CIC110 — Análise e Projeto de Algoritmos I

Pedro H. D. B. Hokama

Originalmente elaborado pelos professores:
Cid C. de Souza
Cândida N. da Silva
Orlando Lee
Pedro J. de Rezende

Qualquer problema ou erro é de minha responsabilidade.

23 de outubro de 2018

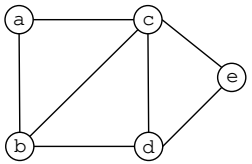
Grafos: Noções Básicas e Representação

Definição de Grafo

- Um *grafo* $G = (V, E)$ é dado por dois conjuntos finitos: o conjunto de *vértices* V e o conjunto de *arestas* E.
- Cada aresta de E é um par não ordenado de vértices de V.
- **Exemplo:**

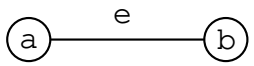
$$V = \{a, b, c, d, e\}$$

$$E = \{(a, b), (a, c), (b, c), (b, d), (c, d), (c, e), (d, e)\}$$



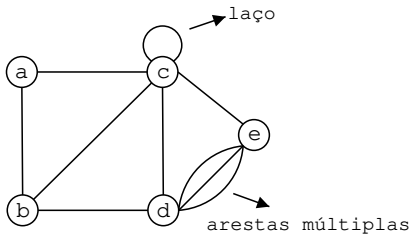
Definição de Grafo

- Dada uma aresta $e = (a, b)$, dizemos que os vértices a e b são os *extremos* da aresta e e que a e b são vértices *adjacentes*.
- Podemos dizer também que a aresta e é *incidente* aos vértices a e b, e que os vértices a e b são incidentes à aresta e.



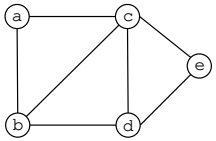
Grafo Simples

- Dizemos que um grafo é *simples* quando não possui laços ou arestas múltiplas.
- Um *laço* é uma aresta com ambos os extremos em um mesmo vértice e *arestas múltiplas* são duas ou mais arestas com o mesmo par de vértices como extremos.
- **Exemplo:**



Tamanho do Grafo

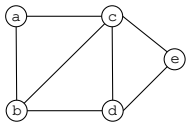
- Denotamos por $|V|$ e $|E|$ a cardinalidade dos conjuntos de vértices e arestas de um grafo G, respectivamente.
- No exemplo abaixo temos $|V| = 5$ e $|E| = 7$.



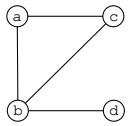
O *tamanho* do grafo G é dado por $|V| + |E|$.

Subgrafo e Subgrafo Gerador

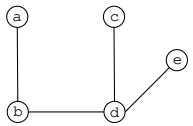
- Um *subgrafo* $H = (V', E')$ de um grafo $G = (V, E)$ é um grafo tal que $V' \subseteq V$, $E' \subseteq E$.
- Um *subgrafo gerador* de G é um subgrafo H com $V' = V$.
- **Exemplo:**



Grafo G



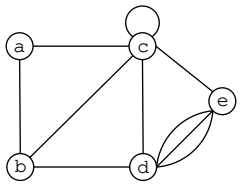
Subgrafo não gerador



Subgrafo gerador

Grau de um vértice

- O *grau* de um vértice v, denotado por $d(v)$ é o número de arestas incidentes a v, com laços contados duas vezes.
- **Exemplo:**



d(a)=2
d(b)=3
d(c)=6
d(d)=5
d(e)=4

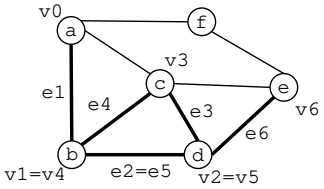
Teorema

Para todo grafo $G = (V, E)$ temos:

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

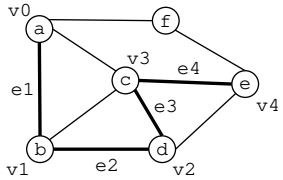
Caminhos em Grafos

- Um *caminho* P de v_0 a v_n no grafo G é uma seqüência finita e não vazia $(v_0, e_1, v_1, \ldots, e_n, v_n)$ cujos elementos são alternadamente vértices e arestas e tal que, para todo $1 \leq i \leq n$, v_{i-1} e v_i são os extremos de e_i.
- O *comprimento* do caminho P é dado pelo seu número de arestas, ou seja, n.
- **Exemplo:**

