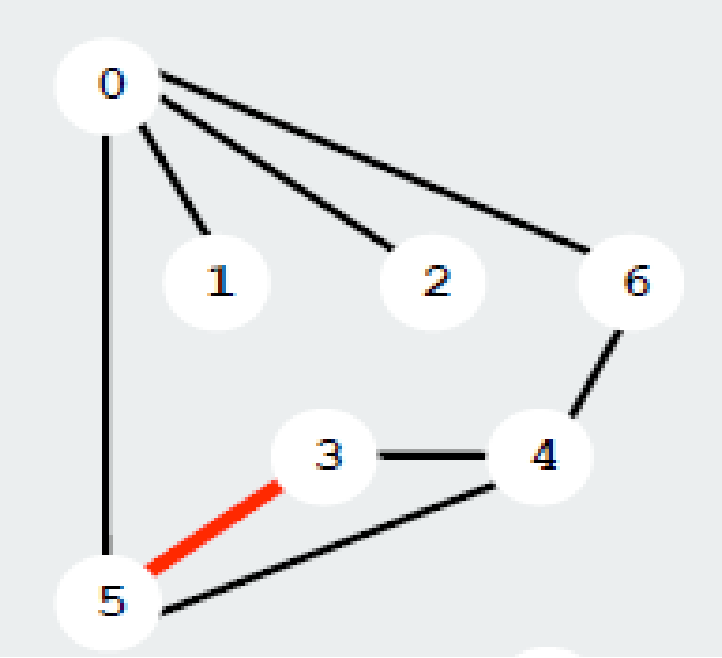
$$A = \{\, (0,4), (0,2), (0,3), (1,0), (1,1), (2,1), (2,3), (3,2), (3,4), (4,1) \,\}$$

Grafo Não Dirigido

- Um Grafo não Dirigido (UnDirected Graph) consiste de um conjunto de nós (vértices) V e um conjunto de arestas que são as linhas que ligam dois nós quaisquer do conjunto V.

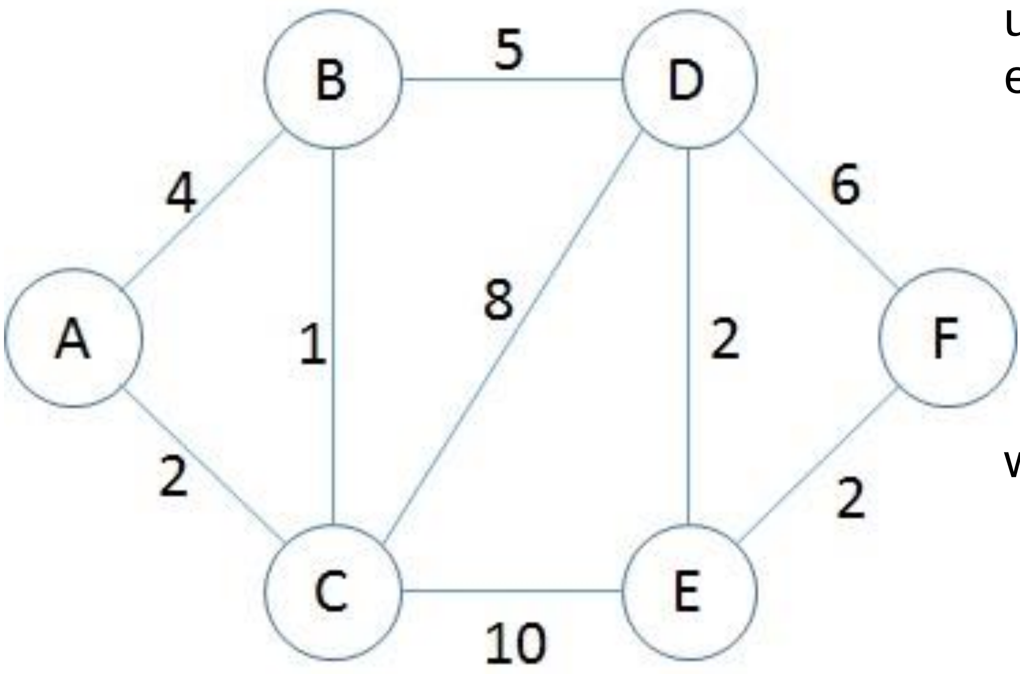
- É importante frisarmos no grafo não dirigido, se existir uma aresta (u,v) obrigatoriamente terenos a aresta (v, u).

$$V = \{ 0,1,2,3,4,5,6 \}$$
$$A = \{(0,1), (0,2), (0,5), (0,6), (1,0), (2,0), (3,4), (3,5), (4,3), (4,5), (4,6),$$
$$(5,0), (5,3), (5,4), (6,0), (6,4)\}$$



Grafo Valorado

Um **grafo valorado**, é um tipo de grafo cujas arestas possuem um peso (weight). Por exemplo considere o grafo a seguir.



O peso (weight) de uma aresta é um valor alocado à aresta. Por exemplo no grafo anterior, temos

$$\text{weight}(A, B) = 4$$
$$\text{weight}(C, D) = 8$$

Genericamente

$$\text{weight}(v\acute{e}rice\ u, v\acute{e}rice\ v) = valor$$

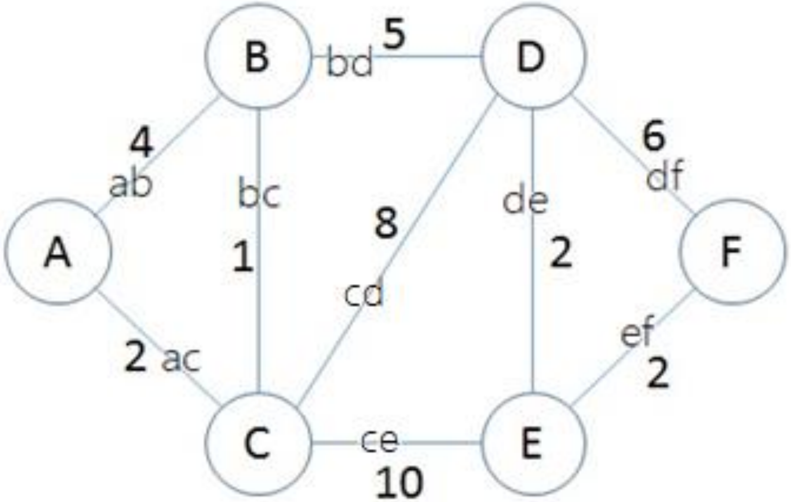
Aplicações

- Análise de circuitos elétricos
- Verificação de caminhos mais curtos
- Análise de planejamento de projetos (*scheduling*)
- Identificação de compostos químicos
- Genética
- Cibernética
- Linguística
- Ciências Sociais, etc

Pode-se afirmar que de todas as estrutura matemáticas, grafos são as que se encontram em uso mais amplo

Terminologia, conceitos e características de um Grafo

Considere o grafo não dirigido a seguir

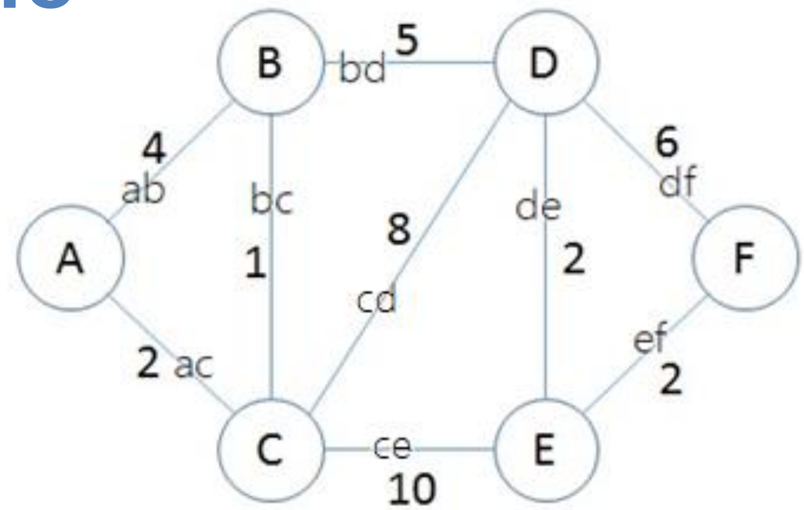


O **rótulo** label) de uma aresta é um identificador da aresta. Por exemplo no grafo anterior podemos determinar alguns rótulos de arestas:

$$label\ (A, B) = ab$$
$$label\ (A, C) = ac$$

opposite

Terminologia, conceitos e características de um Grafo



O vértice **oposto (opposite)** à um vértice u, considerando que o rótulo (label) de uma determinada aresta é seja l, é um vértice v, tal $label\ (u, v) = l$. Na figura anterior podemos obter alguns rótulos de determinados arcos

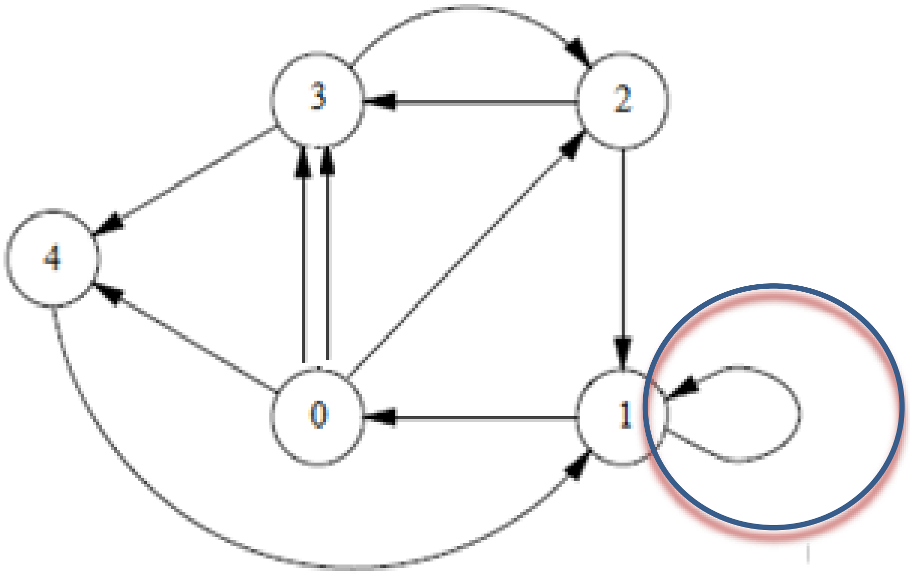
$$opposite(A, ab) = B$$
$$opposite(C, cd) = D$$

Genericamente

$$opposite(vétice\ origem, label\ da\ aresta) = vértice\ chegada$$

Terminologia, conceitos e características de um Grafo

Considere agora o grafo dirigido a seguir

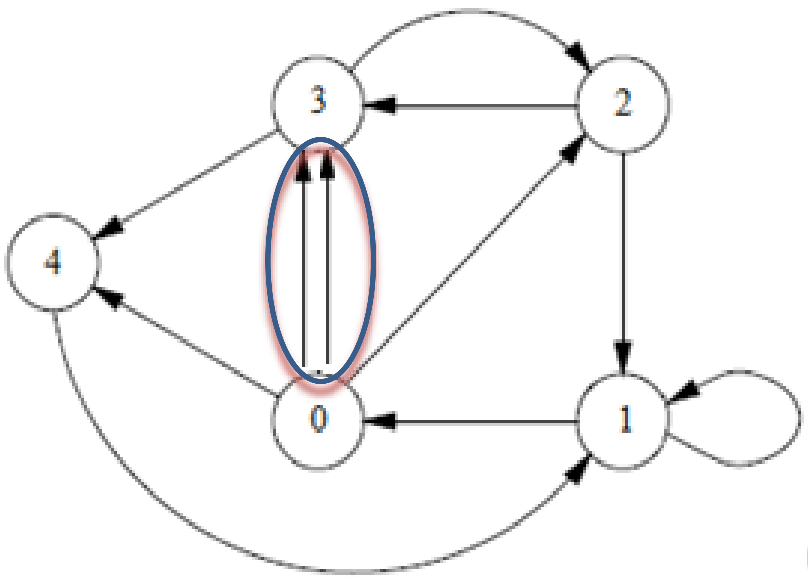


Dizemos que um determinado grafo possui um ou mais laços, quando existir pelo menos um vértice que tem ligação com ele mesmo.

Na figura anterior temos um laço que ocorre no vértice 1.

Terminologia, conceitos e características de um Grafo

Considere agora o grafo dirigido a seguir

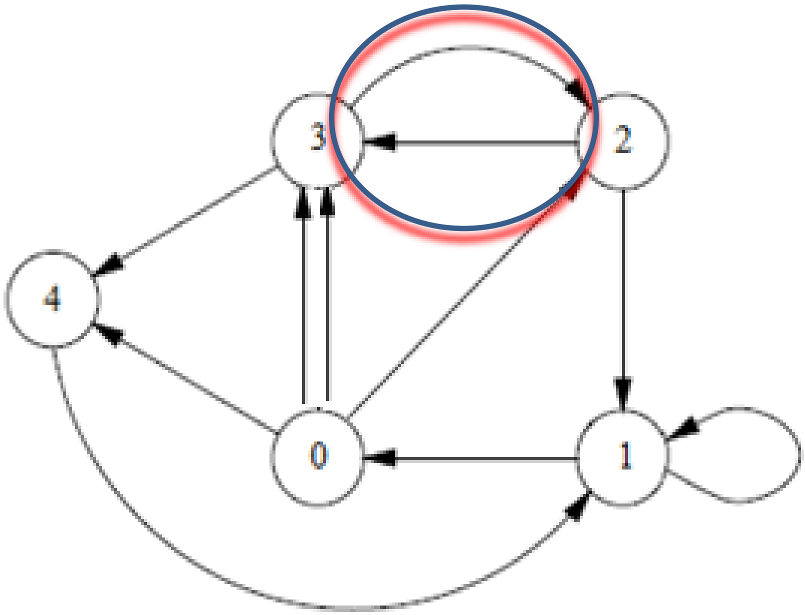


Dizemos que um determinado grafo possui um duas ou mais arestas paralelas quando existir pelo menos um par de vértices (u,v) que possui mais de uma aresta ligando esses vértices.

Na figura anterior temos as arestas paralelas entre os vértices 0 e 3.

Terminologia, conceitos e características de um Grafo

Considere agora o grafo dirigido a seguir

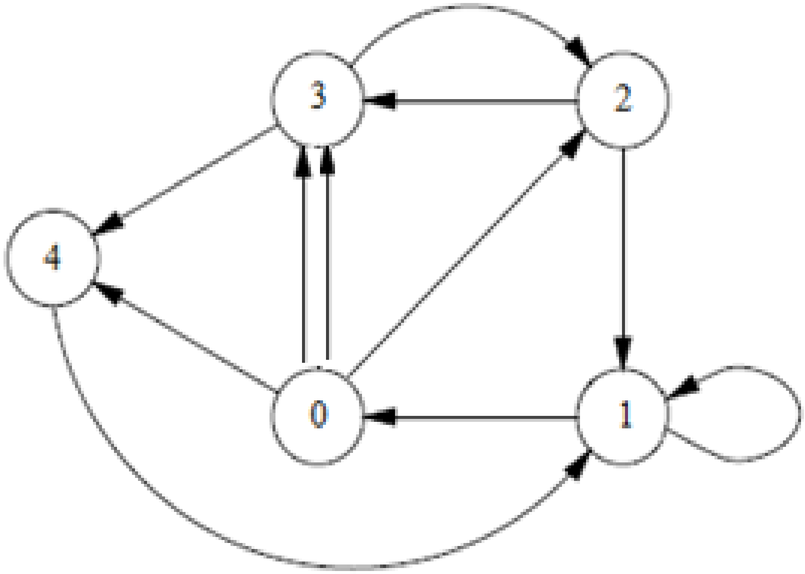


Dizemos que um determinado grafo possui um duas ou mais arestas antiparalelas, quando existir pelo menos um par de vértices (u,v) que possui pelo menos uma pelo menos uma aresta ligando u a v, e pelo menos uma aresta ligando v e u. Nesse caso o grafo tem que ser dirigido.

Na figura anterior temos as arestas antiparalelas entre os vértices 2 e 3.

Terminologia, conceitos e características de um Grafo

Considere agora o grafo dirigido a seguir



Dois vértices u e v, são ditos adjacentes (adjacent), quando existir, pelo menos, uma aresta que liga esses vértices.

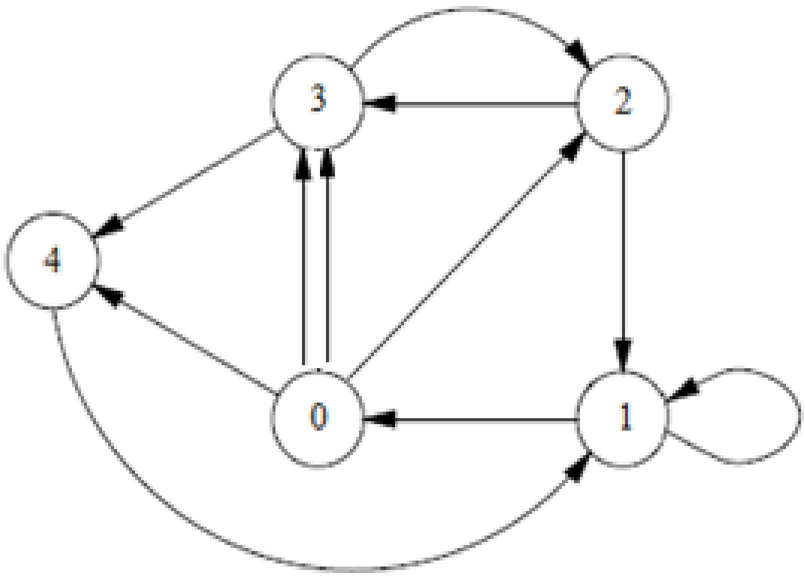
Na figura anterior podemos verificar alguns vértices se são ou não adjacentes

$adjacent(0,4) \quad Verdadeiro$

$adjacent(4,0) \quad Falso$ (repare que o grafo é dirigido)

Terminologia, conceitos e características de um Grafo

Considere agora o grafo dirigido a seguir



O conjunto de vértices adjacentes à um determinado vértice é denominado de sucessores (successors) Na figura anterior podemos obter sucessores de alguns vértices:

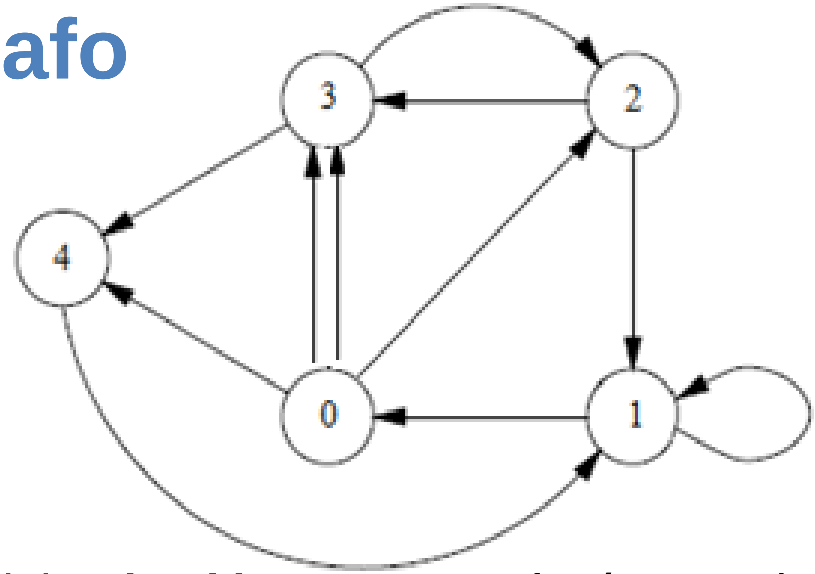
$$sucsessors(0) = \{2,3,4\}$$

$$successors(1) = \{0,1\}$$

Genericamente,

$$\text{successors}(vétice\ origem) = \{\ vértices\ extremidade\}$$

Terminologia, conceitos e características de um Grafo



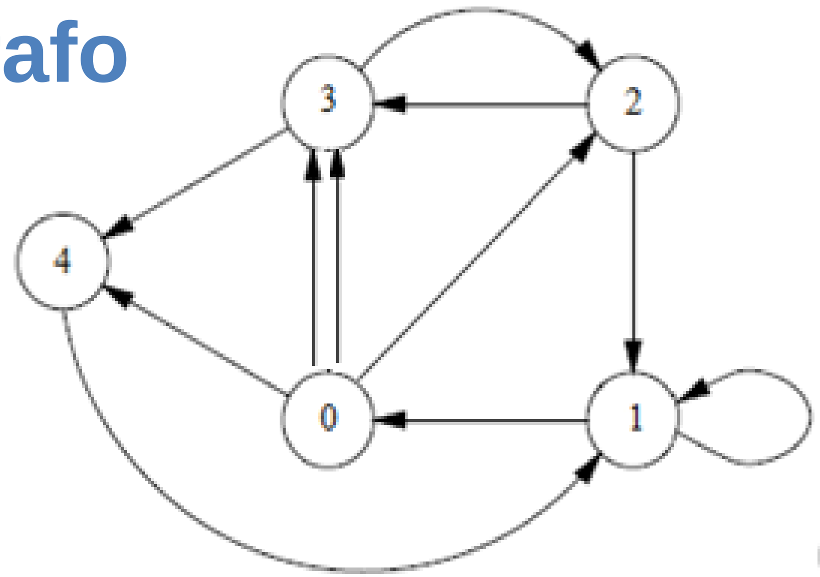
Um caminho **(path)** em um grafo é uma determina sequência de vértices adjacentes que começam em um vértice origem e terminam em um vértice destino. Na figura anterior podemos obter caminhos entre alguns vértices:

$$path(0{,}1) = \{0{,}4{,}1\}$$
$$path(1{,}3) = \{1{,}0{,}3\}$$
$$path(1{,}3) = \{1{,}1{,}0{,}3\}$$ Observe que nesse caminho temos um laço

Genericamente,

$$path(vétice\ origem, vértice\ destino) = \{\ vértice\ origem, \ldots\ldots vértice\ destino\}$$

Terminologia, conceitos e características de um Grafo



Um ciclo $(\boldsymbol{cycle})$ em um grafo é uma determino caminho que começa e termina em um mesmo vértice. Na figura anterior podemos obter ciclos entre alguns vértices:

$$cycle = \{0{,}4{,}1, 0\}$$

$$cycle = \{1{,}0{,}3{,}2{,}1\}$$

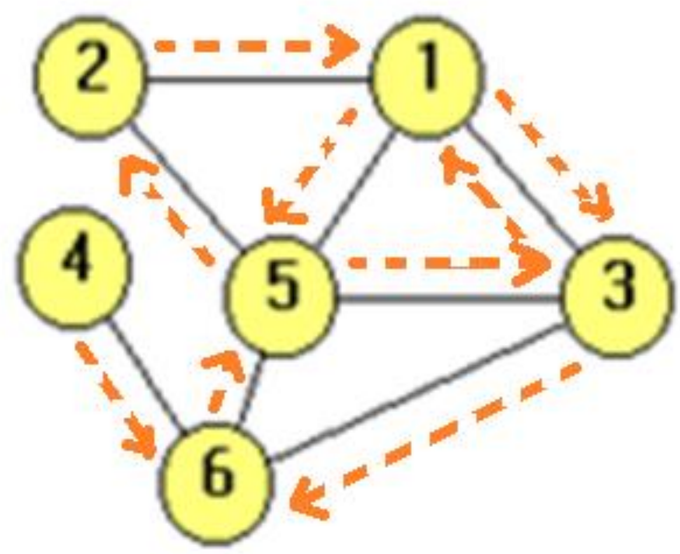
$cycle = \{1\}$ Observe que nesse ciclo temos um laço

Genericamente,

$$cycle = \{\, vértice\ origem, \ldots\ldots vértice\ origem\}$$

Terminologia, conceitos e características de um Grafo

Considere agora o grafo não dirigido a seguir

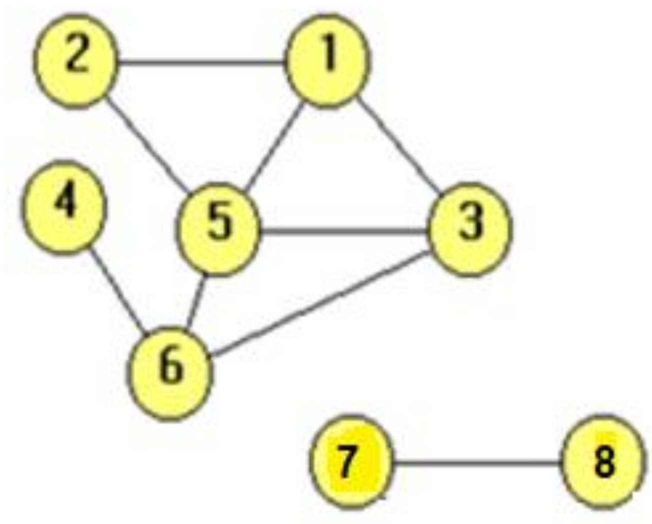


Um grafo é dito conexo **(connected)** quando existir pelo menos um caminho que liga um determinado vértice à todos os outro vértices.

Na figura anterior podemos observar que sempre existirá um caminho que passa por todos os vértices:

Terminologia, conceitos e características de um Grafo

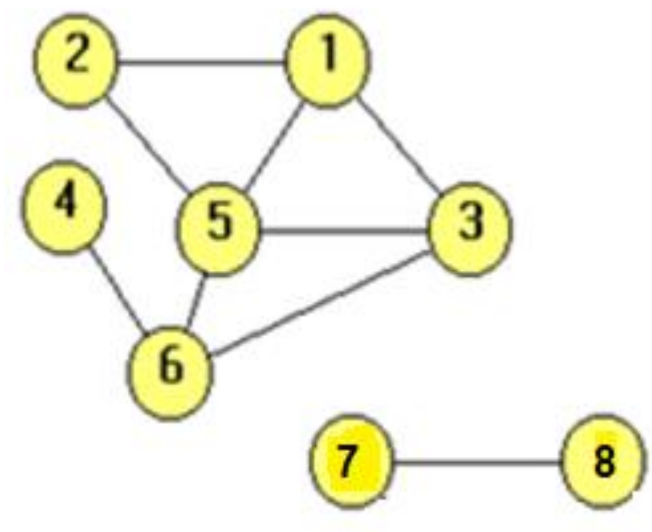
Considere agora o grafo não dirigido a seguir



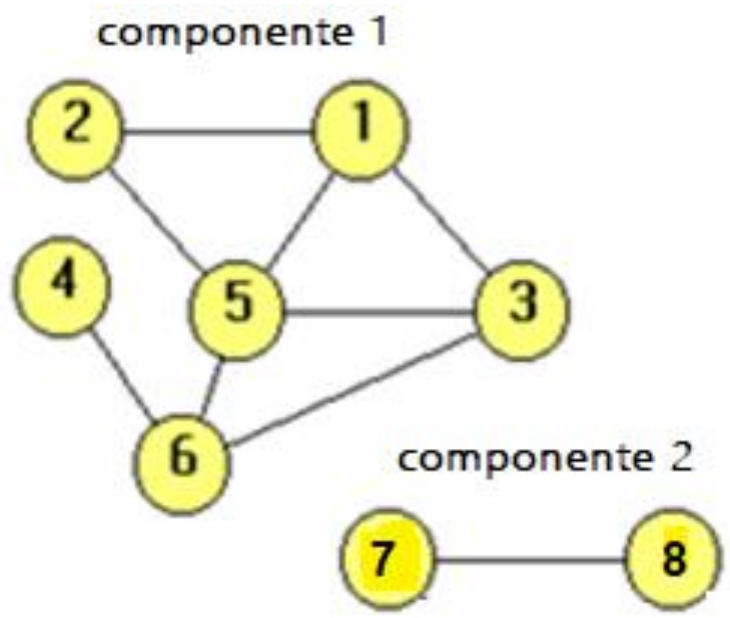
Observe que no grafo desconexo (unconnected) da figura anterior, temos duas partes, que são grafos conexos denominado de componentes (componentes)

Terminologia, conceitos e características de um Grafo

Considere agora o grafo não dirigido a seguir

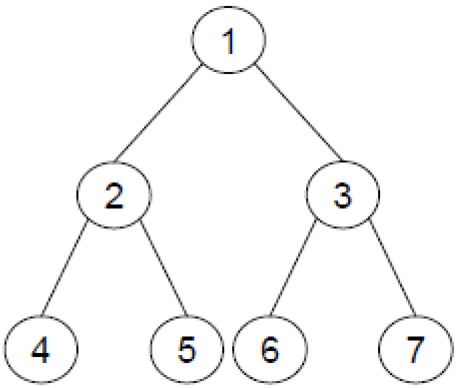


Observe que no grafo desconexo (unconnected) da figura anterior, temos duas partes, que são grafos conexos denominado de componentes (componentes)

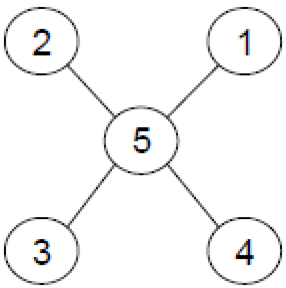


Terminologia, conceitos e características de um Grafo

Dizemos que um grafo é uma árvore quando ele é conexo e sem ciclos



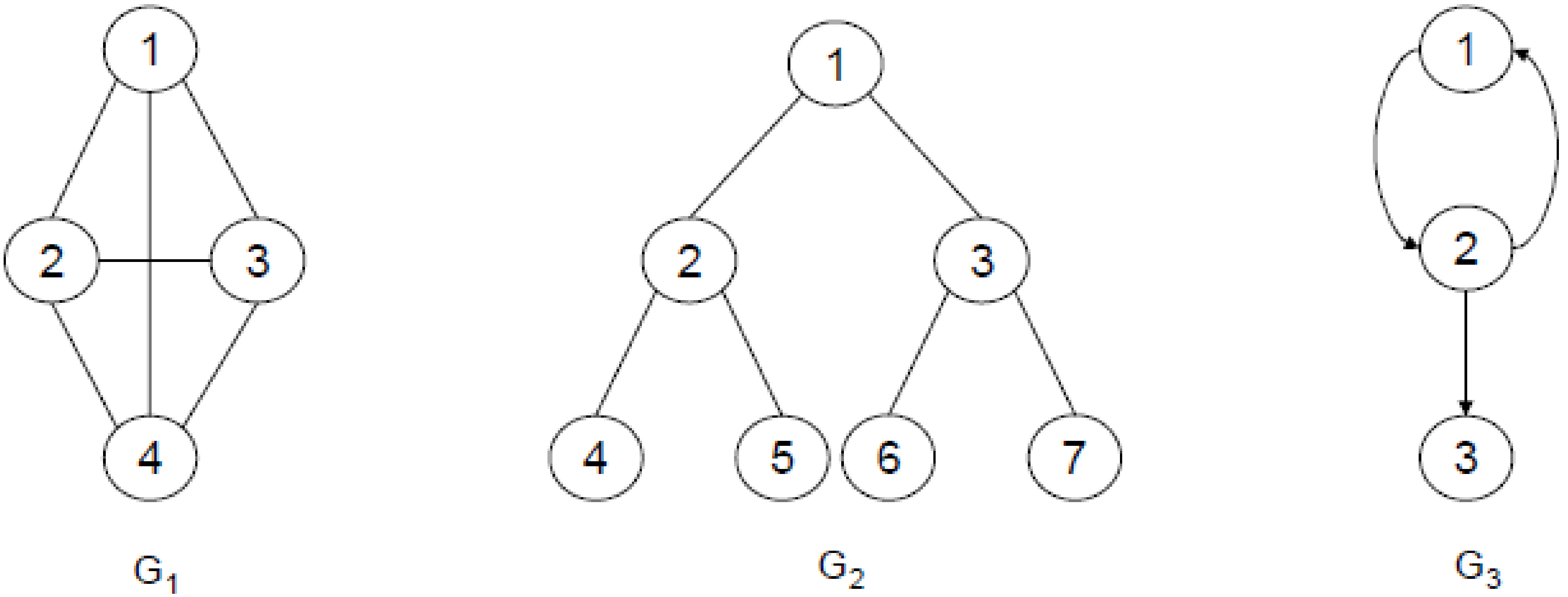
Árvore$_1$



Árvore$_2$

Terminologia, conceitos e características de um Grafo

O grafo G2 é uma árvore, ao passo que os grafos G1 e G3 não são



Terminologia, conceitos e características de um Grafo

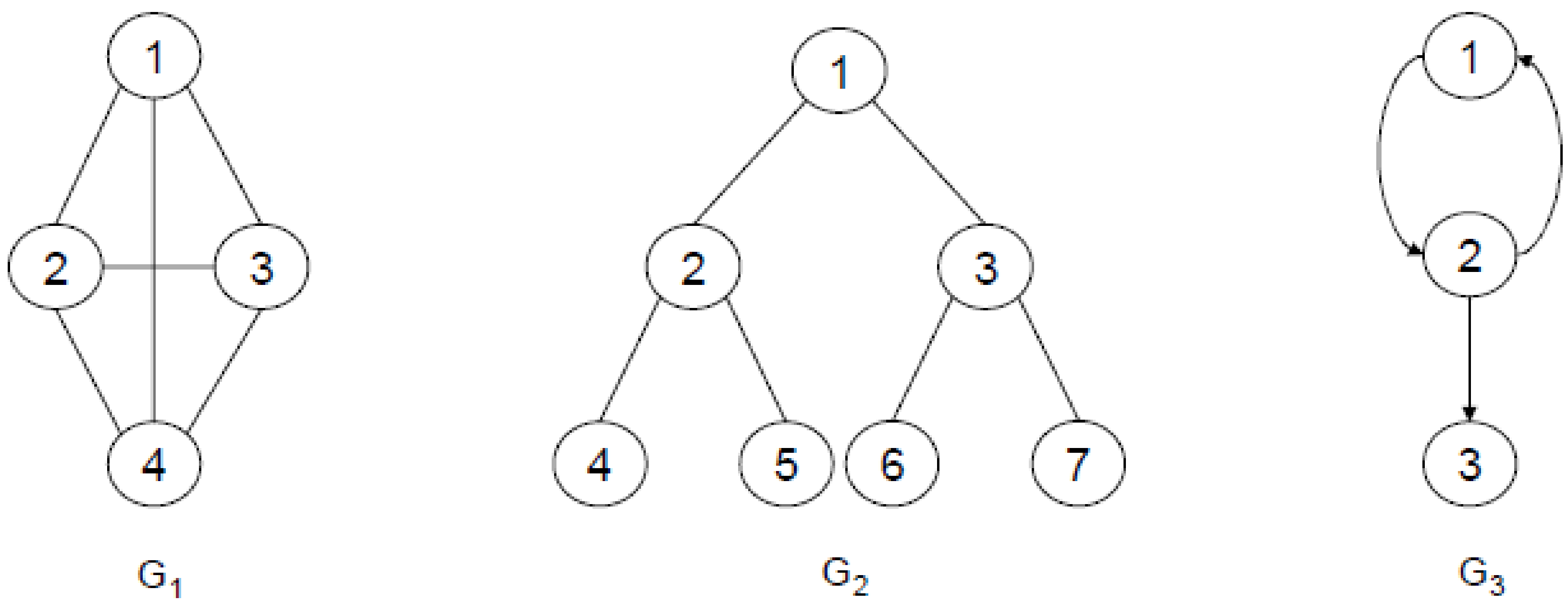
Grau

- O **grau** de um vértice **v,** escrito como **grau(v)**, é o número de arestas incidentes no vértice **v**
- Caso G seja um grafo orientado:
  - o **grau de entrada** de um vértice **v** é definido como sendo o número de arestas para as quais **v** seja o término
  - o **grau de saída** é o número de arestas para as quais **v** é o início
- Geralmente, quando um grafo é orientado, dizemos dígrafo e, para o grafo não-orientado referenciamos simplesmente de grafo

Terminologia, conceitos e características de um Grafo

Grau

- O grau de vértice 1 em G1 é 3
- O vértice 2 de G3 tem grau de entrada igual a 1, grau de saída igual a 2 e grau igual a 3
- Repare que o grau de todos os vértices do grafo G2, que é uma árvore, é, no máximo igual a dois, logo, dizemos que a árvore é binária

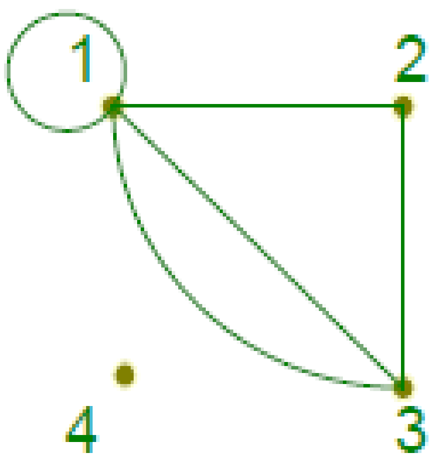


Representação interna dos grafos

Os grafos podem ser estruturados de várias formas. As mais conhecidas são na forma de uma matriz, denominada de matriz de adjacência e, na forma de listas, denominado de lista de adjacência.

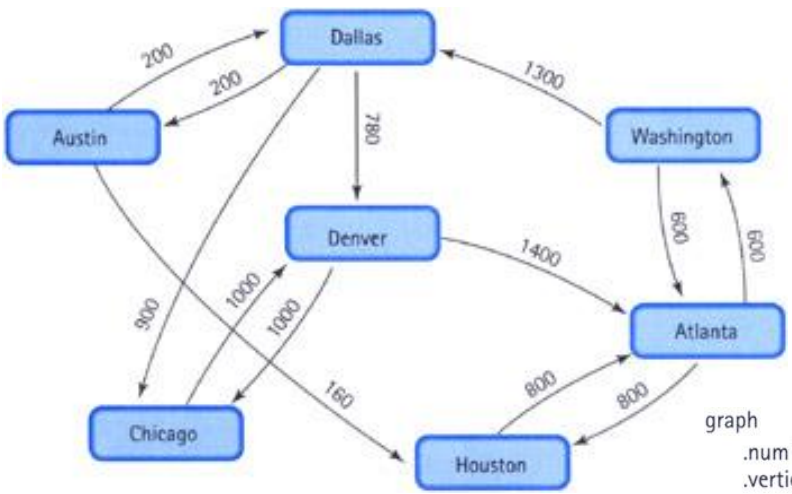
Matriz de adjacências:

Seja G um grafo com o conjunto de vértices $\{v_1, v_2, ..., v_n\}$.
A matriz de adjacências de G é uma matriz $n \times n$, $A = A(G)$ tal que a_{ij} é o número de arestas distintas que ligam v_i a v_j.



$$A = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Matriz de adjacências



graph

.num Vertices 7

.vertices

[0]	"Atlanta "
[1]	"Austin "
[2]	"Chicago "
[3]	"Dallas "
[4]	"Denver "
[5]	"Houston "
[6]	"Washington"
[7]	
[8]	
[9]	

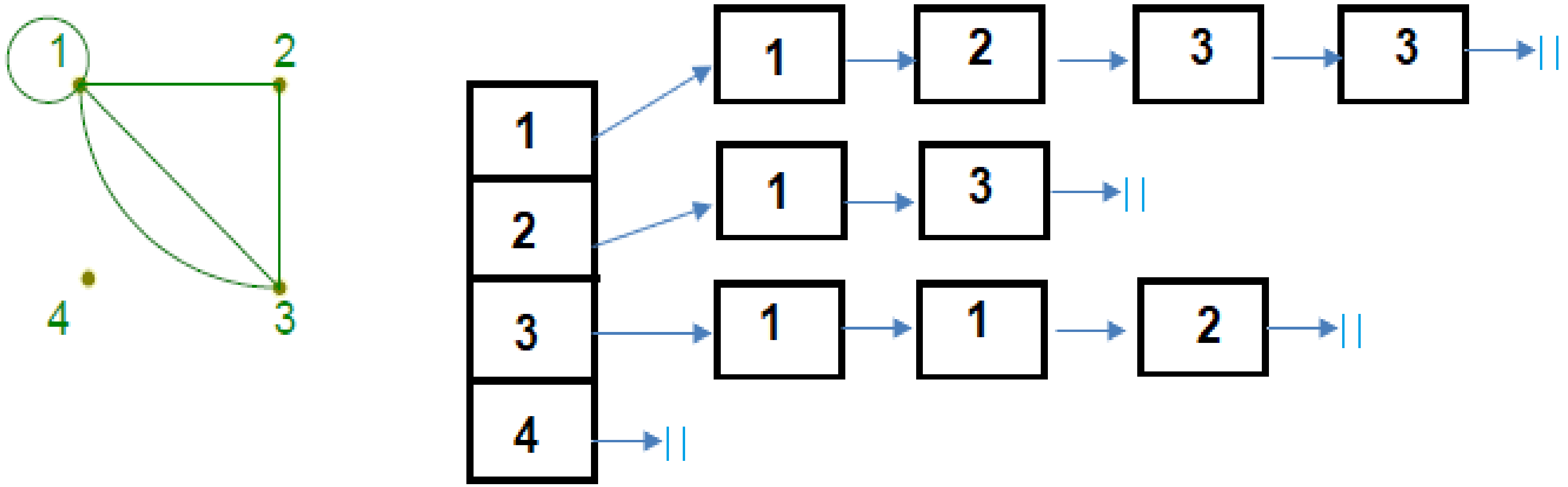
.edges

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•

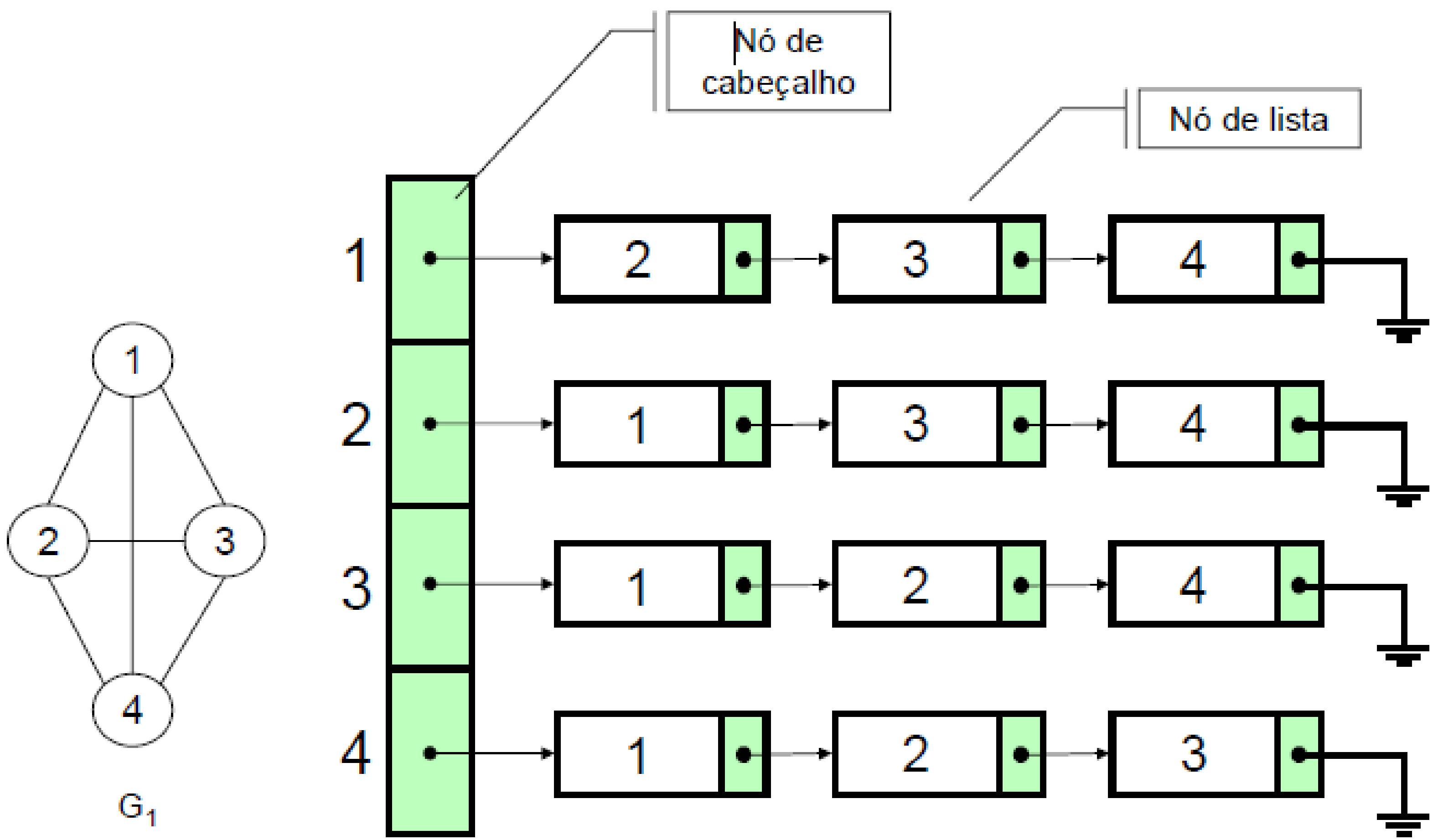
(Array positions marked '•' are undefined)

Lista de adjacências

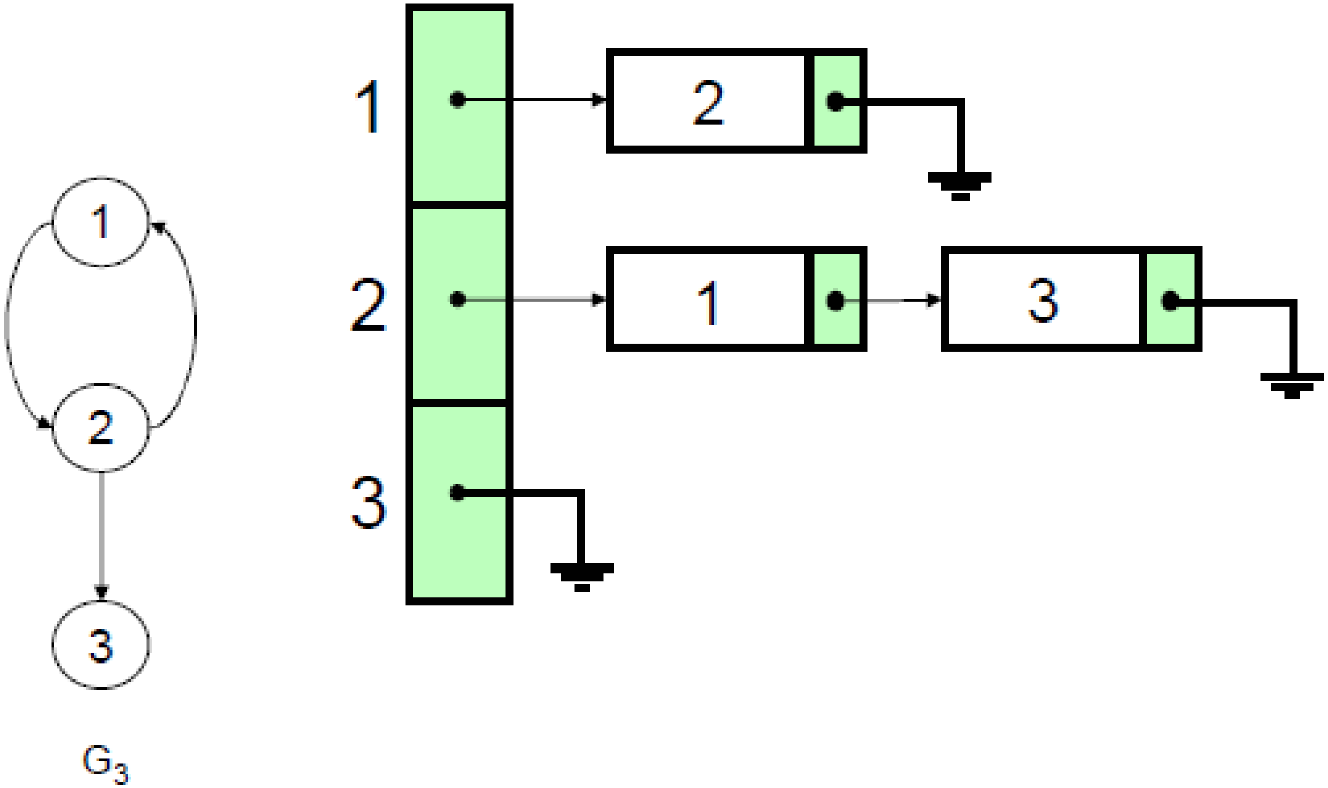
Seja G um grafo com o conjunto de vértices {v_1, v_2, ..., v_n}. A lista de adjacências de G é um conjunto de n células que contêm os vértices e um ponteiro para uma lista de vértices adjacentes.