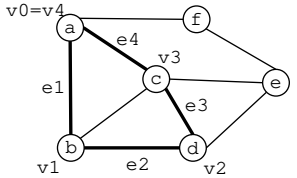


Caminhos Simples e Ciclos

- Um *caminho simples* é um caminho em que não há repetição de vértices e nem de arestas na sequência.
- Um *ciclo* ou *caminho fechado* é uma caminho em que $v_0 = v_n$.
- **Exemplo:**



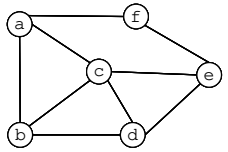
Caminho Simples



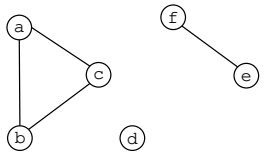
Ciclo

Grafo Conexo

- Dizemos que um grafo é *conexo* se, para qualquer par de vértices u e v de G, existe um caminho de u a v em G.
- Quando o grafo G não é conexo, podemos particionar em *componentes conexos*. Dois vértices u e v de G estão no mesmo componente conexo de G se há caminho de u a v em G.
- **Exemplo:**



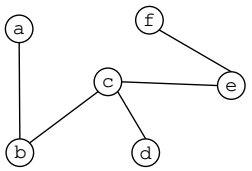
Conexo



Não-conexo com
3 componentes conexos

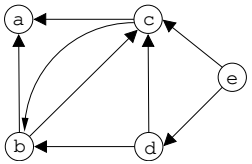
Árvore

- Um grafo G é uma *árvore* se é conexo e não possui ciclos (acíclico). Ou equivalentemente, G é árvore se:
  - É conexo com $|V| - 1$ arestas.
  - É conexo e a remoção de qualquer aresta desconecta o grafo (minimal conexo).
  - Para todo par de vértices u, v de G, existe exatamente um caminho de u a v em G.
- **Exemplo:**



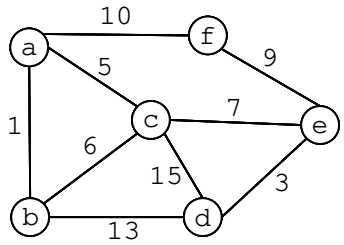
Grafo Orientado

- As definições que vimos até agora são para grafos *não orientados*.
- Um *grafo orientado* é definido de forma semelhante, com a diferença que as arestas (às vezes chamadas de arcos) são dadas por pares ordenados.
- **Exemplo:**



Grafo Ponderado

- Um grafo (orientado ou não) é *ponderado* se a cada aresta e do grafo está associado um valor real $c(e)$, o qual denominamos *custo (ou peso)* da aresta.
- **Exemplo:**



Algoritmos em Grafos - Motivação

- Grafos são estruturas abstratas que podem modelar diversos problemas do mundo real.
- Por exemplo, um grafo pode representar conexões entre cidades por estradas ou uma rede de computadores.
- O interesse em estudar algoritmos para problemas em grafos é que conhecer algoritmo para um único problema em grafos pode significar conhecer algoritmos para diversos problemas reais.

Representação Interna de Grafos

- A complexidade dos algoritmos para solução de problemas modelados por grafos depende fortemente da sua representação interna.
- Existem duas representações canônicas: matriz de adjacências e lista de adjacências.
- O uso de uma ou outra num determinado algoritmo depende da natureza das operações que ditam a complexidade do algoritmo.
- Outras representações podem ser utilizadas mas essas duas são as mais utilizadas por sua simplicidade.
- Para alguns problemas em grafos o uso de estruturas de dados adicionais são fundamentais para o projeto de algoritmos eficientes.