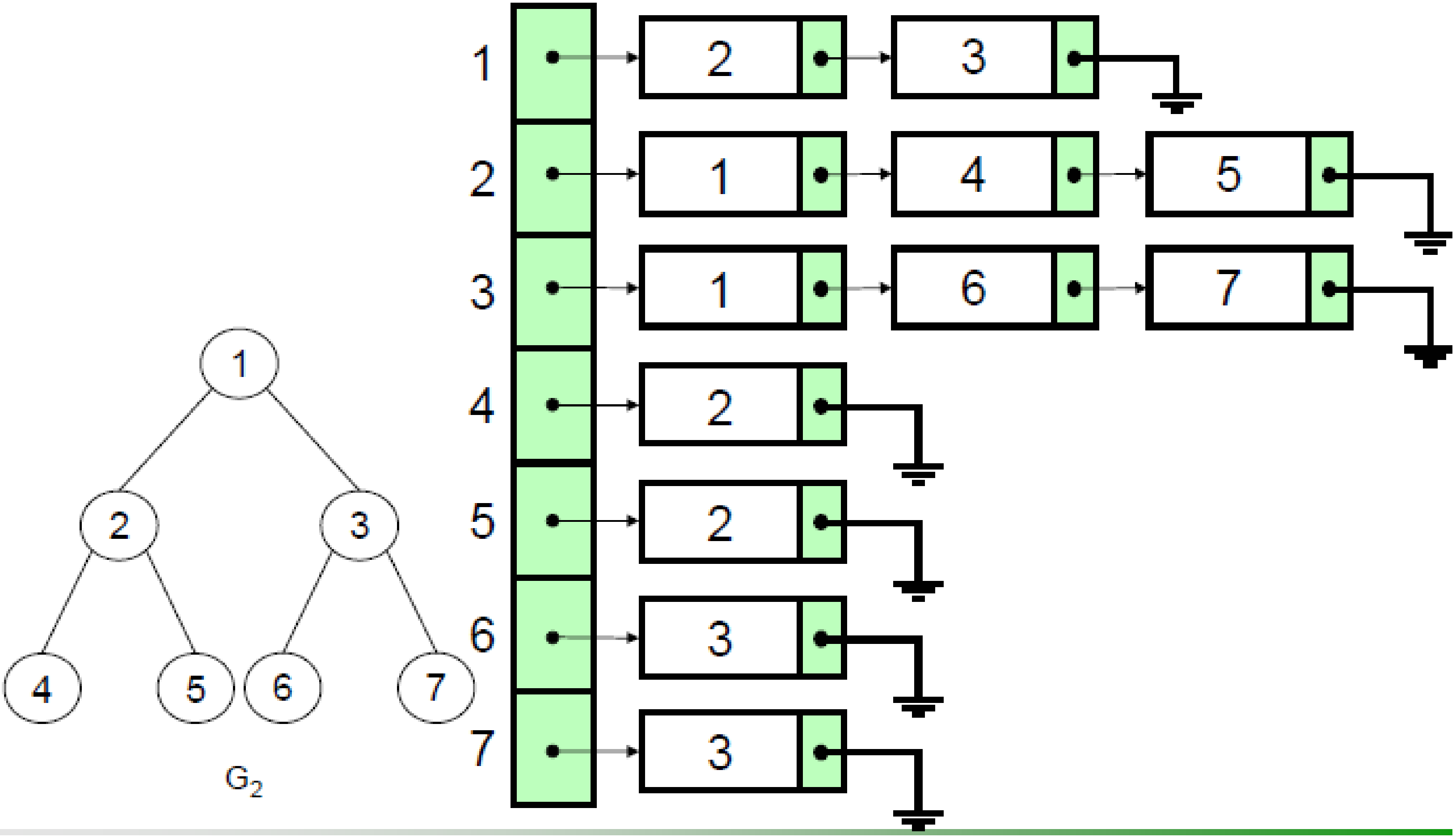
Lista de adjacências



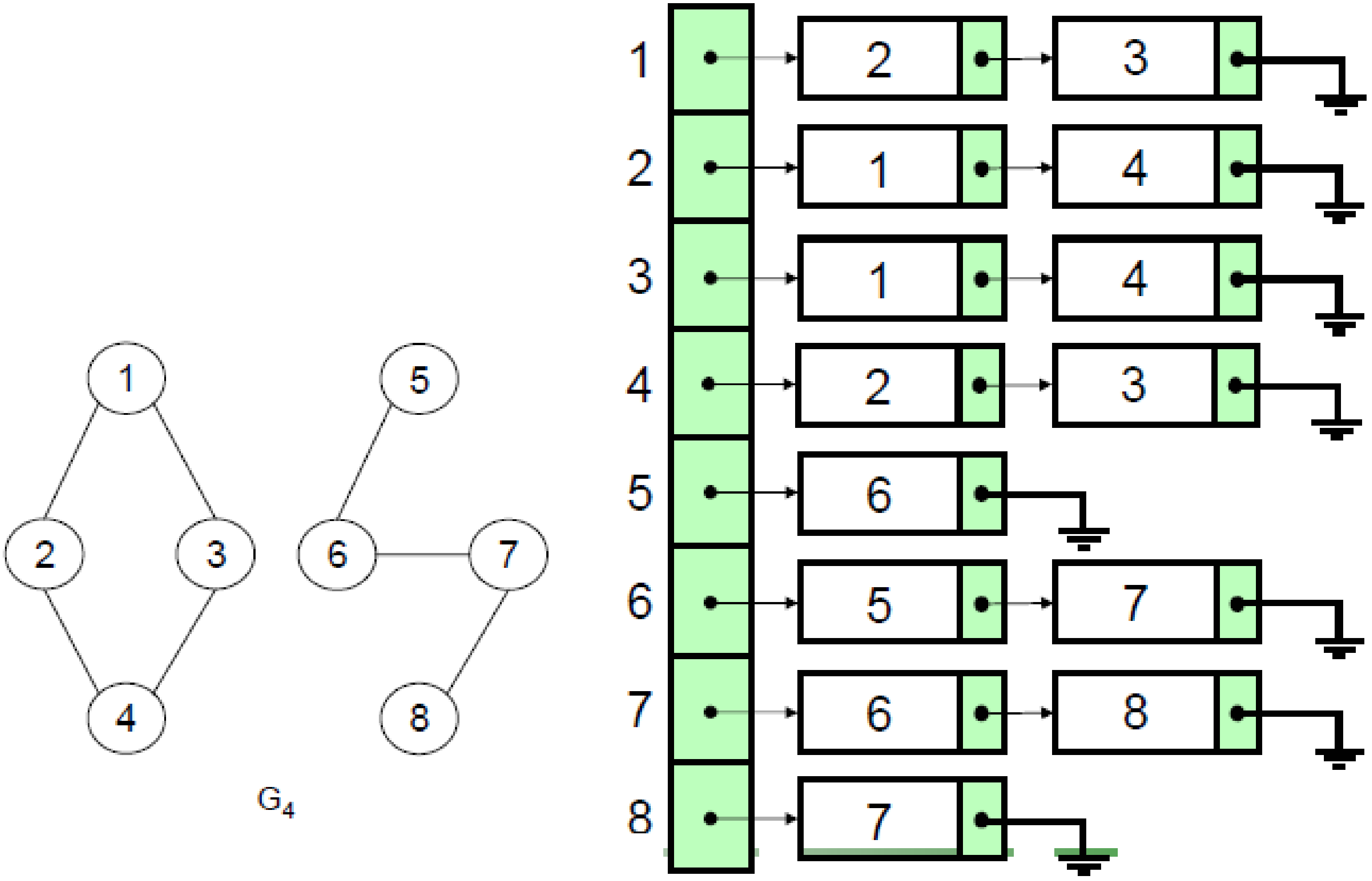
Lista de adjacências



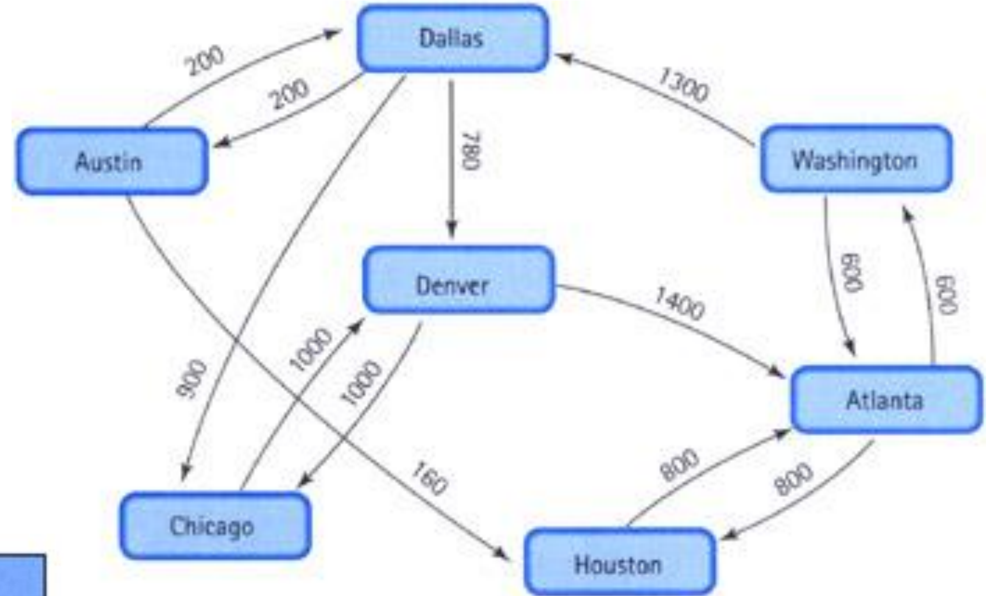
Lista de adjacências

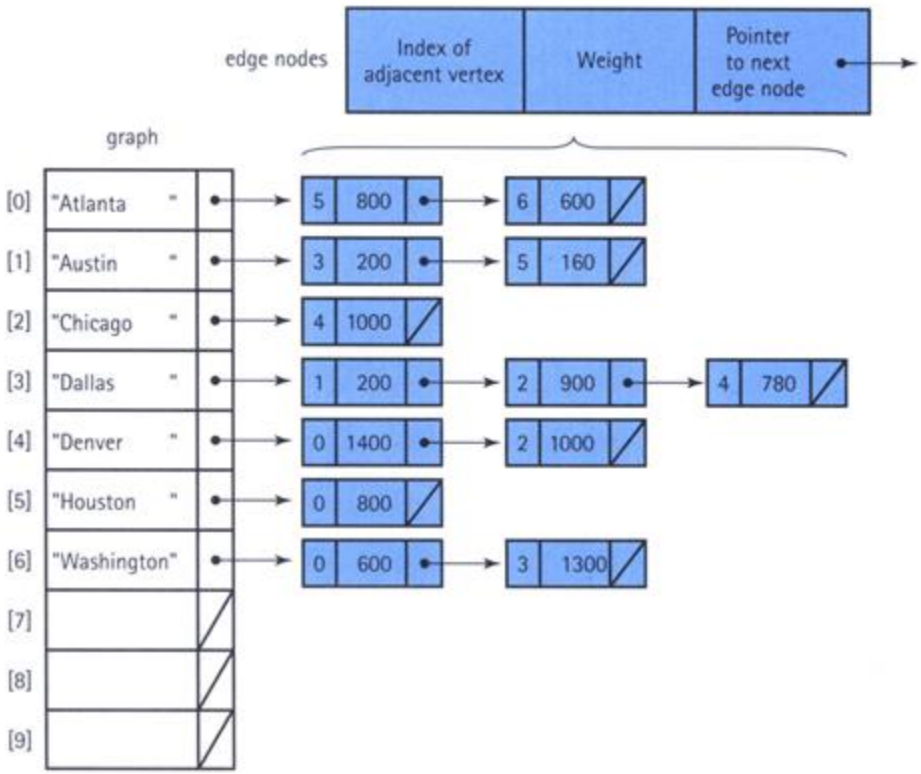


Lista de adjacências

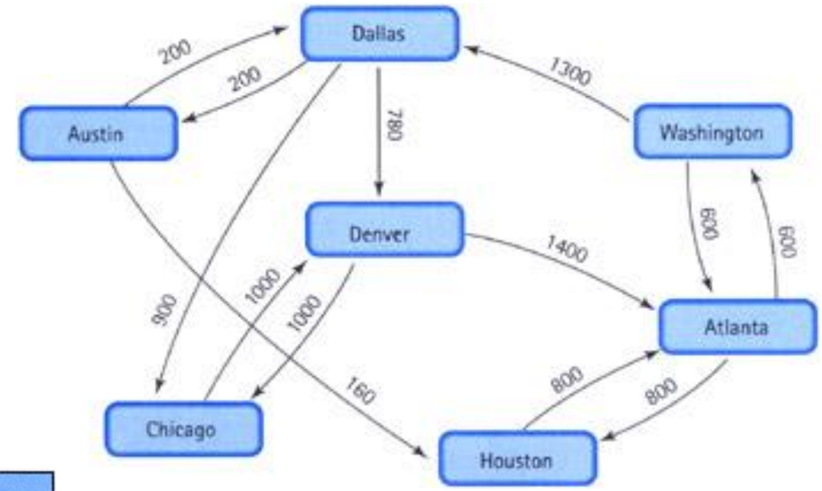


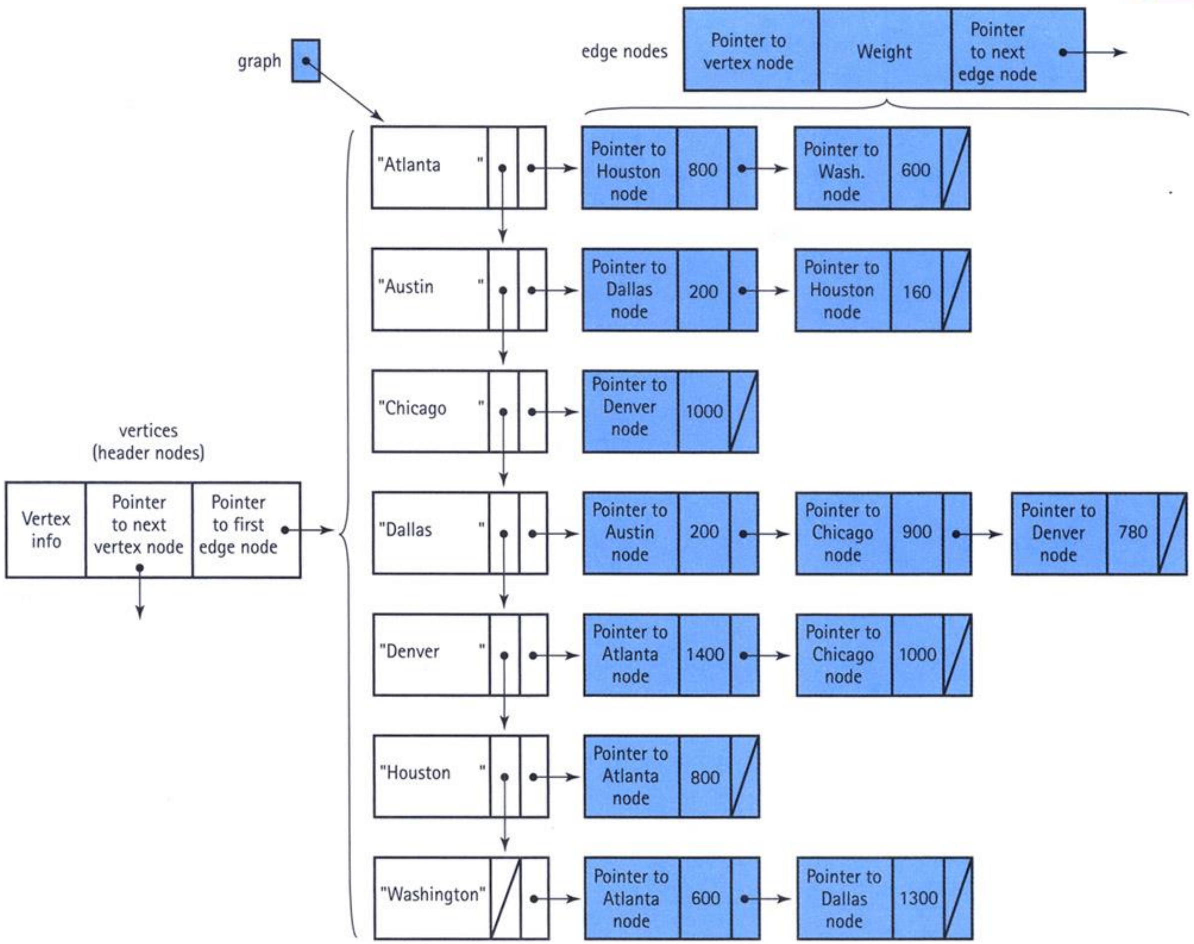
Lista de adjacências
1ª forma





Lista de adjacências
2ª forma





Usando a estrutura dict() do Python

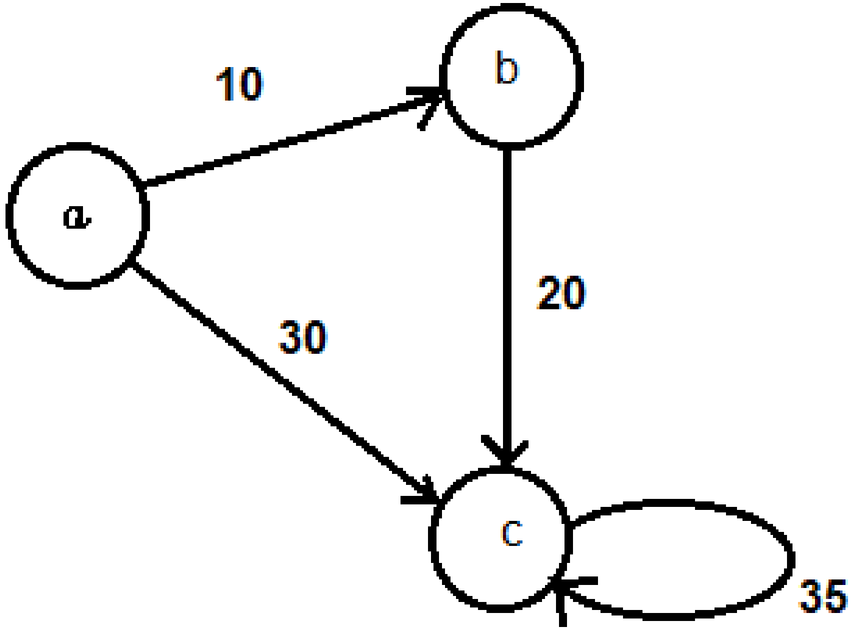
Classe Vertex

Adjacent list

```
{
 vertex.id : { vertex.id: weight ,  vertex.id: weight, .... vertex.id: weight }
 vertex.id : { vertex.id: weight ,  vertex.id: weight, .... vertex.id: weight }
 ....
 vertex.id : { vertex.id: weight ,  vertex.id: weight, .... vertex.id: weight }
}
```

Dicionário com os vértices

Dicionário com os vértices e pesos adjacentes




```
graph={'a': { 'b':10,'c':30},'b':{'c':20},'c':{'c':35}}
```

Busca

- Um problema que ocorre em grafos: sendo G(V,E) um grafo não-orientado e, um vértice **v** em V(G), é visitar todos os vértices em G que são alcançáveis a partir de **v** (isto é, todos os vértices interligados a **v**)

- Problema genérico de busca em grafo

Seja G um grafo conexo

1. Inicialmente, **escolhe-se** um vértice como vértice origem, e inclua-o em uma lista de vértices denominados de **abertos**
2. Retira-se o vértice que está na frente da lista **abertos**, chame-o de corrente e, inclua-o noma lista denominado de **fechados**
3. Após isso, obtenha todos os vértices adjacentes ao vértice corrente e, inclua-os na lista **abertos**
4. Repete-se os passos 2 e 3 até que não exista mais vértices em **abertos**

Busca

- Busca em Profundidade (*depth first search*)

  Uma busca é dita em profundidade quando o critério de escolha do vértice marcado (a partir do qual será realizada a próxima visita de aresta) obedecer ao seguinte critério:

  “Dentre todos os vértices marcados e incidentes a alguma aresta ainda não visitada, escolher aquele ***mais*** *recentemente alcançado na busca”*

  Nesse caso usamos a estrutura PILHA para armazenar os abertos

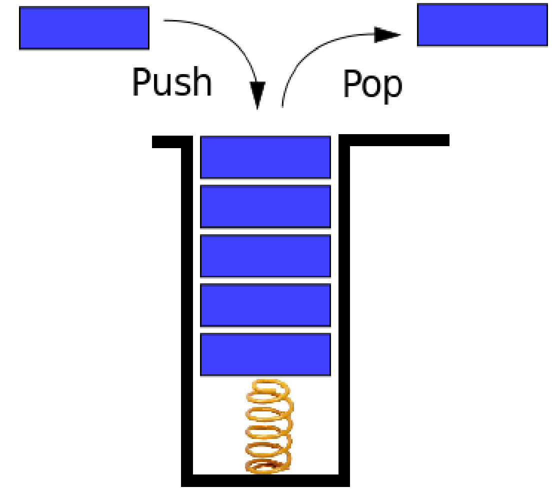
- Busca em Largura (*breadth first search*)

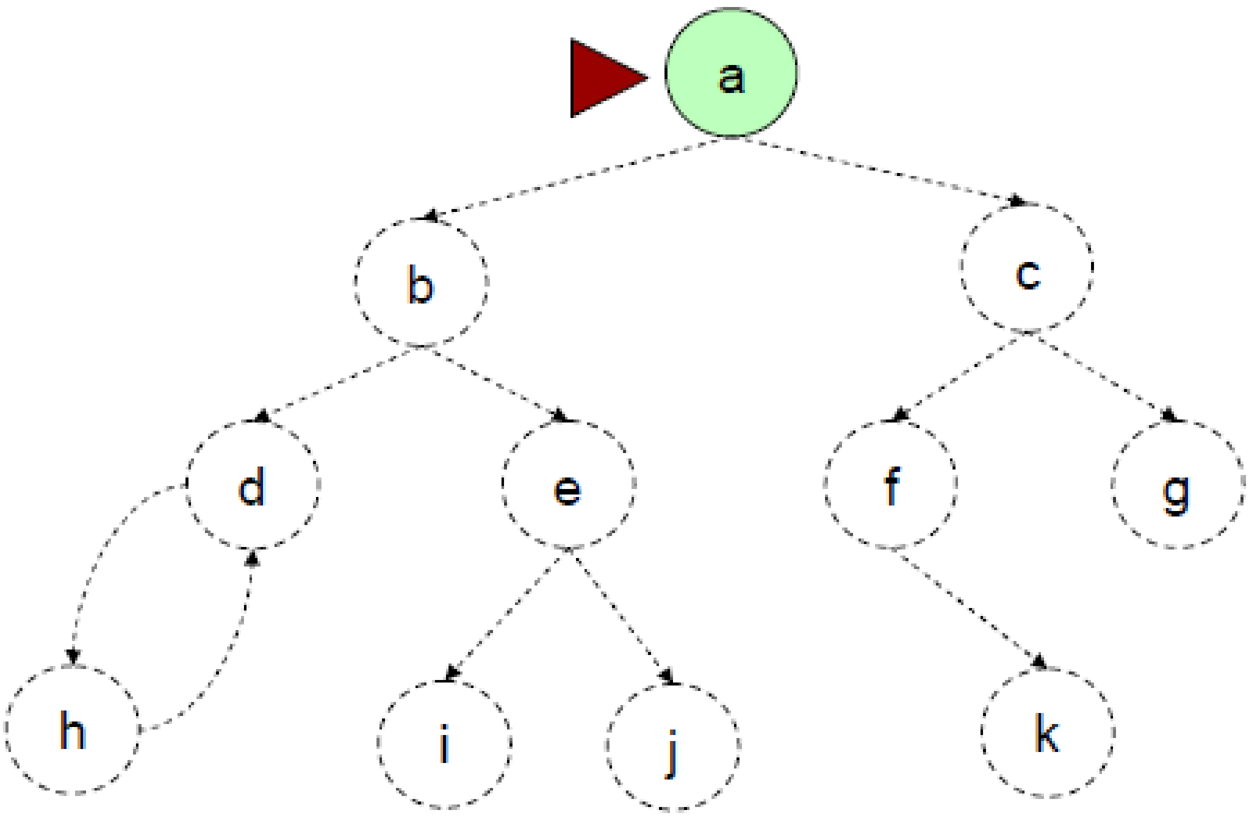
  Uma busca é dita em largura quando o critério de escolha do vértice marcado (a partir do qual será realizada a próxima visita de aresta) obedecer ao seguinte critério:

  “Dentre todos os vértices marcados e incidentes a alguma aresta ainda não visitada, escolher aquele ***menos*** *recentemente alcançado na busca”*

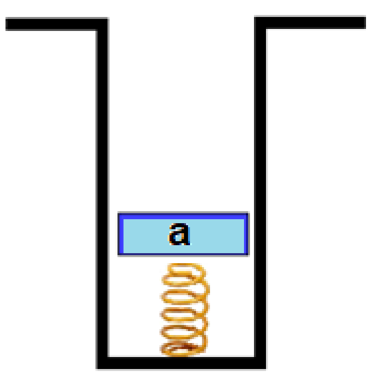
  Nesse caso usamos a estrutura FILA para armazenar os abertos

Busca em profundidade



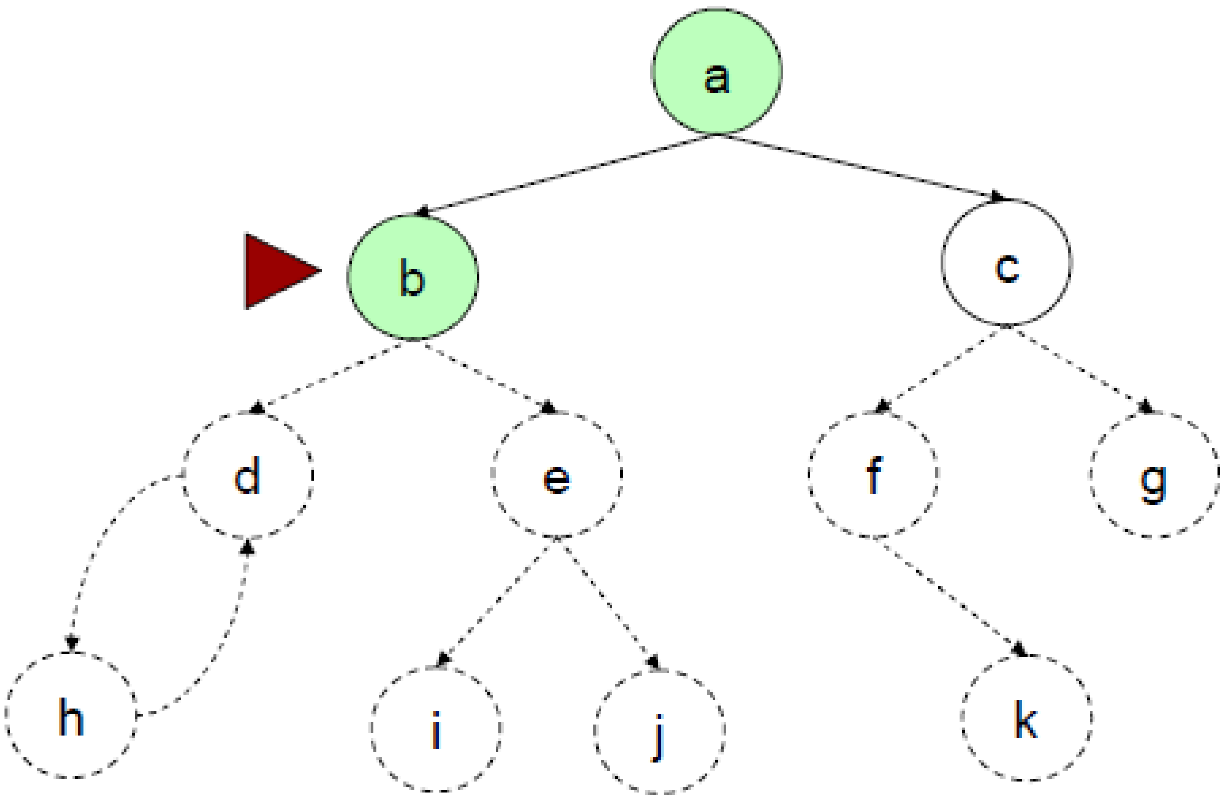


Abertos

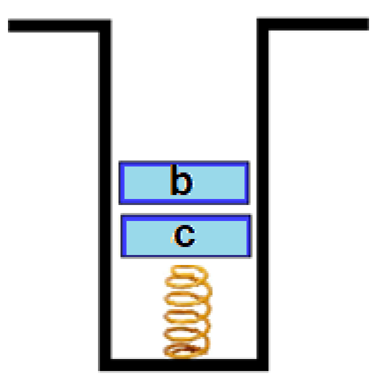


Fechados: [ ]

Busca em profundidade

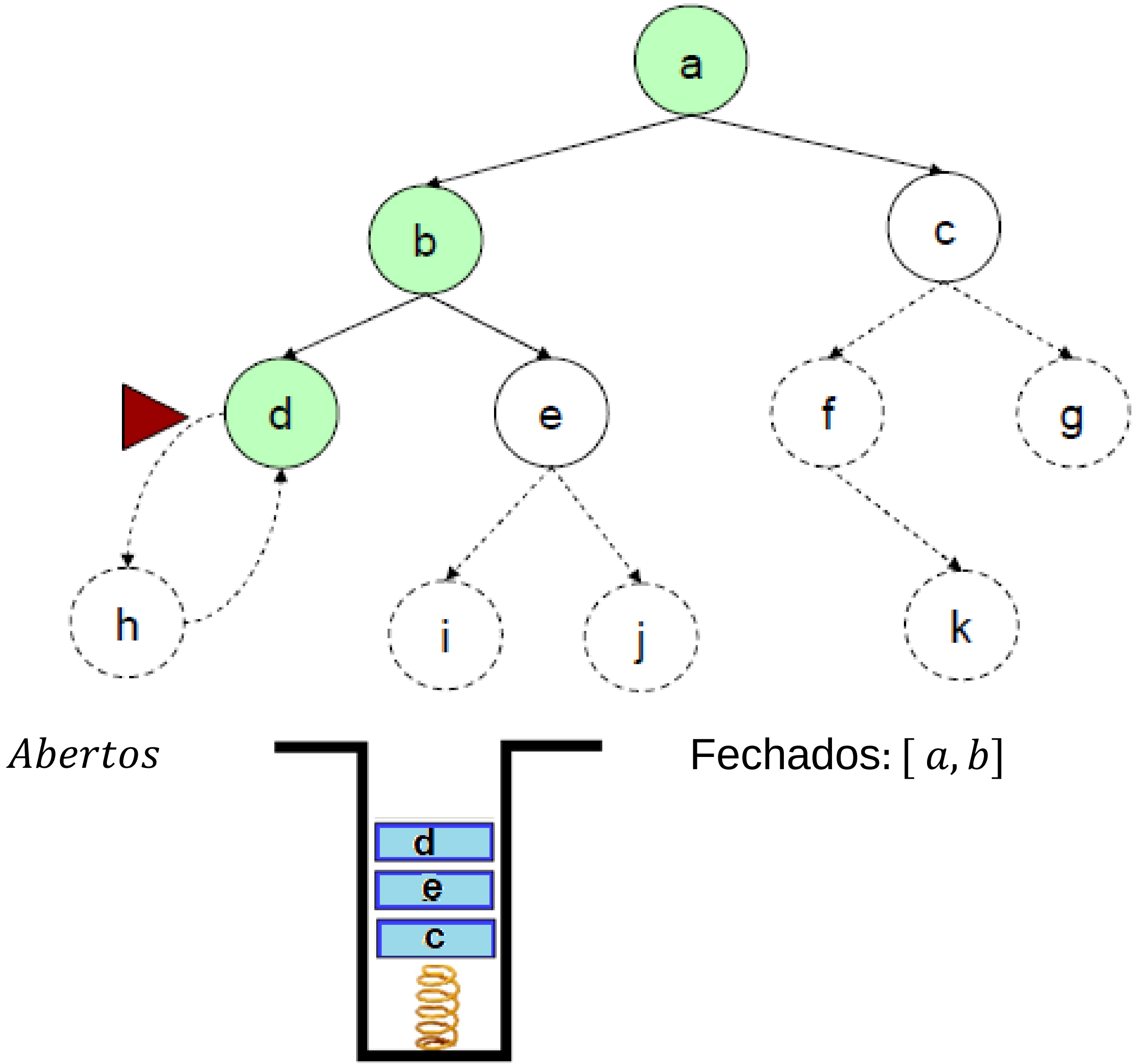


$Abertos$



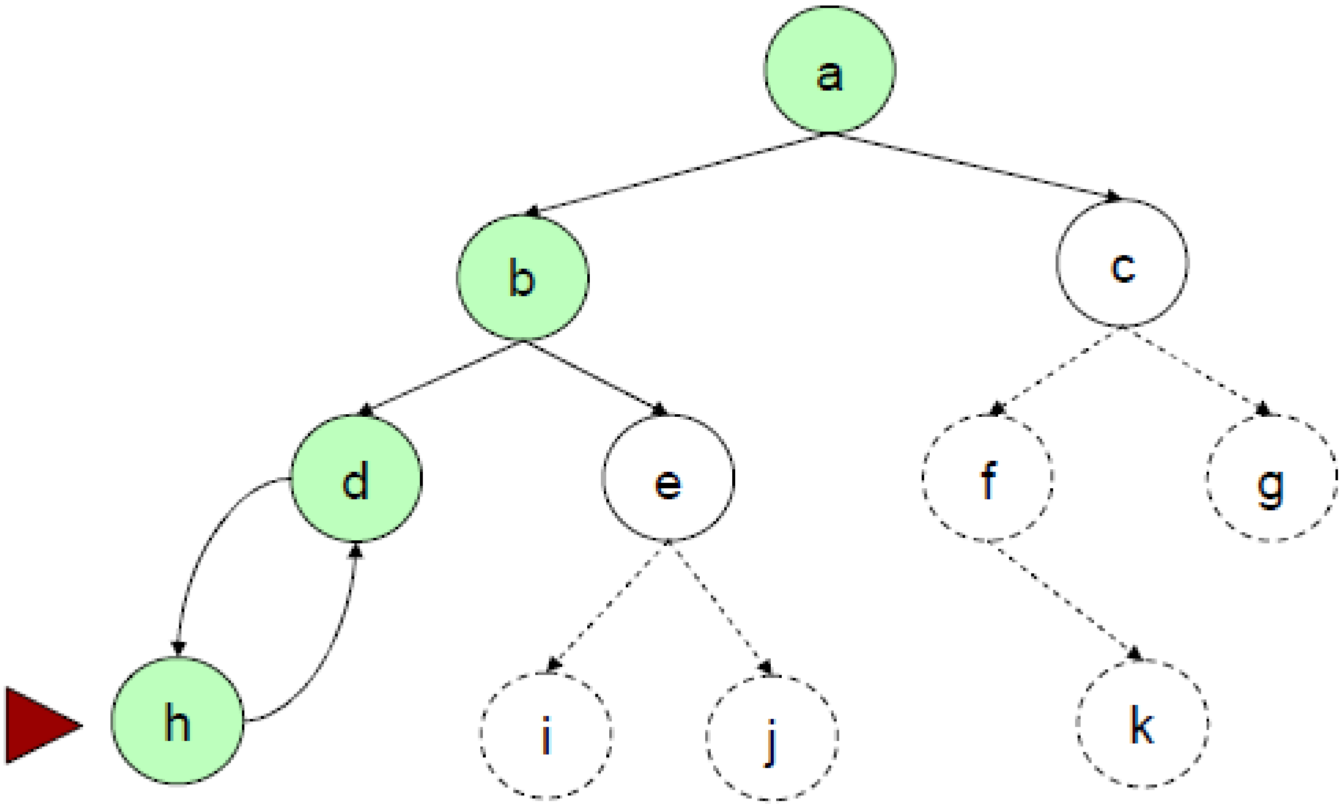
Fechados: [a]

Busca em profundidade

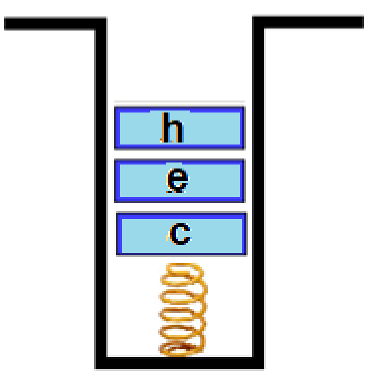


Fechados: [a, b]

Busca em profundidade



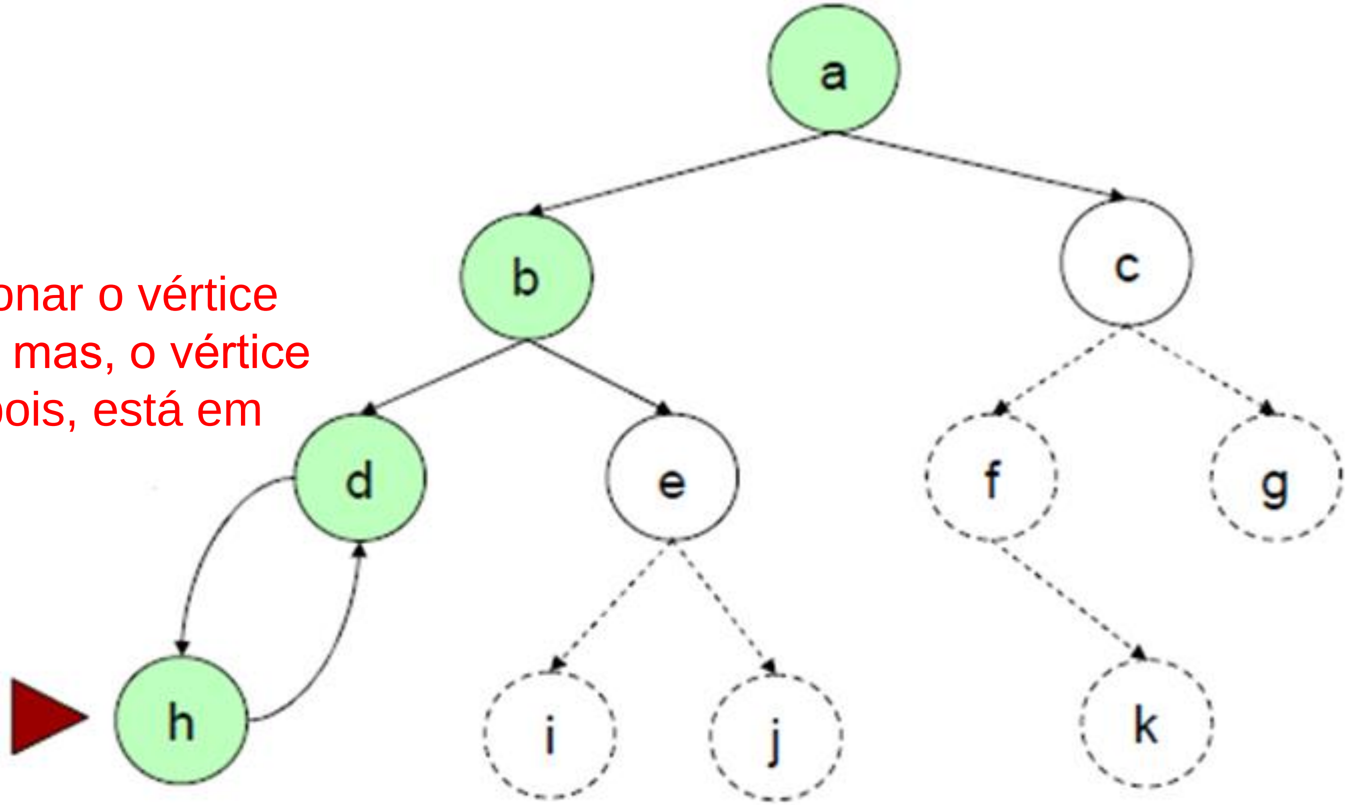
$Abertos$



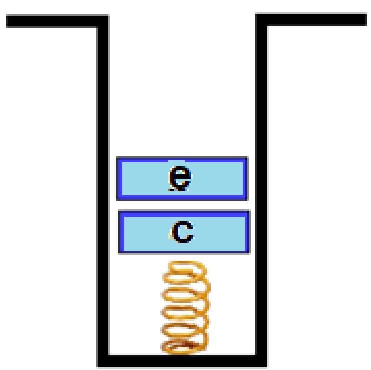
Fechados: [a, b, d]

Busca em profundidade

Ele tenta adicionar o vértice “d” em abertos mas, o vértice já foi visitado pois, está em fechados

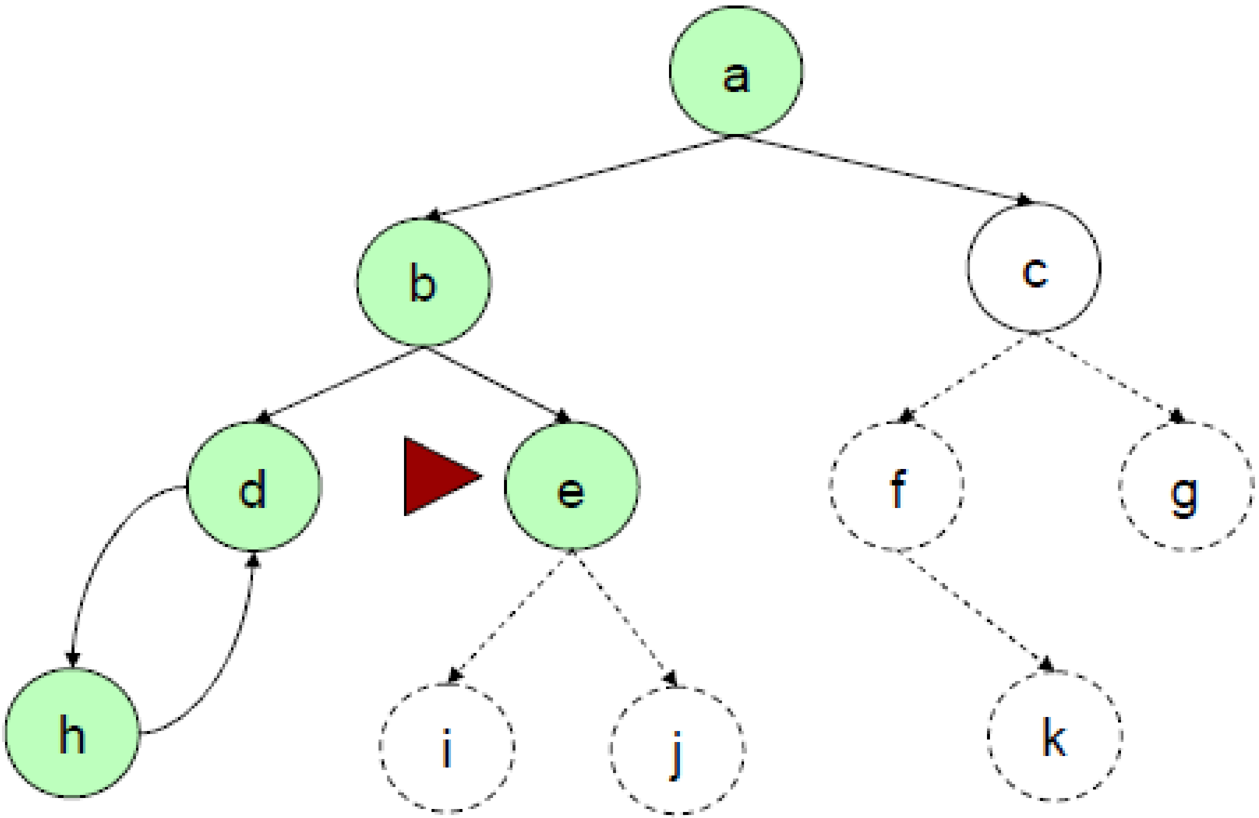


$Abertos$

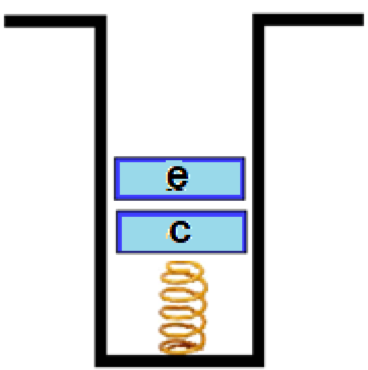


Fechados: $[\,a, b, d, h]$

Busca em profundidade

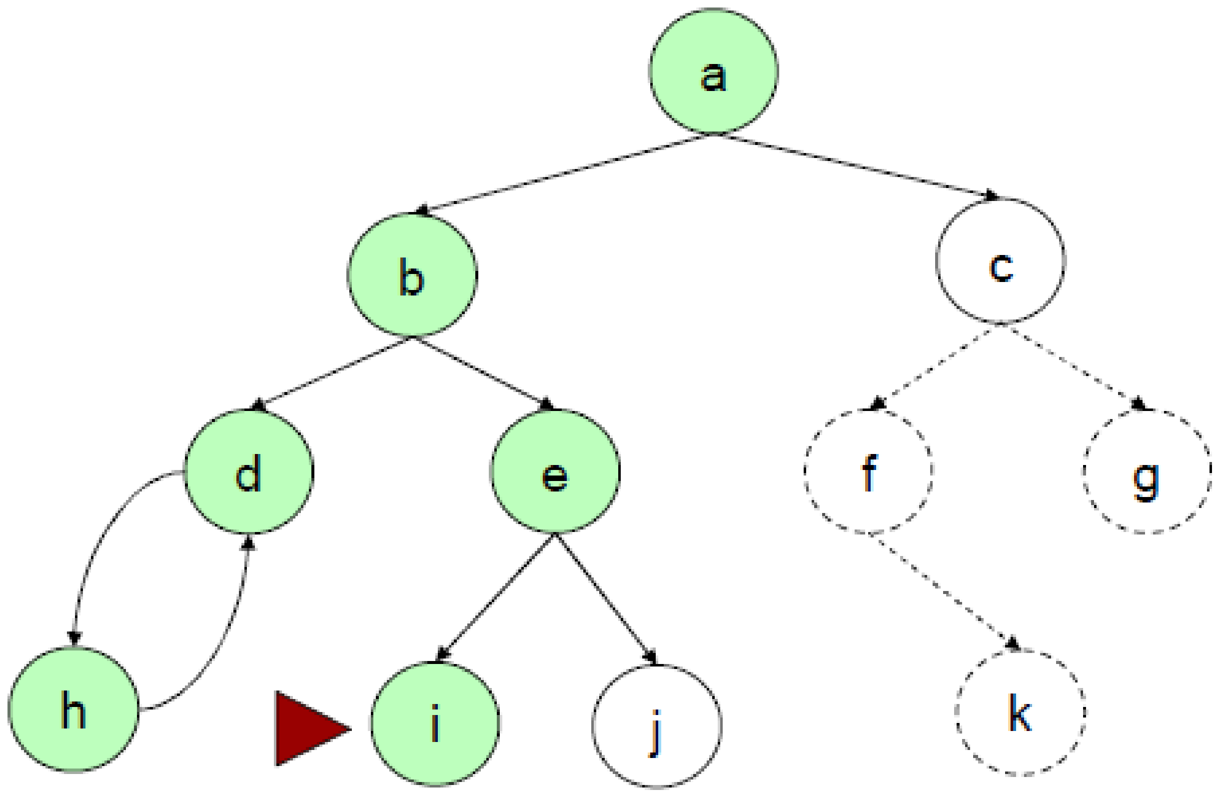


$Abertos$

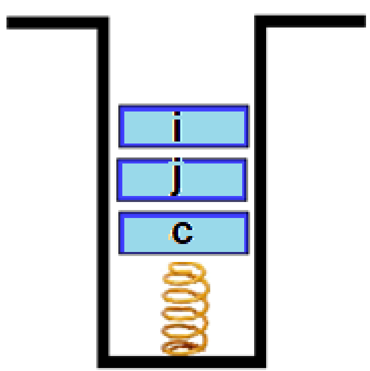


Fechados: [a, b, d, h]