Matriz de adjacências

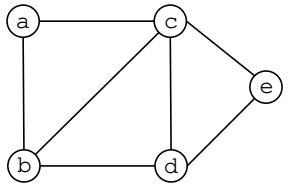
- Um grafo simples $G = (V, E)$, orientado ou não, pode ser representado internamente por uma estrutura de dados chamada *matriz de adjacências*.
- A matriz de ajacências é uma matriz quadrada A de ordem $|V|$, cujas linhas e colunas são indexadas pelos vértices em V, e tal que:

$$A[i, j] = 1 \text{ se } (i, j) \in E \text{ e } 0 \text{ caso contrário.}$$

- Note que se G é não-orientado, então a matriz A correspondente é simétrica.
- Esta definição, para grafos simples, pode ser generalizada: basta fazer $A[i, j]$ ser um registro que contém informações sobre a adjacência dos vértices i e j como multiplicidade ou custo da aresta (i, j).

Matriz de adjacências

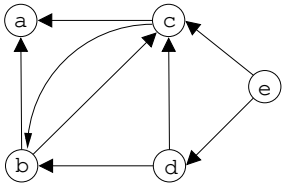
- Exemplo de um grafo não-orientado simples e a matriz de adjacências correspondente.



	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	1	0
c	1	1	0	1	1
d	0	1	1	0	1
e	0	0	1	1	0

Matriz de adjacências

- Exemplo de um grafo orientado simples e a matriz de adjacências correspondente.



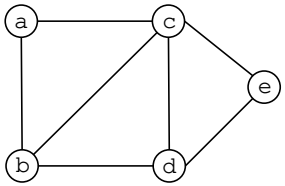
	a	b	c	d	e
a	0	0	0	0	0
b	1	0	1	0	0
c	1	1	0	0	0
d	0	1	1	0	0
e	0	0	1	1	0

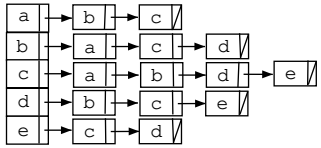
Lista de adjacências

- Uma *lista de adjacências* que representa um grafo simples G é um vetor L indexado pelos vértices em V de forma que, para cada $v \in V$, $L[v]$ é um apontador para o início de uma lista ligada dos vértices que são adjacentes a v em G.
- O tamanho da lista $L[v]$ é precisamente o grau $d(v)$.
- Se G for um grafo não orientado, cada aresta (i, j) é representada duas vezes: uma na lista de adjacências de i e outra na de j.
- Se G for um grafo orientado, a lista ligada $L(v)$ contém apenas os elementos pós-adjacentes (ou pré-adjacentes) a v em G. Assim, cada aresta (ou arco) é representada exatamente uma vez.
- Também podemos usar uma lista ligada para representar grafos não simples ou ponderados: basta que cada nó da lista seja um registro que contenha as demais informações sobre a adjacência de i e j como multiplicidade e custo (ou peso).

Lista de adjacências

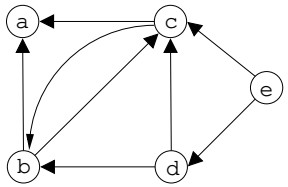
- Exemplo de um grafo não-orientado e a lista de adjacências correspondente.

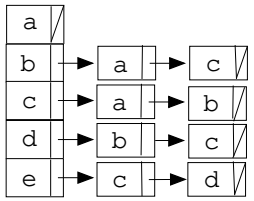




Lista de adjacências

Exemplo de um grafo orientado e a lista de adjacências correspondente.





Matriz $\times$ Lista de adjacências

- Cada uma dessas estruturas tem suas vantagens:
  - Matrizes de adjacências são mais adequadas quando queremos descobrir rapidamente se uma dada aresta está ou não presente num grafo, ou os atributos dela.
  - Por outro lado, lista de adjacências são mais adequadas quando queremos determinar todos os vértices adjacentes a um dado vértice.
  - Matrizes de adjacências ocupam espaço proporcional a $|V|^2$ e são, portanto, mais adequadas a grafos densos ($|E| = \Theta(|V|^2)$).
  - Listas de adjacências ocupam espaço proporcional a $|E|$, sendo mais adequadas a grafos esparsos ($|E| = \Theta(|V|)$).
- Em alguns casos pode ser útil ter as duas estruturas simultaneamente.