Busca em profundidade

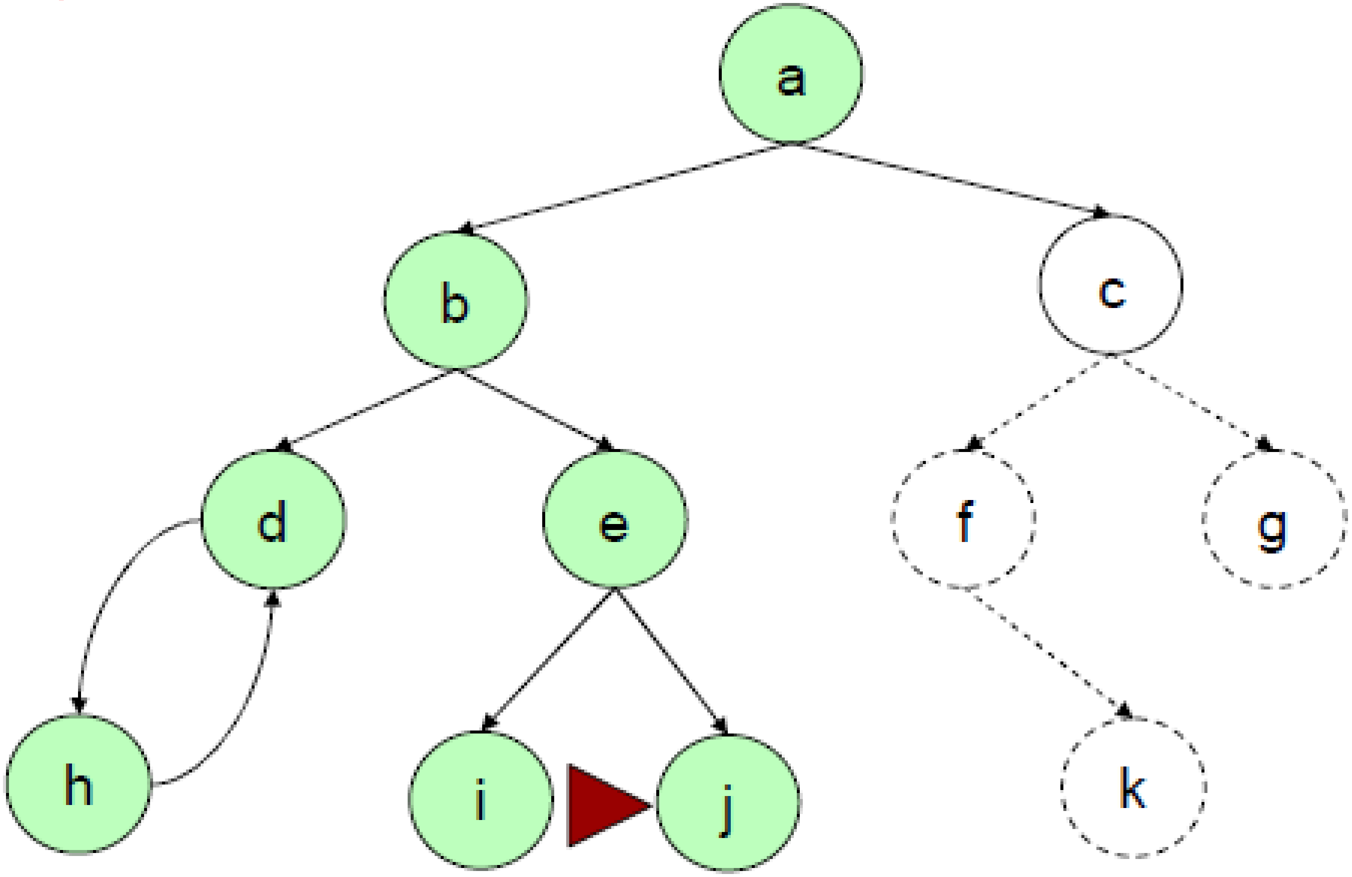


$Abertos$

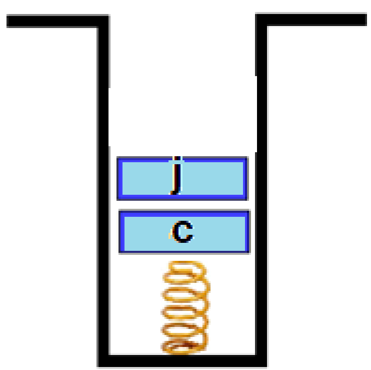


Fechados: [$a, b, d, h. e$]

Busca em profundidade

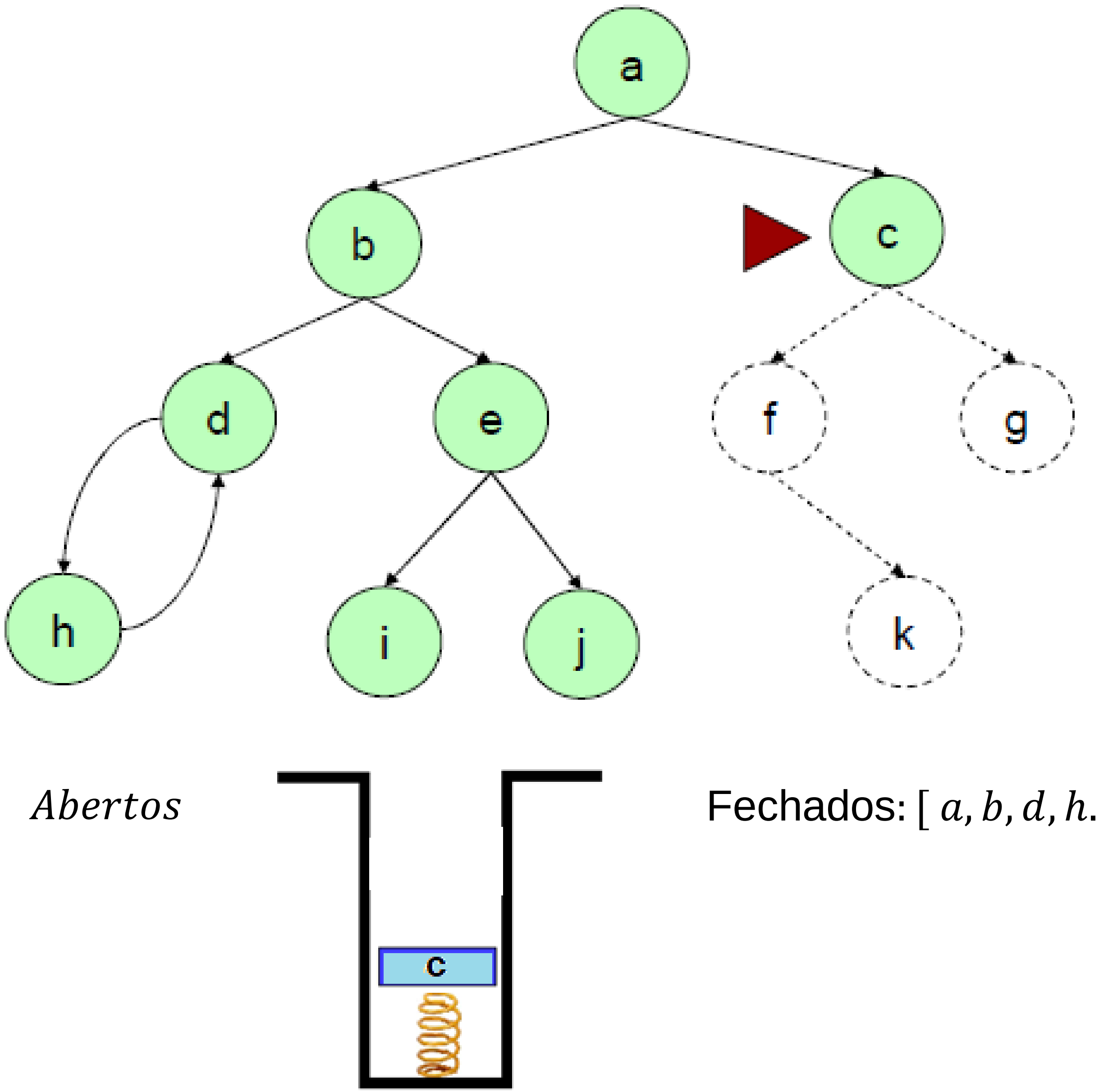


$Abertos$



Fechados: [$a, b, d, h. e. i$]

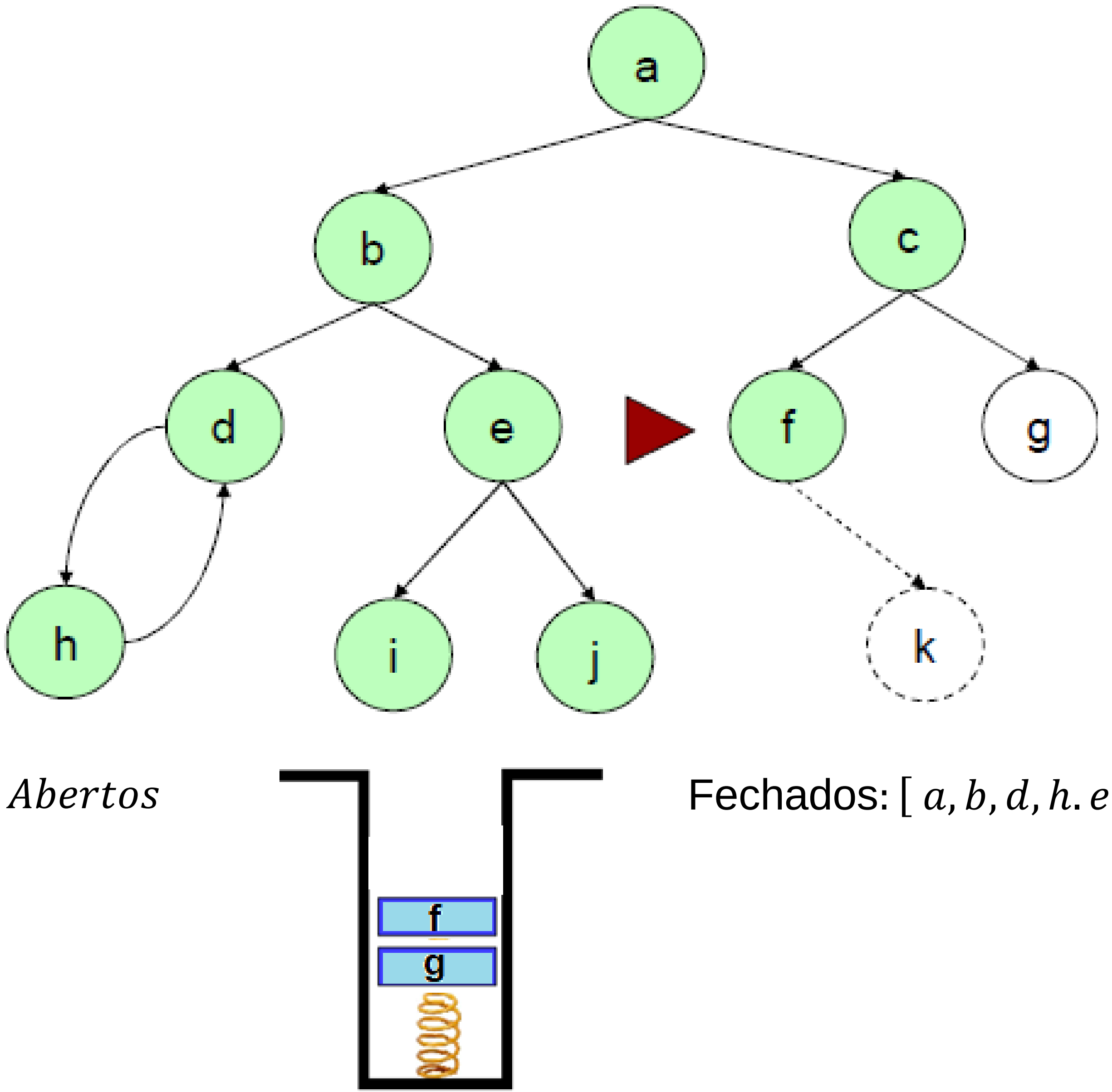
Busca em profundidade



Abertos

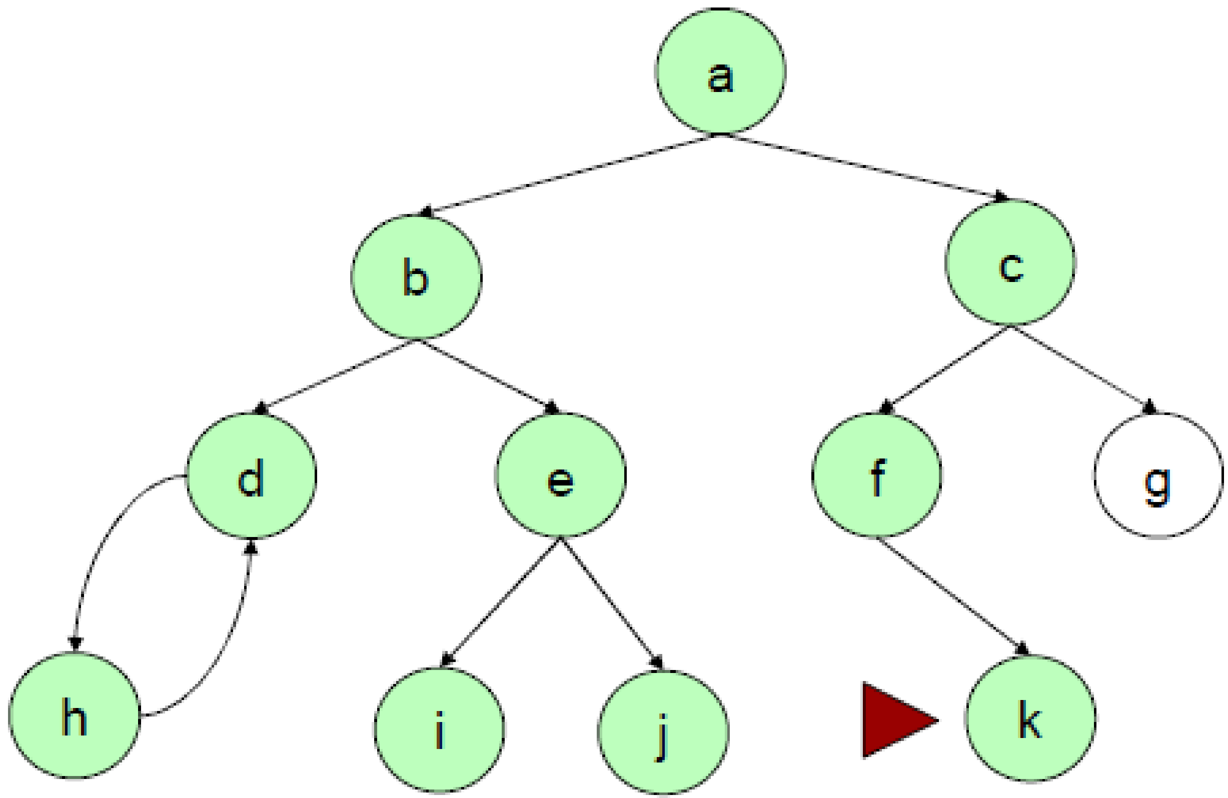
Fechados: [$a, b, d, h. e. i. j$]

Busca em profundidade

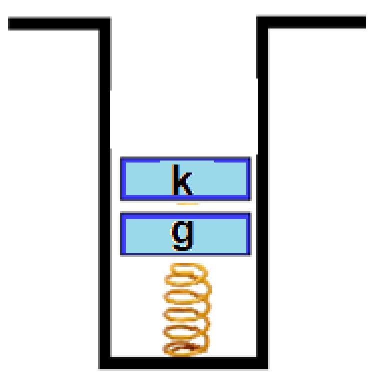


Fechados: [$a, b, d, h. e. i. j, c$]

Busca em profundidade

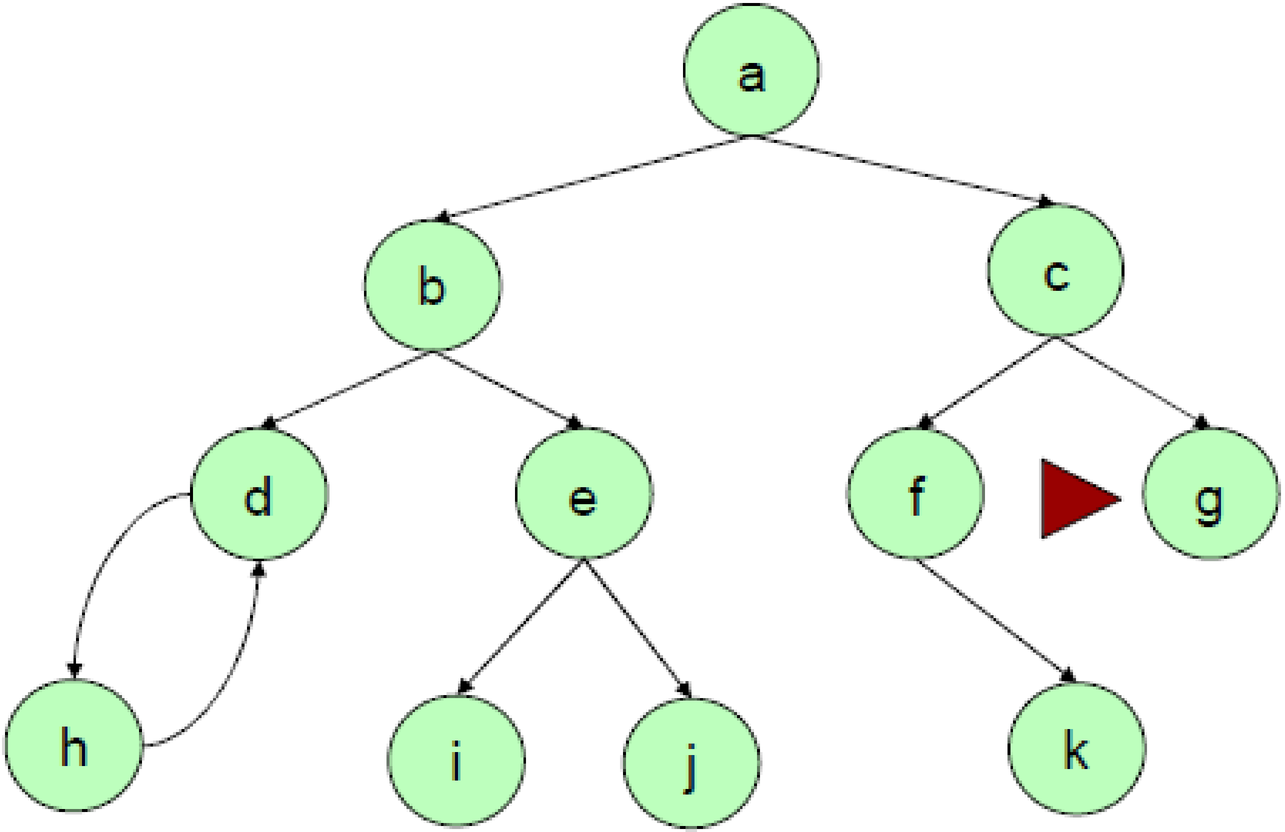


Abertos

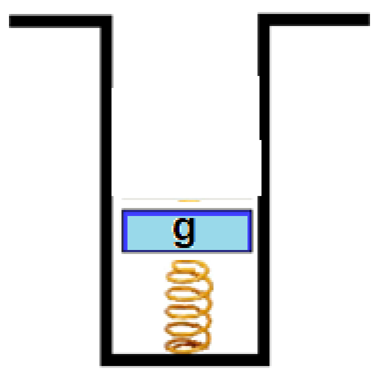


Fechados: [$a, b, d, h. e. i. j, c, f$]

Busca em profundidade

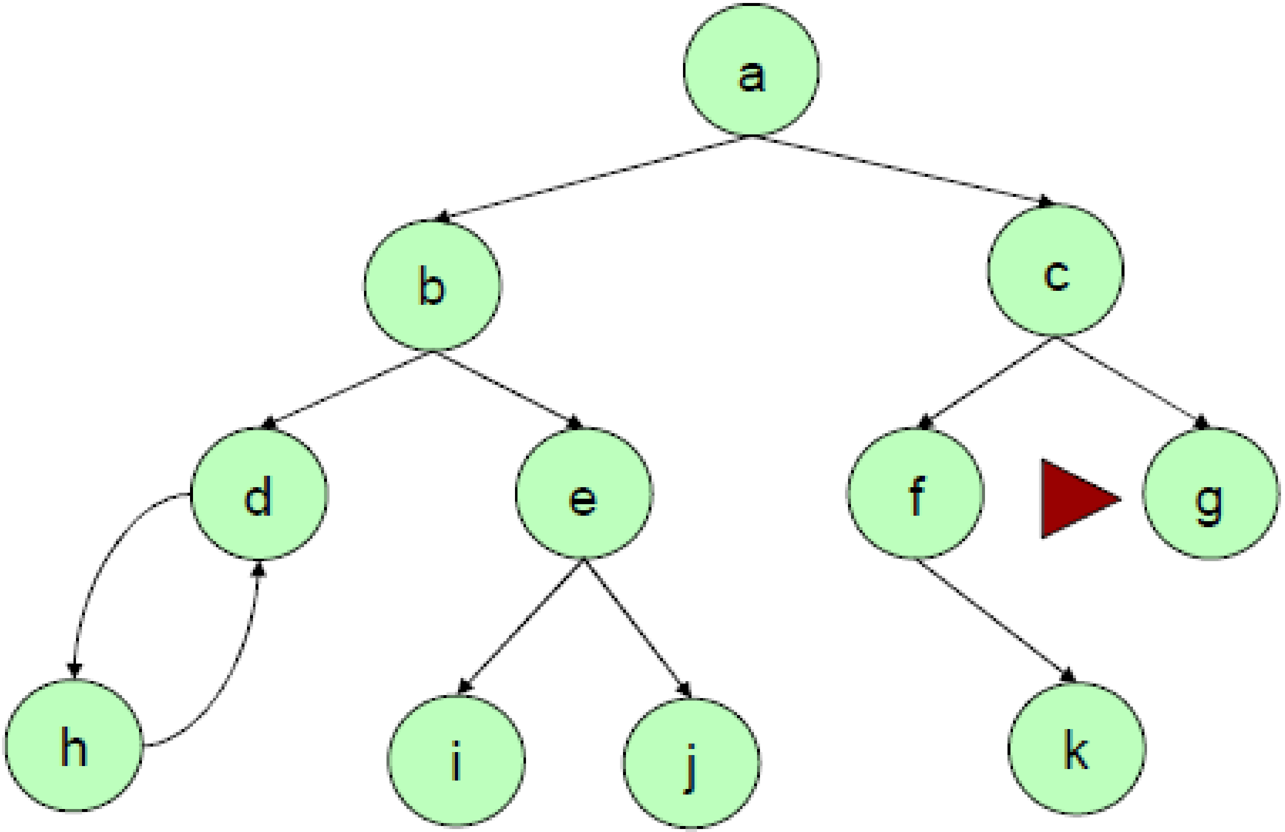


$Abertos$

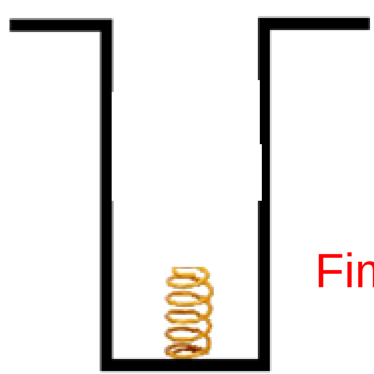


Fechados: [$a, b, d, h. e. i. j, c, f, k$]

Busca em profundidade



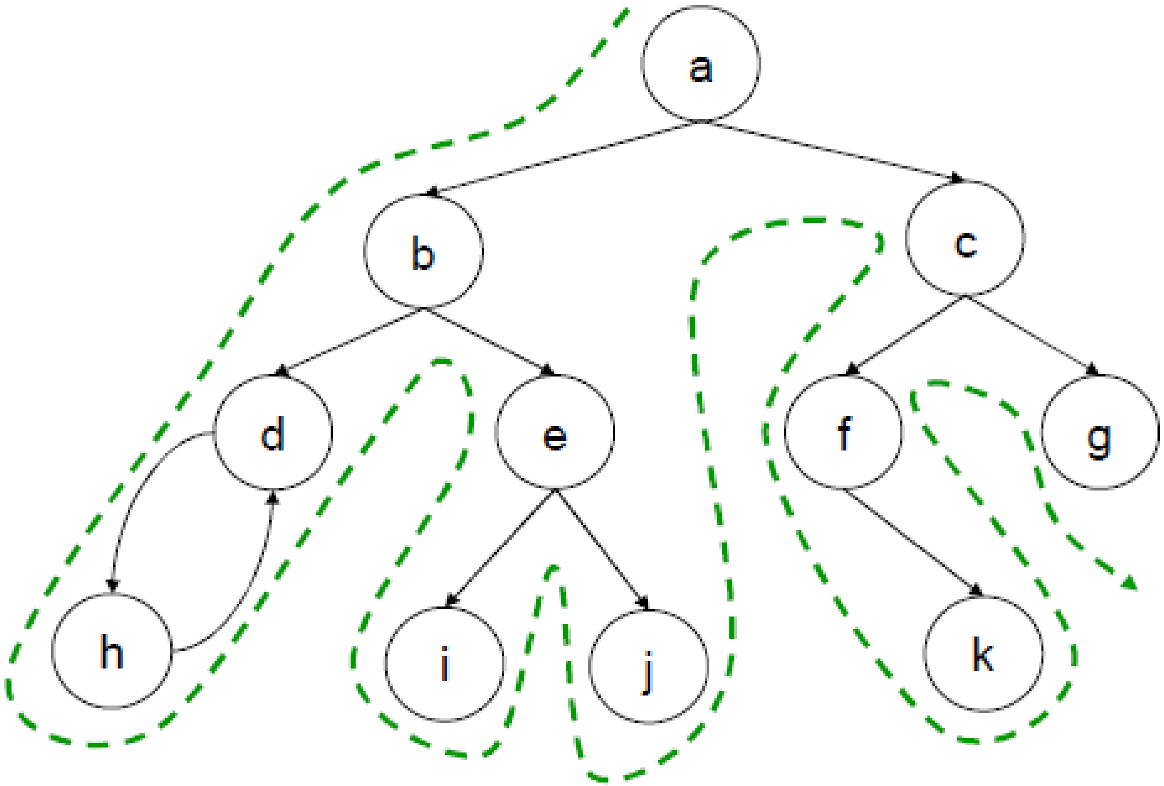
$Abertos$



Fechados: [$a, b, d, h. e. i. j, c, f, k, g$]

Fim da pesquisa! Abertos está vazia!!

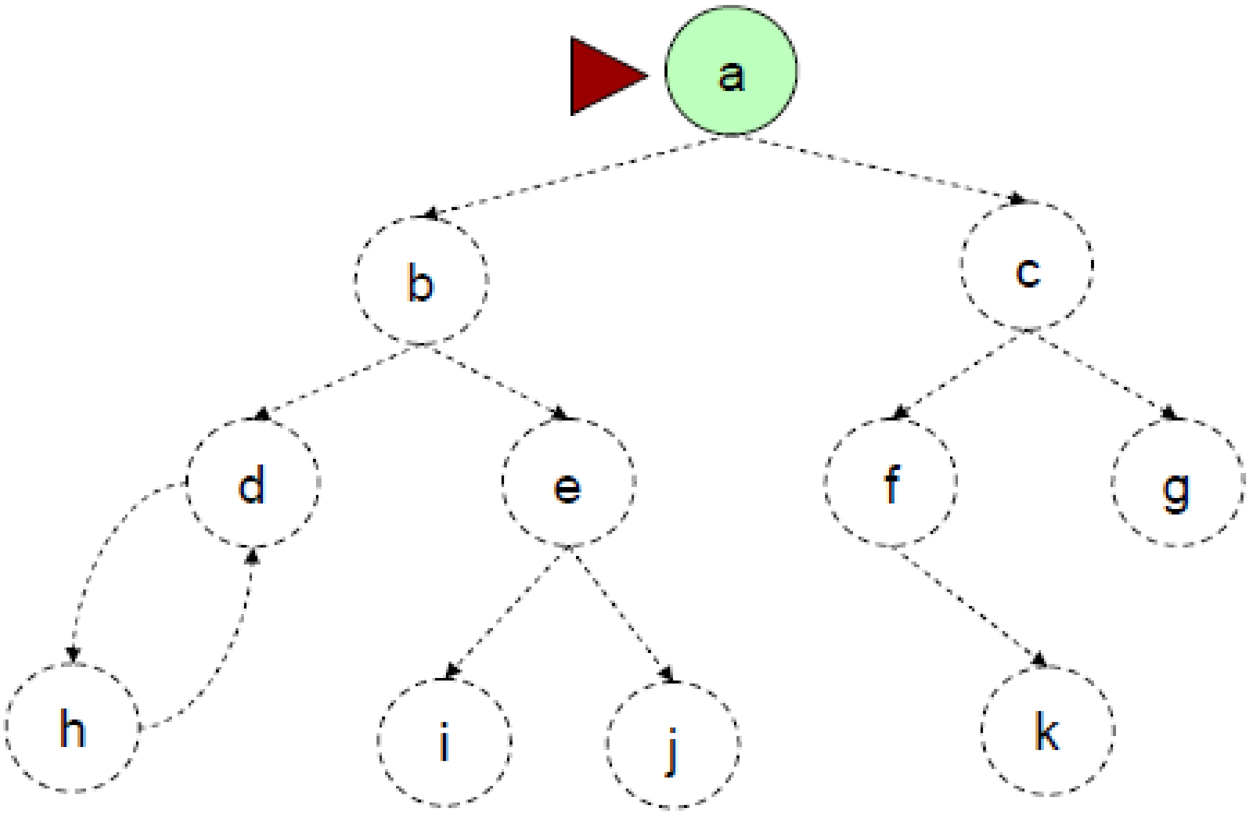
Busca em profundidade



Ordem de visita partindo do vértice a: a,b,d,h,e,i,j,c,f,g,k

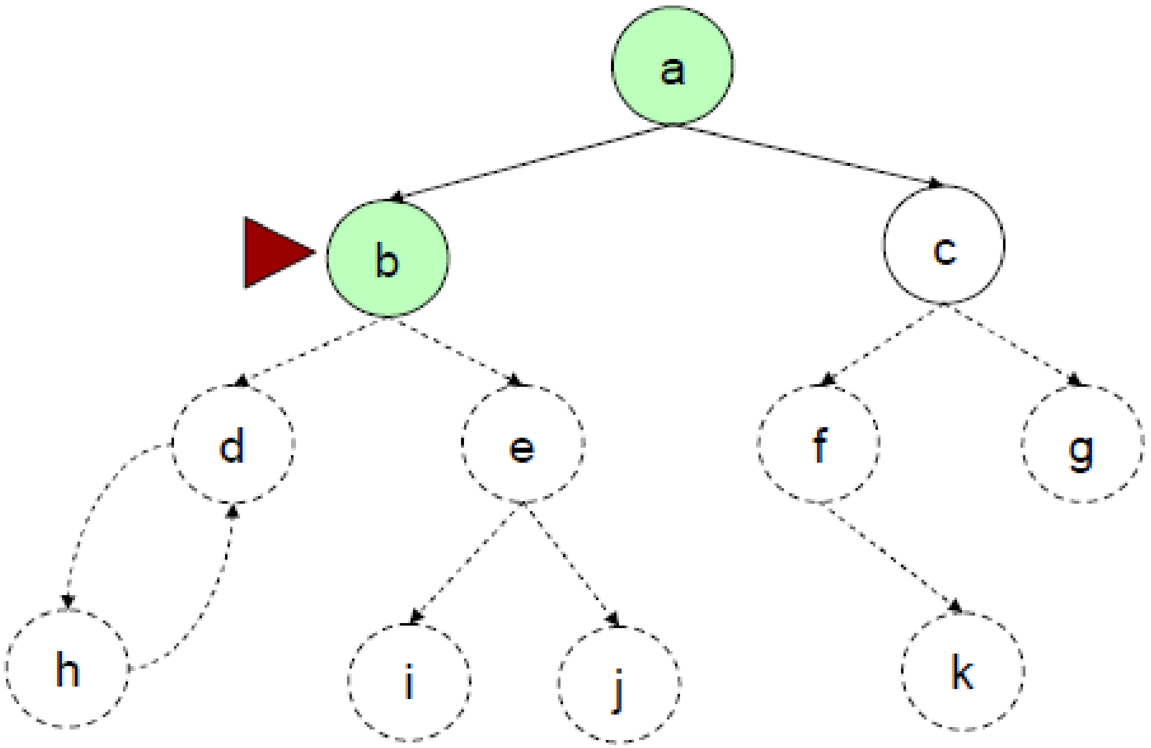
```
def depth__first_search(graph,initial):
    from Stack import stack
    frontier = stack()
    frontier.push(initial)
    explored = list()
    while not frontier.isEmpty():
        curr = frontier.pop()
        explored.append(curr)
        for child,weight in graph.successors(curr):
            if child not in frontier and child not in explored:
                frontier.push(child)
    return explored
```

Busca em largura



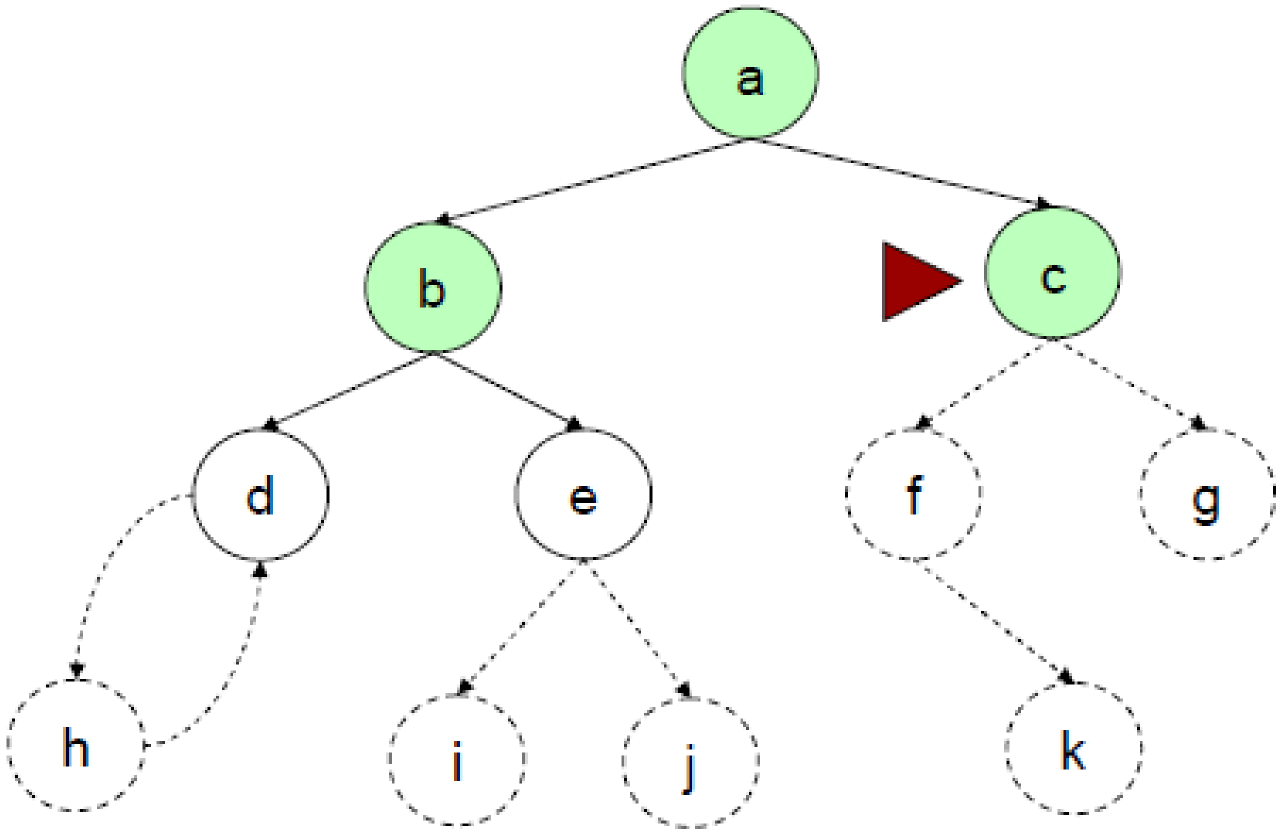
$Abertos$: $[a]$
Fechados: [ ]

Busca em largura



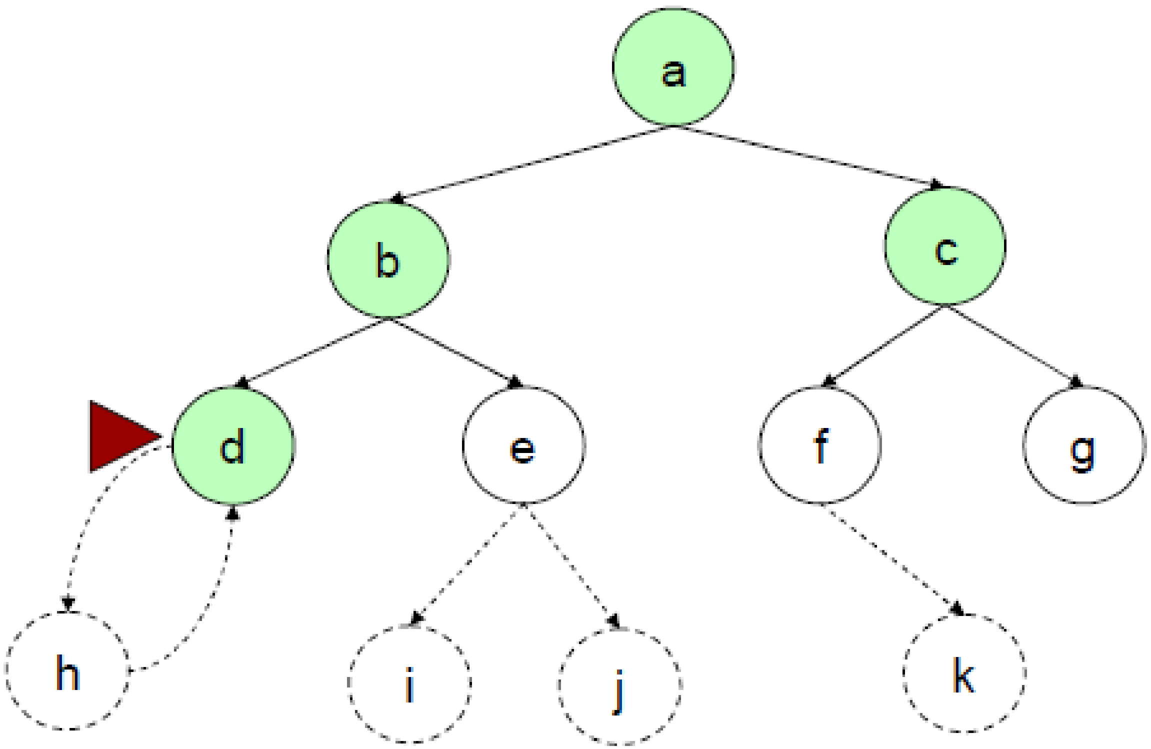
$Abertos$: $[b, c]$
Fechados: $[a\ ]$

Busca em largura



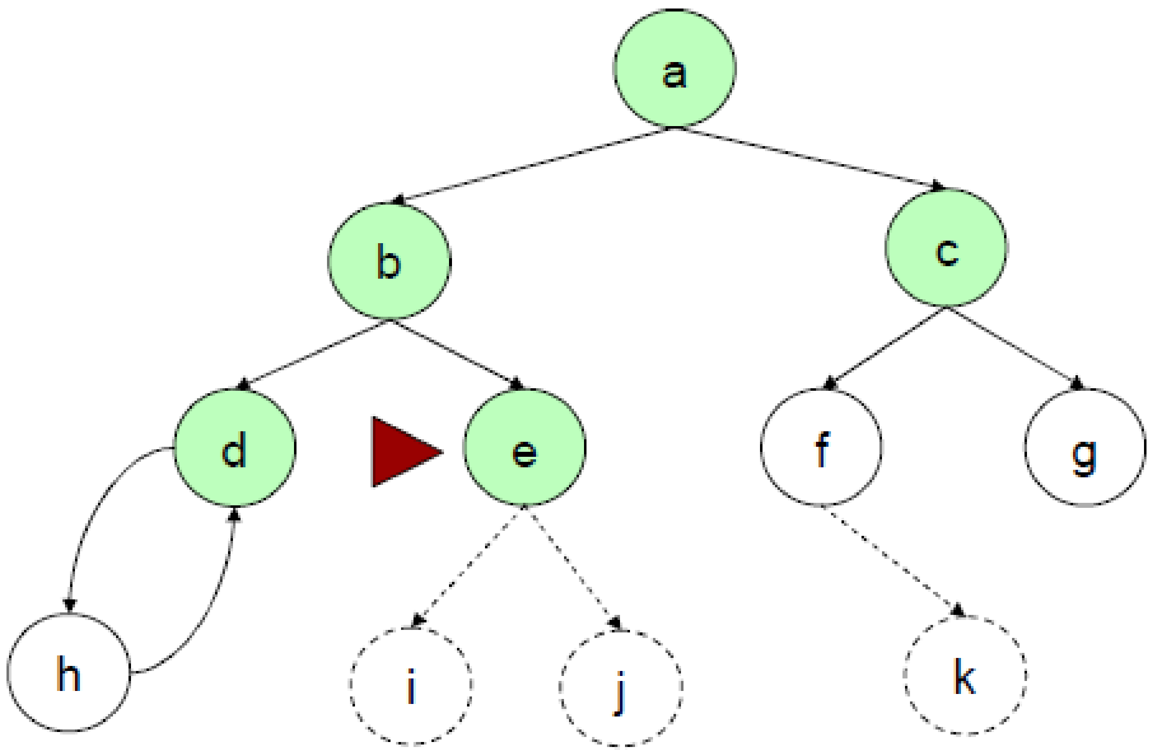
$Abertos$: $[c, d, e]$
Fechados: $[a, b\ ]$

Busca em largura



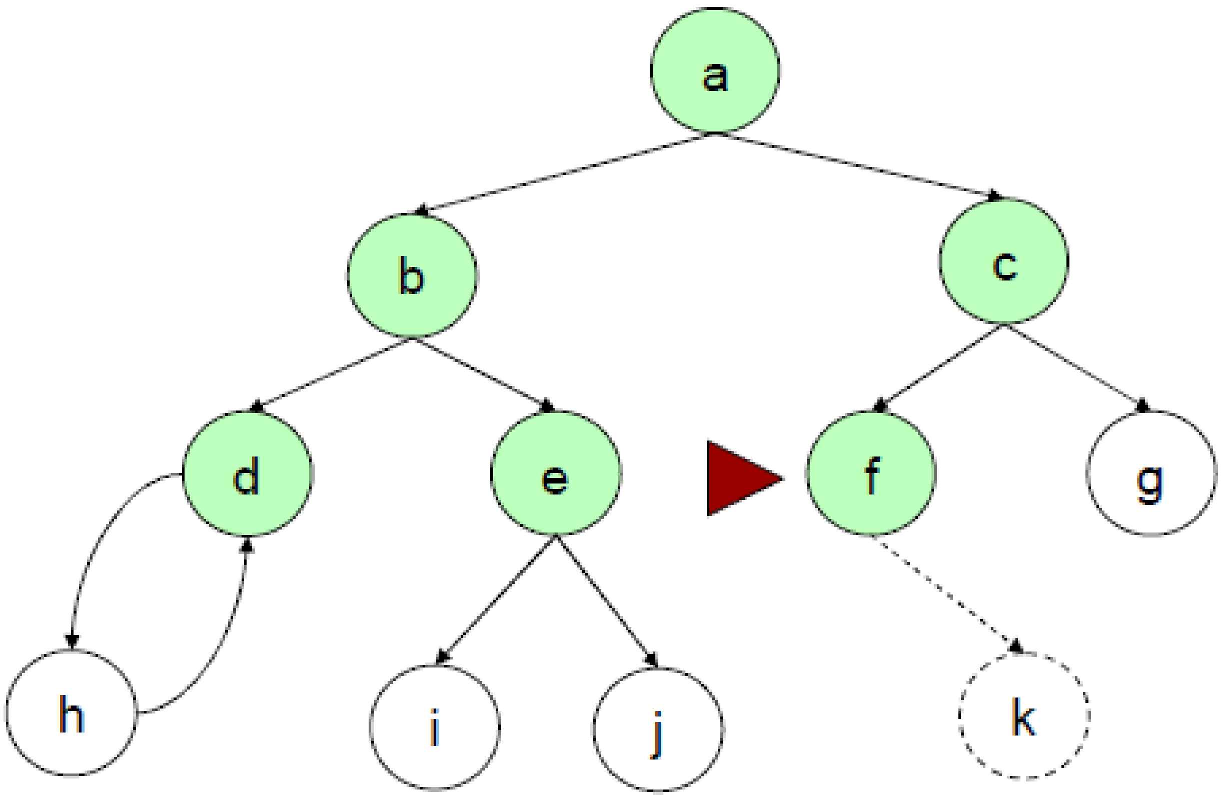
$Abertos$: $[d, e, f, g]$
Fechados: $[a, b, c\ ]$