Buscas em grafos

Buscas em grafos

- Em muitas aplicações em grafos é necessário percorrer rapidamente o grafo visitando-se todos os seus vértices.
- Para que isso seja feito de modo sistemático e organizado são utilizados algoritmos de busca, semelhantes àqueles que já foram vistos para percursos em árvores na disciplina de Estruturas de Dados.
- As buscas são usadas em diversas aplicações para determinar informações relevantes sobre a estrutura do grafo de entrada.
- Além disso, alterações, muitas vezes pequenas, nos algoritmos de busca permitem resolver de forma eficiente problemas mais complexos definidos sobre grafos.
- São dois os algoritmos básicos de busca em grafos: a busca em largura e a busca em profundidade.

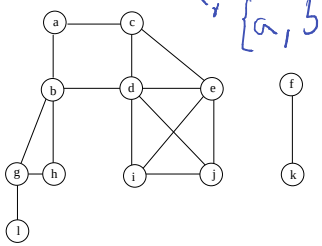
Busca em largura

- Diremos que um vértice u é **alcançável** a partir de um vértice v de um grafo G se existe um caminho de v para u em G.
- Definição: um vértice u está a uma **distância** k de um vértice v se k é o comprimento do menor caminho que começa em v e termina em u.
  Se u **não é alcançável** a partir de v, então arbitra-se que a distância entre eles é **infinita**.

Uma busca em largura ou BFS (do inglês *breadth first search*) em um grafo $G = (V, E)$ é um método em que, partindo-se de um vértice especial u denominado *raiz da busca*, percorre-se G visitando-se todos os vértices alcançáveis a partir de u em **ordem crescente de distância**.
Nota: a ordem em que os vértices equidistantes de u são percorridos é irrelevante e depende da forma como as adjacências dos vértices são armazenadas e percorridas.

Busca em largura

No grafo da figura abaixo, assumindo-se o vértice *a* como sendo a raiz da busca e que as listas de adjacências estão ordenadas alfabeticamente, a ordem de visitação dos vértices seria $\{a, b, c, d, g, h, e, i, j, f, k, l\}$. Os vértices f e k não seriam visitados porque não são alcançáveis a partir do vértice a.



Busca em largura

- A busca em largura é um exemplo interessante de aplicação de **filas**.
- O algoritmo descrito a seguir usa um critério de coloração para ir controlando os vértices do grafo que já foram visitados e/ou cujas listas de adjacências já foram exploradas durante a busca.
- Um vértice é identificado como **visitado** no momento em que a busca o atinge pela primeira vez.
- Uma vez visitado, este vértice poderá permitir que sejam atingidos outros vértices ainda não alcançados pela busca, o que é feito no momento em que se explora a sua vizinhança.
- Concluída esta operação, o vértice é dito estar **explorado**.

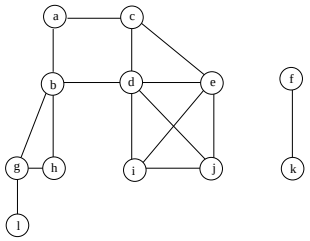
Busca em largura

- São listados abaixo os principais elementos do pseudo-código que descreve a busca em largura.
- o vetor `cor` contendo $|V|$ posições. O elemento `cor`$[i]$ será `branco` se o vértice i ainda não foi visitado, `cinza` se ele foi visitado mas a sua vizinhança não foi explorada e `preto` se o vértice i foi visitado e sua vizinhança já foi explorada.
- o vetor `dist` contendo $|V|$ posições. O elemento `dist`$[i]$ é inicializado com o valor infinito e armazenará a distância entre o vértice i e a raiz da busca.
- o vetor `pred` contendo $|V|$ posições. O elemento `pred`$[i]$ armazenará a informação sobre qual era o vértice cuja adjacência estava sendo explorada quando o vértice i foi visitado pela primeira vez.
- a fila Q armazena a lista dos vértices visitados cuja vizinhança ainda deve ser explorada, ou seja, aqueles de cor cinza.