Busca em largura



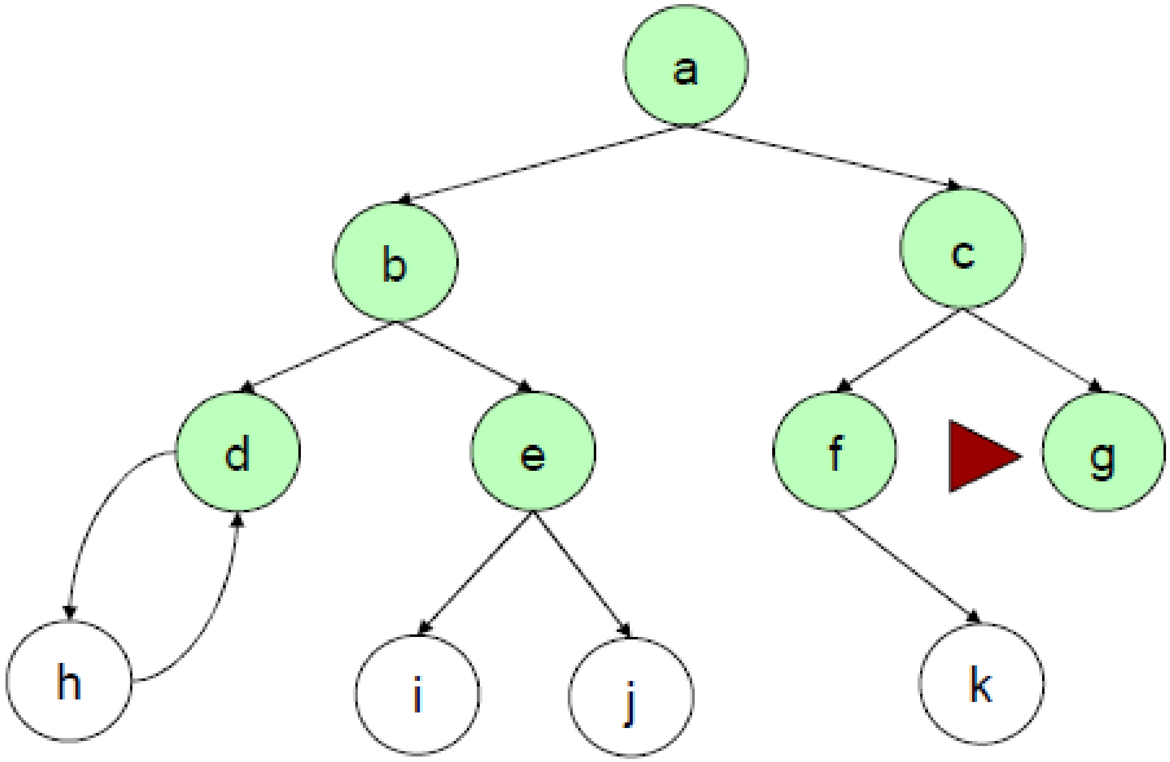
$Abertos$: $[e, f, g, h]$
Fechados: $[a, b, c, d\ ]$

Busca em largura



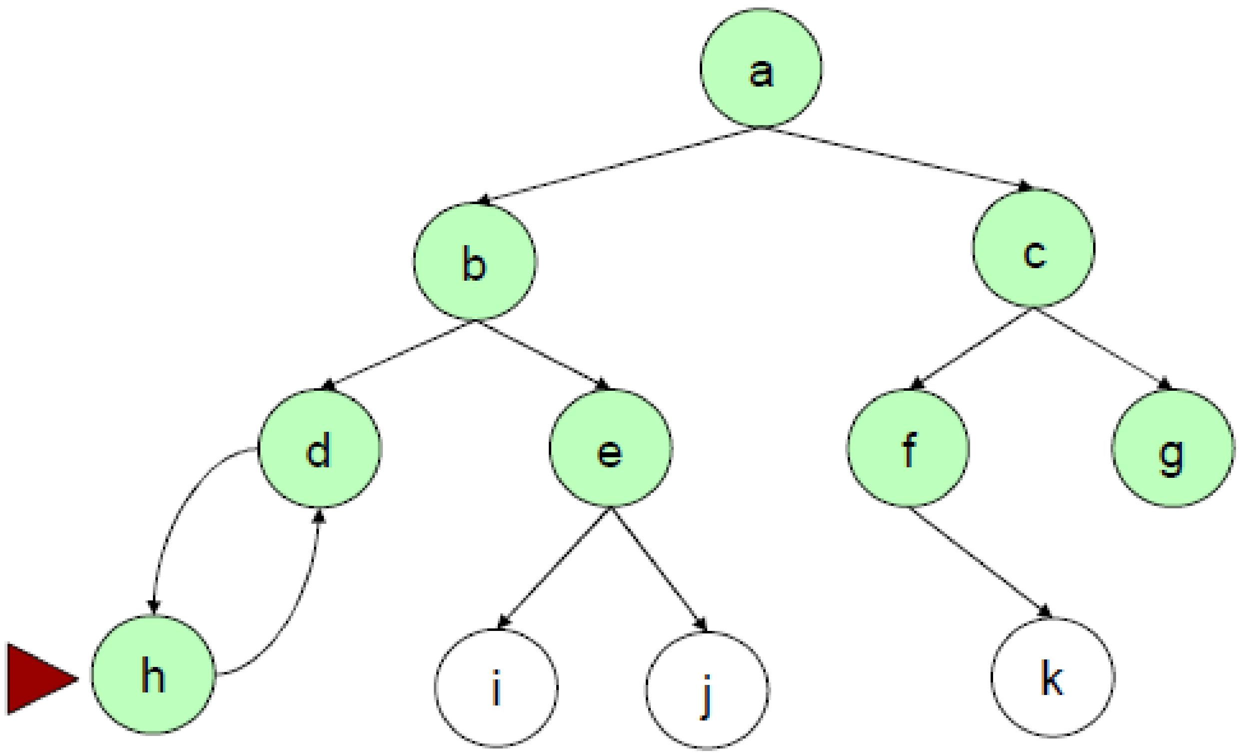
$Abertos$: $[f, g, h, i, j]$
Fechados: $[a, b, c, d, e]$

Busca em largura



$Abertos$: $[g, h, i, j, k]$
Fechados: $[a, b, c, d\ , e, f]$

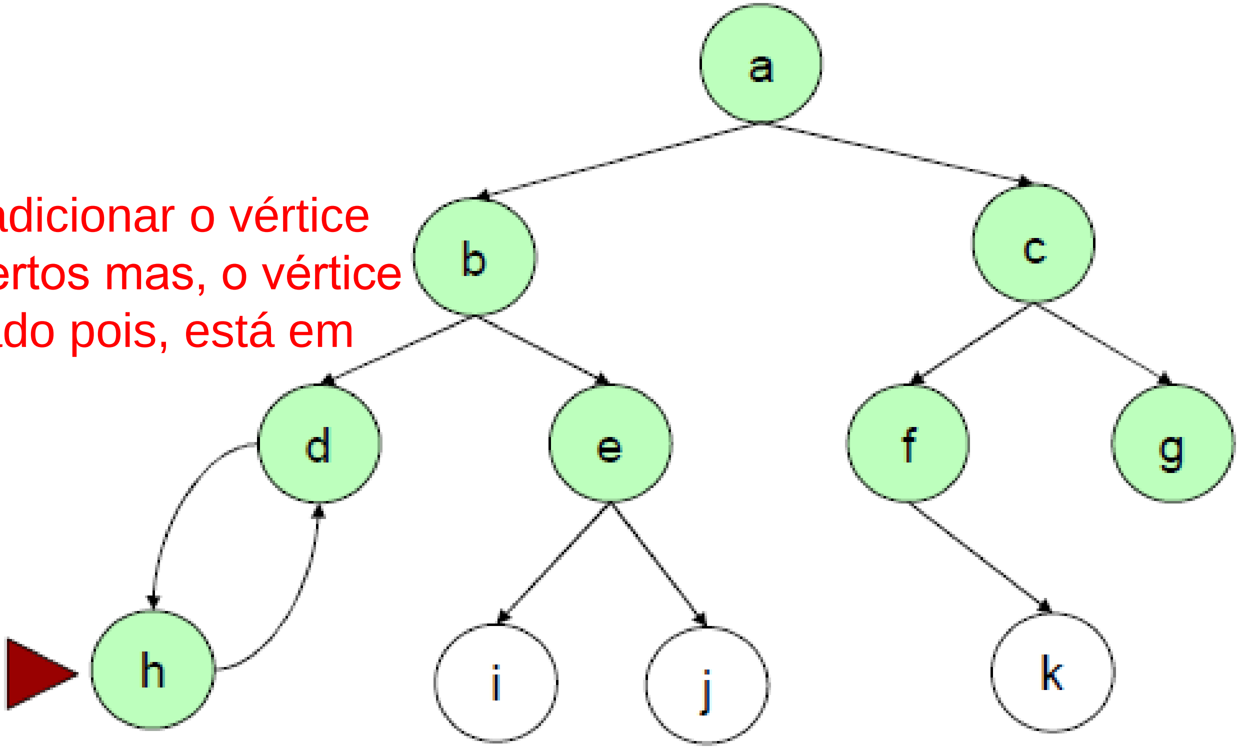
Busca em largura



$Abertos$: $[h, i, j, k]$
Fechados: $[a, b, c, d\ , e, f\ , g]$

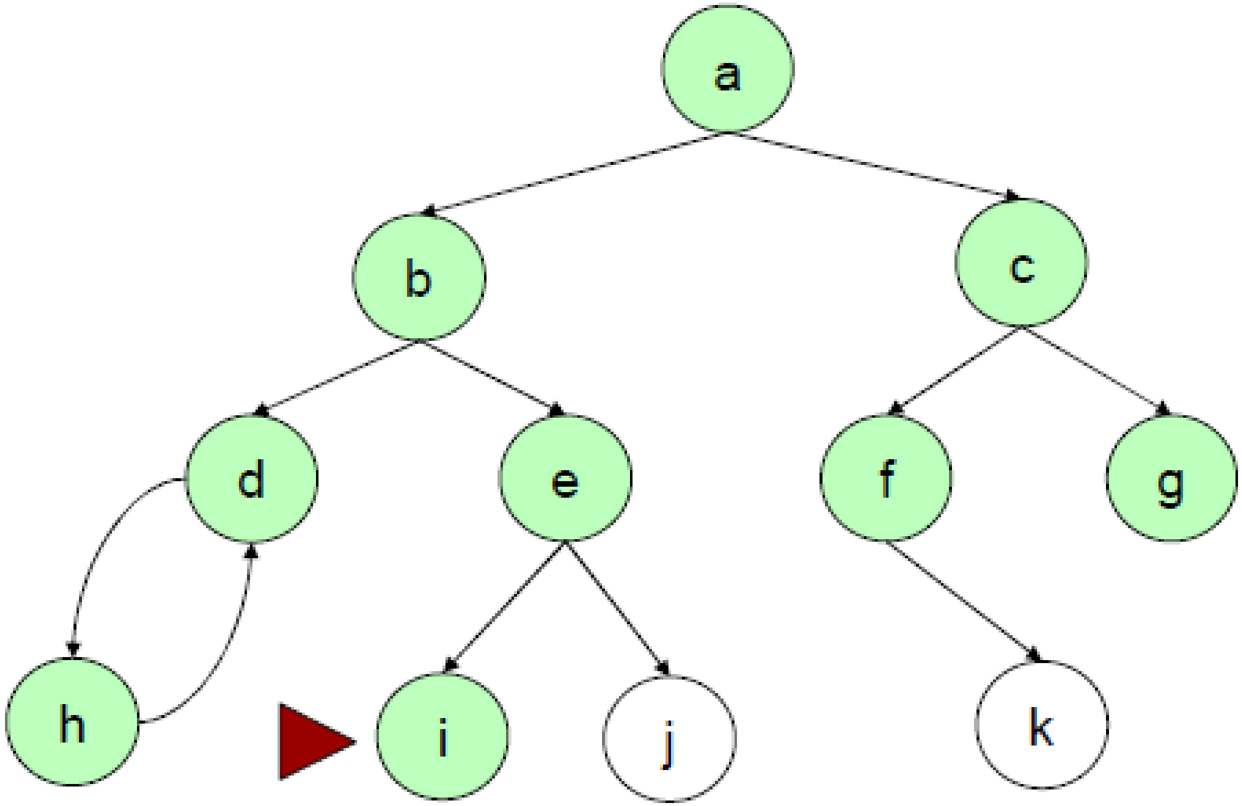
Busca em largura

Ele tenta adicionar o vértice “d” em abertos mas, o vértice já foi visitado pois, está em fechados



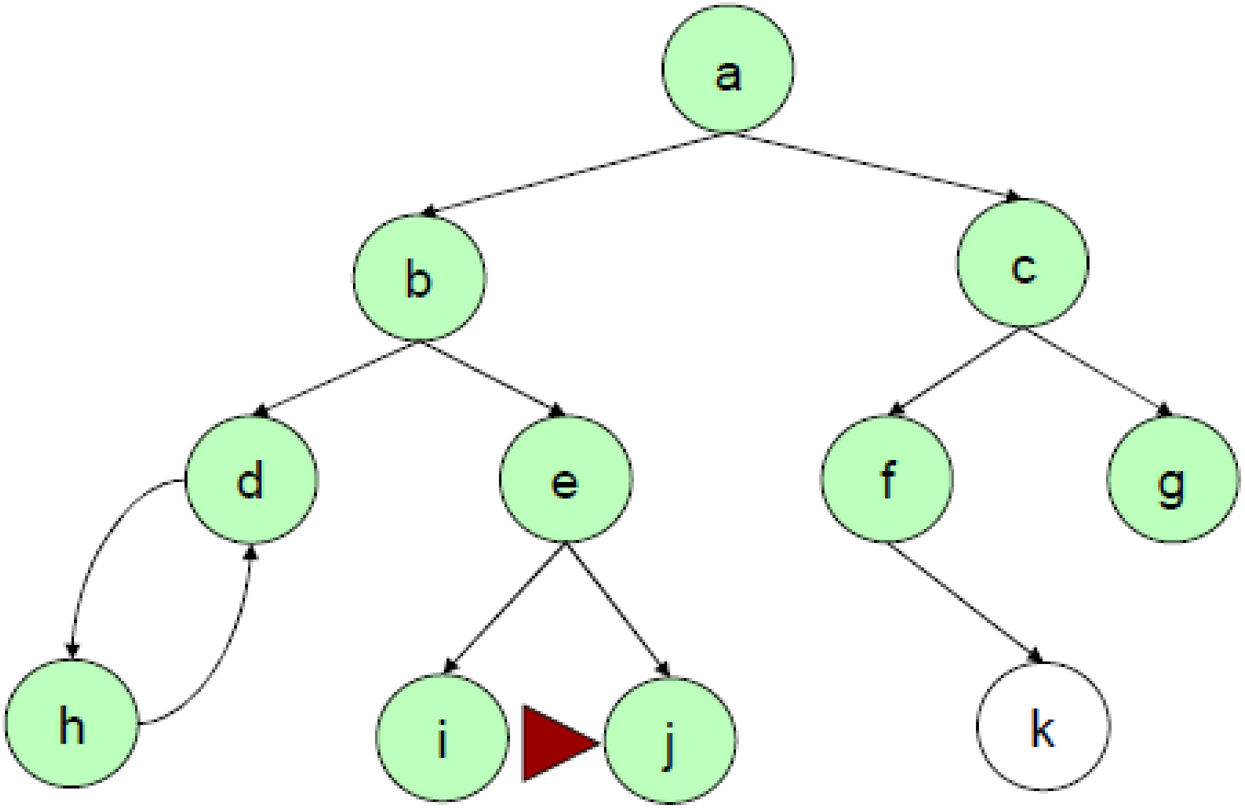
$Abertos$: $[h, i, j, k]$
Fechados: $[a, b, c, d\ , e, f\ , g]$

Busca em largura



$Abertos$: $[i, j, k]$
Fechados: $[a, b, c, d\ , e, f\ , g, h]$

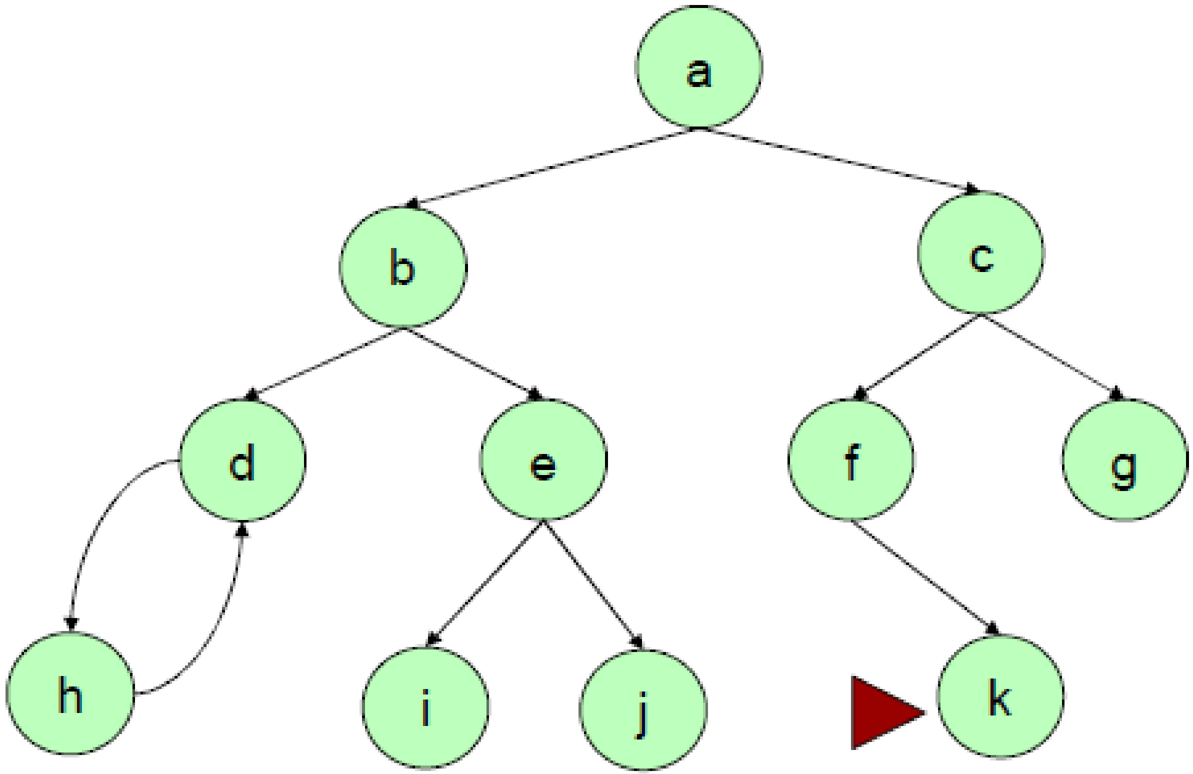
Busca em largura



$Abertos$: $[j, k]$

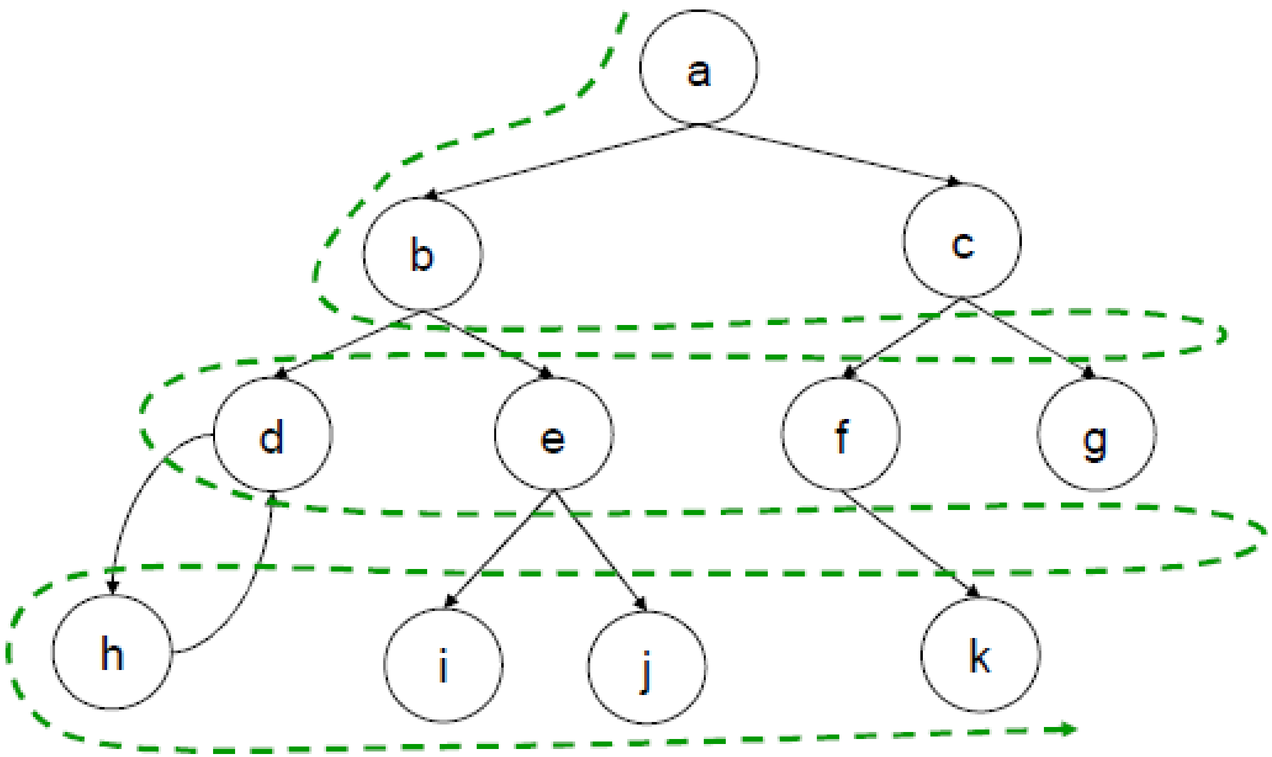
Fechados: $[a, b, c, d\ , e, f\ , g, h, i]$

Busca em largura



$Abertos$: $[k]$
Fechados: $[a, b, c, d\ , e, f\ , g, h, i, j]$

Busca em largura



Ordem de visita partindo do vértice a: a,b,c,d,e,f,g,h,i,j,k

$Abertos$: [] Fim da pesquisa! Abertos está vazia!!
Fechados: $[a, b, c, d\ , e, f\ , g, h, i, j, k]$

```python
def breadth_first_search(graph,initial):
    from Queue import queue
    frontier = queue()
    frontier.enqueue(initial)
    explored = list()
    while not frontier.isEmpty():
        curr = frontier.dequeue()
        explored.append(curr)
        for child,weight in reversed(graph.successors(curr)):
            if child not in frontier and child not in explored:
                frontier.enqueue(child)
    return explored
```

Caminhos Mais Curtos

- Os grafos podem ser utilizados para representar a estrutura rodoviária de um estado ou de um país, com os vértices representando cidades e as arestas representando trechos de rodovia

- As arestas podem então ter ponderações podem ser a distância entre as duas cidades interligadas pela aresta, ou o tempo médio necessário para percorrer a referida seção da rodovia ou mesmo o custo de combustível

- Um motorista que queira ir da cidade *A* para a cidade *B* estaria interessado em ter as respostas para:
  - Existe um caminho que vai de *A* para *B?*
  - Havendo mais de um caminho de *A* para *B,* qual o caminho mais curto?

Algoritmo de Dijkstra

- **1º passo:** iniciam-se os valores:

```
para todo v ∈ V[G]
     d[v] ← ∞
     π[v] ← -1
d[s] ← 0
```

Vetor de distâncias

Vetor de vértices prévios

Coloca da distância do vértice origem (s) com 0

V[G] é o conjunto de vértices(v) que formam o Grafo G. d[v] é o vetor de distâncias de s até cada v. Admitindo-se a pior estimativa possível, o caminho infinito.
π[v] identifica o vértice de onde se origina uma conexão até v de maneira a formar um caminho mínimo.