Busca em largura

```
BFS(G,raiz)
▷ Q é o conjunto de vértices a serem explorados (cinzas)
1.  InicializaFila(Q);          ▷ inicializa fila como sendo vazia
2.  Para todo v ∈ V − {raiz} faça
3.      dist[v] ← ∞;   cor[v] ← branco;   pred[v] ← NULO;
4.  dist[raiz] ← 0;   cor[raiz] ← cinza;   pred[raiz] ← NULO;
5.  InsereFila(Q, raiz);
6.  Enquanto (!FilaVazia(Q)) faça
7.      RemoveFila(Q, u)        ▷ tirar u do conjunto Q
8.      Para todo v ∈ Adj[u] faça
9.          Se cor[v] = branco então
10.             dist[v] ← dist[u] + 1;   pred[v] ← u;
11.             cor[v] ← cinza;   InsereFila(Q, v);
12.     cor[u] ← preto;
13. Retorne(dist, pred).
```


Busca em largura: exemplo



Nota: o subgrafo $G_{\texttt{pred}}$ formado por $(V, E_{\texttt{pred}})$, onde $E_{\texttt{pred}} = \{(\texttt{pred}[v], v) : \forall v \in V\}$, é uma árvore, que é chamada de **árvore BFS**.

Busca em largura: complexidade

- Faz-se uma análise agregada onde é contado o *total* de operações efetuadas no laço das linhas 6 a 12 no *pior caso*.
- Supõe-se que o grafo é armazenado por listas de adjacências.
- Depois da inicialização da linha 3, nenhum vértice é colorido como branco. Logo todo vértice é inserido e removido no máximo uma vez e, portanto, o tempo total gasto nas operações de manutenção da fila Q é $O(|V|)$.
- A lista de adjacências de um vértice só é percorrida quando ele é removido da fila. Logo, ela é varrida no máximo uma vez. Como o comprimento total de todas as listas de adjacências é $\Theta(E)$, o tempo total gasto explorando estas listas é $\Theta(E)$.
- O gasto de tempo com as inicializações é claramente $\Theta(V)$. Assim, o tempo total de execução do algoritmo é $O(|V| + |E|)$.

Busca em profundidade

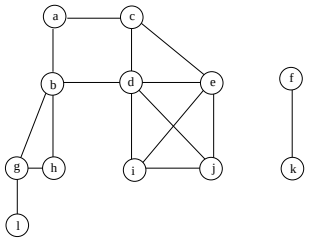
Intuitivamente, pode-se dizer que a estratégia de percorrer um grafo usando uma busca em profundidade ou DFS (do inglês *depth first search*) difere daquela adotada na busca em largura na medida em que, a partir do vértice raiz, ela procura ir o mais "longe" possível no grafo sempre passando por vértices ainda não visitados.

Se ao chegar em um vértice v a busca não consegue avançar, ela retorna ao vértice `pred`$[v]$ cuja adjacência estava sendo explorada no momento em que v foi visitado pela primeira vez e volta a explorá-la.

É uma generalização do percurso em **pré-ordem** para árvores.

Busca em profundidade

No grafo da figura abaixo, novamente supondo o vértice a como sendo raiz da busca e que as listas de adjacência estão ordenadas em ordem alfabética, a ordem de visitação dos vértices seria: $\{a, b, d, c, e, i, j, g, h, l\}$.



Busca em profundidade

- O algoritmo a seguir implementa uma busca em profundidade em um grafo. Os principais elementos do algoritmo são semelhantes aos do algoritmo da busca em largura.
  As diferenças fundamentais entre os dois algoritmos assim como alguns elementos específicos da DFS são destacados abaixo.
- para que a estratégia de busca tenha o comportamento desejado, deve-se usar uma **pilha** no lugar da fila. Por razões de eficiência, a pilha deve armazenar não apenas uma informação sobre o vértice cuja adjacência deve ser explorada mas também um apontador para o próximo elemento da lista a ser percorrido.
- o vetor `dist` conterá não mais a distância dos vértices alcançáveis a partir da raiz e sim o número de arestas percorridas pelo algoritmo até visitar cada vértice pela primeira vez.

Busca em profundidade

- p é uma variável apontadora de um registro da lista de adjacências, para o qual é assumida a existência de dois campos: `vert`, contendo o rótulo que identifica o vértice, e `prox` que aponta para o próximo registo da lista.
- Excluídas estas pequenas diferenças, as implementações dos algoritmos BFS e DFS são bastante parecidas. Assim, é fácil ver que **a complexidade da busca em profundidade também é** $O(|V| + |E|)$, isto é, linear no tamanho da representação do grafo por listas de adjacências.