Algoritmo de Dijkstra

- **1º passo:** iniciam-se os valores:

```
para todo v ∈ V[G]
     d[v] ← ∞
     π[v] ← -1
d[s] ← 0
```

Vetor de distâncias

Vetor de vértices prévios

Coloca da distância do vértice origem (s) com 0

- **2º passo:** Incluir numa fila de prioridade (Q) o conjunto dos pares (distância, vértice)

```
Q ← V[G],d
```

Algoritmo de Dijkstra

- **1º passo:** iniciam-se os valores:

```
para todo v ∈ V[G]
      d[v] ← ∞   Vetor de distâncias
      π[v] ← -1  Vetor de vértices prévios
d[s] ← 0   Coloca da distância do vértice origem (s)  com 0
```

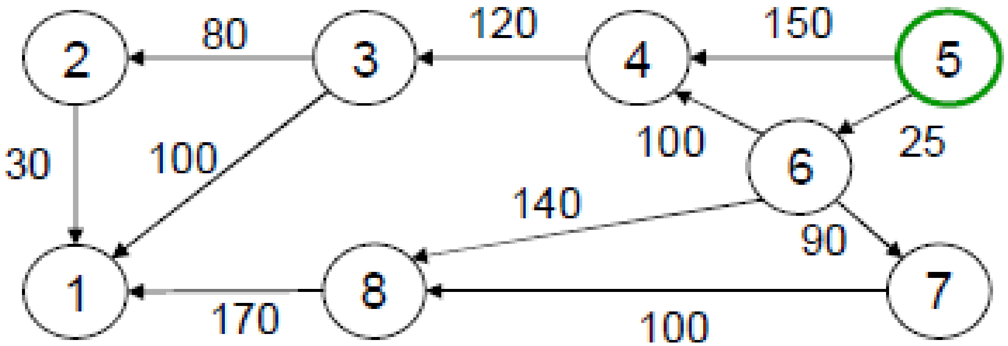
- **2º passo:** Incluir numa fila de prioridade (Q) o conjunto dos pares (distância, vértice)

```
Q ← V[G],d
```

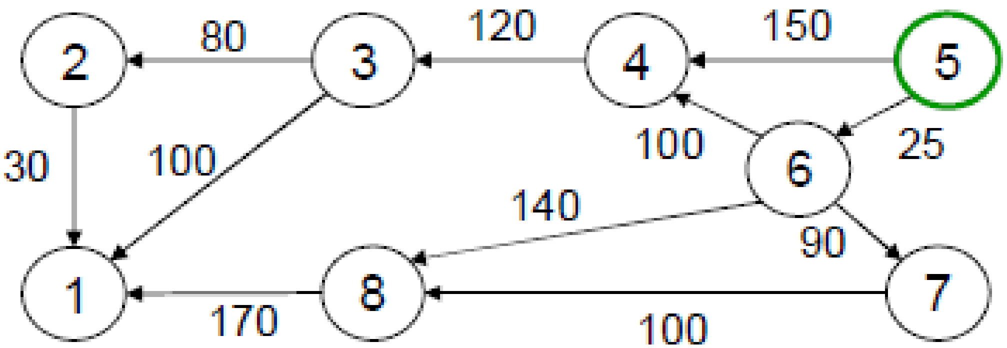
- **3º passo:** Execute a rotina

```
enquanto Q ≠ ϕ
          u ← extrair-mín(Q)  //Q ← Q - {u}
          para cada v adjacente a u
               se d[v] > d[u] + peso(u, v)
                  então d[v] ← d[u] + peso(u, v)
                        π[v] ← u
                        atualize (v,d[v] ) em Q
```

Algoritmo de Dijkstra



Algoritmo de Dijkstra



Primeiramente criamos uma fila de prioridade com todos os vértices com o par (distância, vértice) = $(\infty, vértice)$

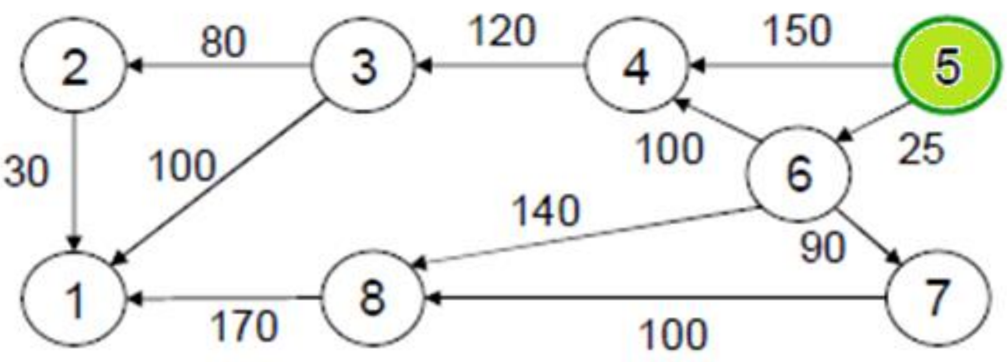
$$Q = [(\infty, 1), (\infty, 2), (\infty, 3), (\infty, 4), (\infty, 5), (\infty, 6), (\infty, 7), (\infty, 8)]$$

Depois, inicializamos, um vetor, com as distâncias, com o valor, infinito. O índice, corresponde ao vértice.

Dist

	∞	∞	∞	∞	∞	∞	∞	∞
0	1	2	3	4	5	6	7	8

Algoritmo de Dijkstra



Vamos supor, que o vértice origem, seja o vértice 5. Primeiramente atualizamos o par (distância, vértice origem) = $(0, 5)$

$$Q = [(0,5), (\infty, 1), (\infty, 2), (\infty, 3), (\infty, 4), (\infty, 6), (\infty, 7), (\infty, 8)]$$

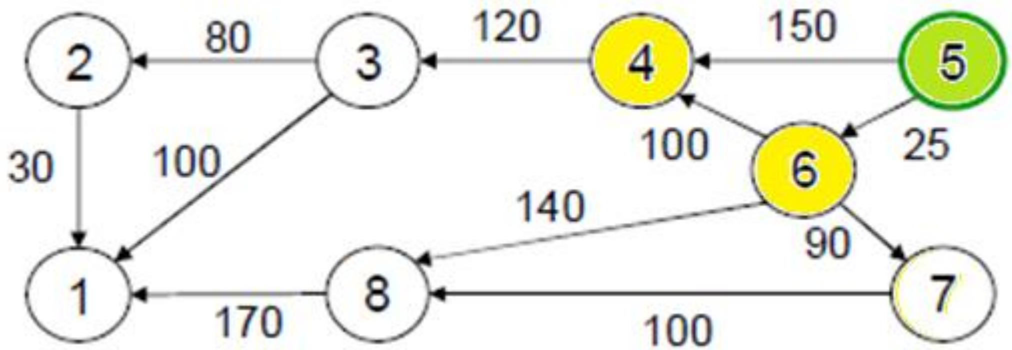
Observe, que o par $(0,5)$ é o primeiro da fila de prioridade!

Depois, atualizamos o vetor de distâncias, com o valor, 0 na posição do vértice 5

Dist

	∞	∞	∞	∞	**0**	∞	∞	∞
0	1	2	3	4	5	6	7	8

Algoritmo de Dijkstra



Escolhemos, o vértice que está na frente da fila de prioridade, vértice 5, e excluímos da fila de prioridade

$$Q = [(0{,}5), (\infty, 1), (\infty, 2), (\infty, 3), (\infty, 4), (\infty, 6), (\infty, 7), (\infty, 8)]$$

$$Q = [(\infty, 1), (\infty, 2), (\infty, 3), (\infty, 4), (\infty, 6), (\infty, 7), (\infty, 8)]$$

$(0{,}5)$: Agora, vamos obter todos os sucessores do vértice 5, e alterar os seus valores (distância, vértice), com as distância acumuladas, na fila de prioridade. Detalhe, só atualizamos se a distância acumulada for menor que a distância que está na fila de prioridade.

$$distância\ cumulada \mathrel{+}= disância\ entre\ os\ vértices$$

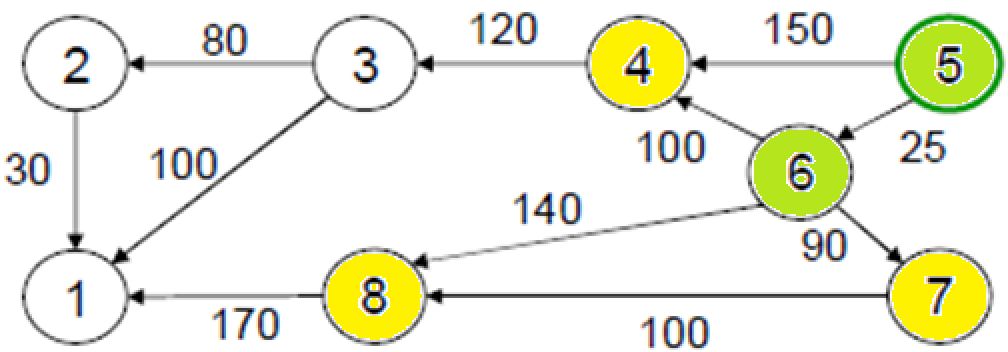
$$Q = [(25{,}6), (150{,}4), (\infty, 1), (\infty, 2), (\infty, 3), (\infty, 7), (\infty, 8)]$$

Depois, atualizamos o vetor de distâncias, com o valores das distâncias para cada vértice adjacente

Dist

	∞	∞	∞	150	0	25	∞	∞
0	1	2	3	4	5	6	7	8

Algoritmo de Dijkstra



Novamente escolhemos, o vértice que está na frente da fila de prioridade, vértice 6, excluímos da fila de prioridade.

$$Q = [(25{,}6), (150{,}4), (\infty, 1), (\infty, 2), (\infty, 3), (\infty, 7), (\infty, 8)]$$

$$Q = [(150{,}4), (\infty, 1), (\infty, 2), (\infty, 3), (\infty, 7), (\infty, 8)]$$

$(25{,}6)$: Agora, vamos obter todos os sucessores do vértice 6, e alterar os seus valores (distância, vértice), com as distância acumuladas, na fila de prioridade. Detalhe, só atualizamos se a distância acumulada for menor que a distância que está na fila $\quad distância\ cumulada \mathrel{+}= disância\ entre\ os\ vértices$

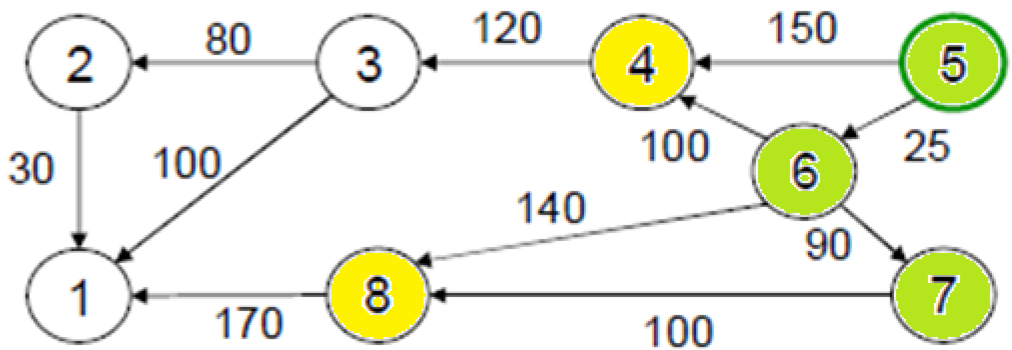
$$Q = [(115{,}7), (125{,}4), (165{,}8), (\infty, 1), (\infty, 2), (\infty, 3)]$$

Depois, atualizamos o vetor de distâncias, com o valores das distâncias para cada vértice adjacente

Dist

	∞	∞	∞	125	**0**	**25**	115	165
0	1	2	3	4	5	6	7	8

Algoritmo de Dijkstra



Novamente escolhemos, o vértice que está na frente da fila de prioridade, vértice 7, excluímos da fila de prioridade

$$Q = [(\color{red}{115},7),(125,4),(165,8),(\infty,1),(\infty,2),(\infty,3)]$$

$$Q = [(125,4),(165,8),(\infty,1),(\infty,2),(\infty,3)]$$

$(115,7)$: Agora, vamos obter todos os sucessores do vértice 7, e alterar os seus valores (distância, vértice), com as distância acumuladas, na fila de prioridade. Detalhe, só atualizamos se a distância acumulada for menor que a distância que está na fila

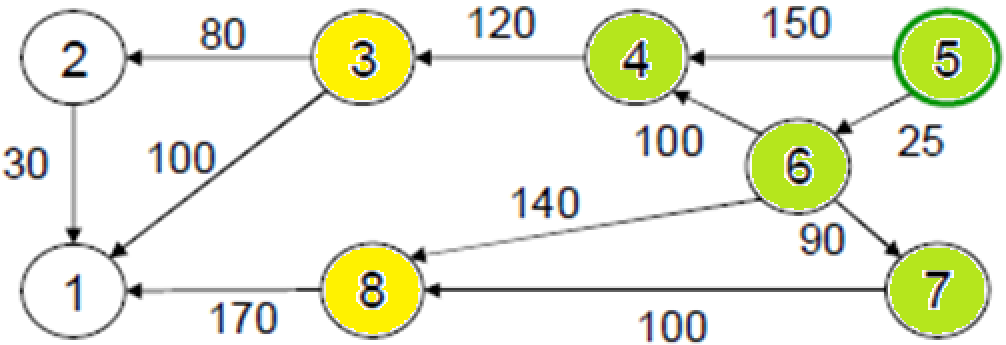
$$Q = [(125,4),(165,8),(\infty,1),(\infty,2),(\infty,3)]$$

Observe que não atualizamos a distância, do vértice 8, pois a distância acumulada (215) é maior do que a distância na fila de prioridade (165)

Dist

0	1	2	3	4	5	6	7	8
	∞	∞	∞	125	**0**	**25**	115	165

Algoritmo de Dijkstra



Novamente escolhemos, o vértice que está na frente da fila de prioridade, vértice 4, excluímos da fila de prioridade

$$Q = [(125{,}4), (165{,}8), (\infty, 1), (\infty, 2), (\infty, 3)]$$

$$Q = [(165{,}8), (\infty, 1), (\infty, 2), (\infty, 3)]$$

$(125{,}4)$: Agora, vamos obter todos os sucessores do vértice 4, e alterar os seus valores (distância, vértice), com as distância acumuladas, na fila de prioridade. Detalhe, só atualizamos se a distância acumulada for menor que a distância que está na fila

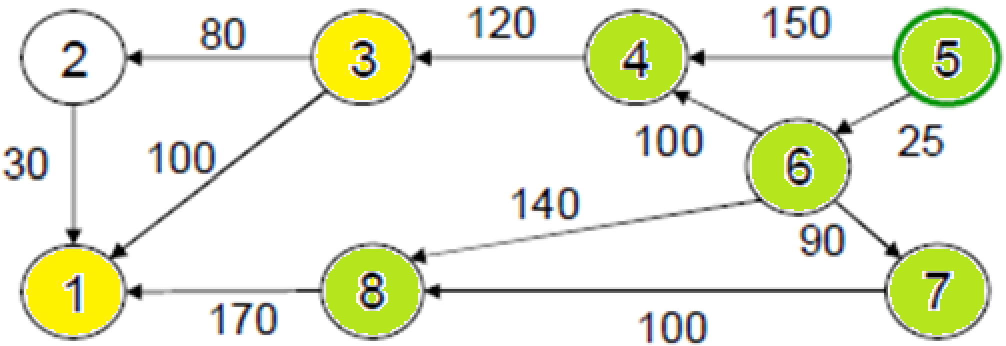
$$Q = [(165{,}8), (245{,}3), (\infty, 1), (\infty, 2)]$$

Atualizamos as distâncias no vetor de distâncias, com a distância acumulada se esta, for menor que a que esta na fila de prioridade

Dist

0	1	2	3	4	5	6	7	8
	∞	∞	245	125	0	25	115	165

Algoritmo de Dijkstra



Novamente escolhemos, o vértice que está na frente da fila de prioridade, vértice 8, excluímos da fila de prioridade e, adicionamos na lista de

$$Q = [(165{,}8), (245{,}3), (\infty, 1), (\infty, 2)]$$

$$Q = [(245{,}3), (\infty, 1), (\infty, 2)]$$

$(165{,}8)$: Agora, vamos obter todos os sucessores do vértice 8, e alterar os seus valores (distância, vértice), com as distância acumuladas, na fila de prioridade. Detalhe, só atualizamos se a distância acumulada for menor que a distância que está na fila

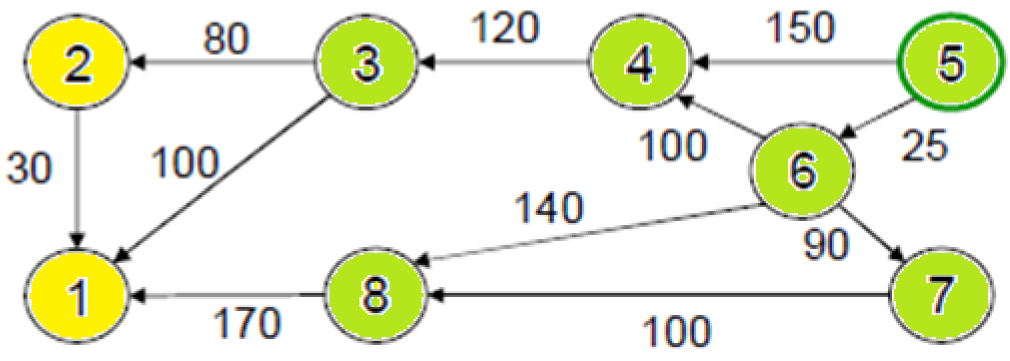
$$Q = [(245{,}3), (335{,}1), (\infty, 2)]$$

Atualizamos as distâncias no vetor de distâncias , com a distância acumulada se esta, for menor que a que esta na fila de prioridade

Dist

	335	∞	245	125	0	25	115	165
0	1	2	3	4	5	6	7	8

Algoritmo de Dijkstra



Novamente escolhemos, o vértice que está na frente da fila de prioridade, vértice 3, excluímos da fila de prioridade e, adicionamos na lista de

$$Q = [(245{,}3), (335{,}1), (\infty, 2)]$$

$$Q = [(335{,}1), (\infty, 2)]$$

$(245{,}3)$: Agora, vamos obter todos os sucessores do vértice 3, e alterar os seus valores (distância, vértice), com as distância acumuladas, na fila de prioridade. Detalhe, só atualizamos se a distância acumulada for menor que a distância que está na fila

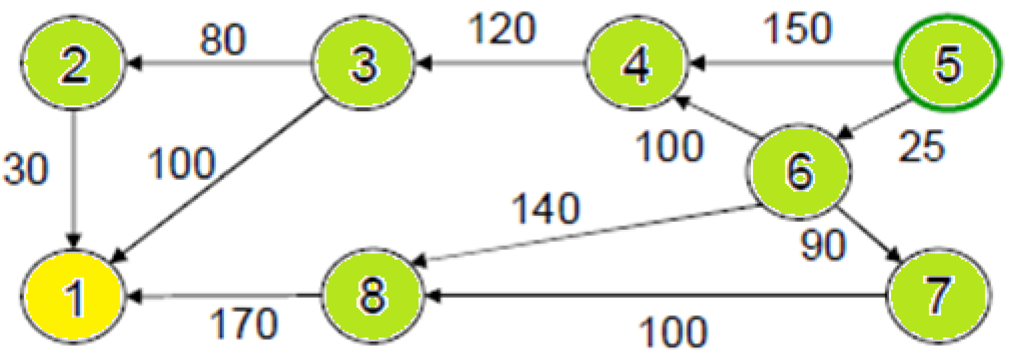
$$Q = [(325{,}2), (335{,}1)]$$

Atualizamos as distâncias no vetor de distâncias , com a distância acumulada se esta, for menor que a que esta na fila de prioridade

Dist

	335	325	245	125	0	25	115	165
0	1	2	3	4	5	6	7	8

Algoritmo de Dijkstra



Novamente escolhemos, o vértice que está na frente da fila de prioridade, vértice 2, excluímos da fila de prioridade e, adicionamos na lista de

$$Q = [(325,2),(335,1)]$$

$$Q = [(335,1)]$$

$(325,2)$ **:** Agora, vamos obter todos os sucessores do vértice 2, e alterar os seus valores (distância, vértice), com as distância acumuladas, na fila de prioridade. Detalhe, só atualizamos se a distância acumulada for menor que a distância que está na fila

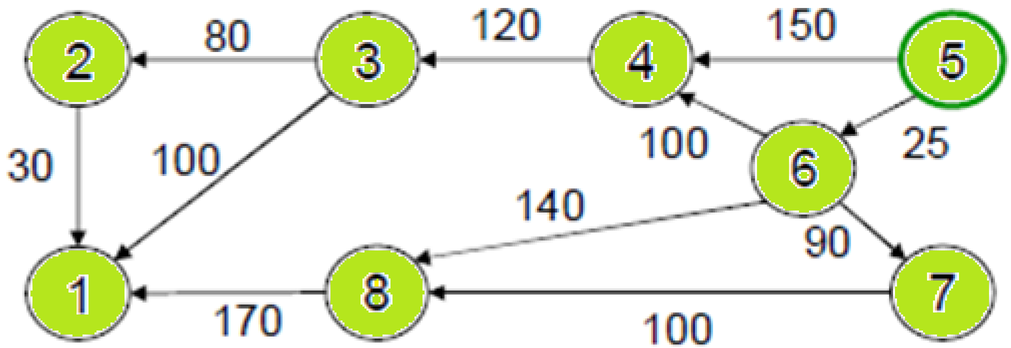
$$Q = [(335,1)]$$

Atualizamos as distâncias no vetor de distâncias, como a distância acumulada (355) é menor que a da fila de prioridade (335) não atualizamos.

Dist

	335	325	245	125	0	25	115	165
0	1	2	3	4	5	6	7	8

Algoritmo de Dijkstra



Novamente escolhemos, o vértice que está na frente da fila de prioridade, vértice 1, excluímos da fila de prioridade e, adicionamos na lista de

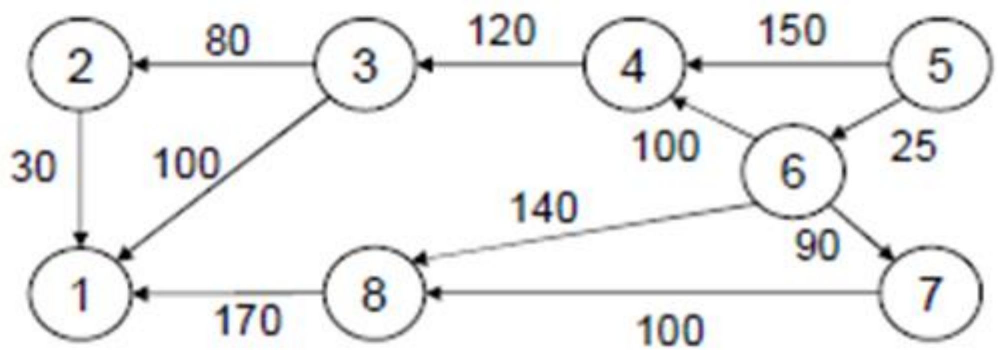
$$Q = [(335,1)]$$

$$Q = [\ ]$$

$(355,1)$: Agora, vamos obter todos os sucessores do vértice 1. Não existe mais sucessores e a fila de prioridade está vazia, logo, o procedimento termina!

Dist

	335	325	245	125	**0**	**25**	115	165
0	1	2	3	4	5	6	7	8



Iteração	S	Vértice Selecionado	DIST							
			1	2	3	4	5	6	7	8
inicial		5	+∞	+∞	+∞	150	0	25	+∞	+∞
1	5	6	+∞	+∞	+∞	125	0	25	115	165
2	5,6	7	+∞	+∞	+∞	125	0	25	115	165
3	5,6,7	4	+∞	+∞	245	125	0	25	115	165
4	5,6,7,4	8	335	+∞	245	125	0	25	115	165
5	5,6,7,4,8	3	335	325	245	125	0	25	115	165
6	5,6,7,4,8,3	2	335	325	245	125	0	25	115	165
final	5,6,7,4,8,3,2	1	335	325	245	125	0	25	115	165

```python
def dijkstra(graph, start):
    from PriorityQueue import PriorityQueue as queue
    # Initialize both vertices, distance and previous list
    vertices = graph.vertices()
    n = len(vertices) # vertices list lenght
    dist = [float('inf')] * n
    previous = [None] * n # for path
    i = vertices.index(start)
    dist[i] = 0          # Distance from start to current

    Q = queue()
    # Initialize PriorityQueue (0,start) another all( inf, current)
    for v in vertices:
        Q.insert(tuple([float('inf'),v]))
    i = Q.index(start)
    Q.update(i,tuple([0,start]))
```

```
Q.update(i,tuple([0,start]))
while not Q.isEmpty():     # main loop
     d,u = Q.delMin()  # tuple(distance,vertex)
     i = vertices.index(u)
     dist[i] = d  # distance
     for child,weight in graph.successors(u): # where v has not yet been r
          k = vertices.index(child)
          dist_between = weight  #  dist_between u v
          alt = dist[i] + dist_between
          j = Q.index(child)
          dist[k],v = Q.get(j) # Q.queue[j] = (distance,vertex)
          if alt < dist[k]:
               dist[k] = alt
               Q.update(j,tuple([dist[k],child])) # Updtade tuple in Queue
               previous[k] = u

return previous,dist
```