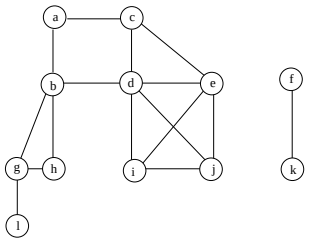
Busca em profundidade: algoritmo iterativo

```
DFS(G,raiz);
▷ Q é o conjunto de vértices a serem explorados (cinzas)
1.  InicializaPilha(Q);          ▷ inicializa pilha como sendo vazia
2.  dist[raiz] ← 0;   cor[raiz] ← cinza;   pred[raiz] ← NULO;
3.  Para todo v ∈ V − {raiz} faça
4.      { dist[v] ← ∞;   cor[v] ← branco;   pred[v] ← NULO; }
5.  Empilha(Q, raiz, Adj[raiz]);

6.  Enquanto (!PilhaVazia(Q)) faça
7.      Desempilha(Q, u, p);        ▷ tirar u do conjunto Q
8.      Enquanto (p ≠ NULO) e (cor[p ¬ vert] ≠ branco) faça p ← p ¬ prox;
9.      Se p ≠ NULO então
10.         Empilha(Q, u, p ¬ prox);   v ← p ¬ vert;   dist[v] ← dist[u] + 1;
11.         pred[v] ← u;   cor[v] ← cinza;   Empilha(Q, v, Adj[v]);
12.     Se não cor[u] ← preto;

13. Retorne (dist, pred).
```


Busca em profundidade: exemplo



Nota: o subgrafo $G_{\texttt{pred}}$ formado por $(V, E_{\texttt{pred}})$, onde $E_{\texttt{pred}} = \{(\texttt{pred}[v], v) : \forall v \in V\}$, é uma árvore, chamada de **árvore DFS**.

Busca em profundidade: versão recursiva

- Apresenta-se a seguir uma versão recursiva da busca em profundidade.
- Nesta versão, **todos** vértices do grafo serão visitados. Ou seja, ao final, teremos uma **floresta** de árvores DFS.
- O mesmo pode ser feito para BFS mas, é mais natural nas aplicações de busca em profundidade.
- Uma variável `tempo` será usada para marcar os instantes onde um vértice é descoberto (d) pela busca e onde sua vizinhança termina de ser explorada (f).
- A exemplo do que ocorre na busca em largura, algumas das variáveis do algoritmo são desnecessárias para efetuar a busca em profundidade pura e simples.

  Contudo, elas são fundamentais para aplicações desta busca e, por isso, são mantidas aqui.

Busca em profundidade: versão recursiva

DFS(G)

```
1.  para cada vértice u ∈ V faça
2.      cor[u] ← branco;        pred[u] ← NULO;
3.  tempo ← 0;
4.  para cada vértice u ∈ V faça
5.      se cor[u] = branco então  DFS-AUX(u)

DFS-AUX(u)        ▷ u acaba de ser descoberto
1.  cor[u] ← cinza;
2.  tempo ← tempo + 1;        d[u] ← tempo;
3.  para v ∈ Adj[u] faça        ▷ explora aresta (u, v)
4.      se cor[v] = branco então
5.          pred[v] ← u;        DFS-AUX(v);
6.  cor[u] ← preto;        ▷ u foi explorado
7.  tempo ← tempo + 1;        f[u] ← tempo;
```

Propriedades das buscas: BFS

Notação: $\delta(s, v)$ é a distância de s a v, ou seja, menor comprimento de um caminho ligando estes dois vértices.

Lema 22.1: (Cormen, 2ed)

Seja $G = (V, E)$ um grafo (direcionado ou não) e s um vértice qualquer de V. Então, para toda aresta $(u, v) \in E$, tem-se que $\delta(s, v) \leq \delta(s, u) + 1$.

Prova: u é alcançável a partir de s ... Se não for ... □

Lema 22.2: (Cormen, 2ed)

Suponha que uma BFS é executada em G tendo s como raiz. Ao término da execução, para cada vértice $v \in V$, o valor de $\texttt{dist}[v]$ computado pelo algoritmo satisfaz $\texttt{dist}[v] \geq \delta(s, v)$.

Propriedades das buscas: BFS

Prova do Lema 22.2: indução no número de operações de inserção na fila.

Hipótese indutiva (HI): $\texttt{dist}[v] \geq \delta(s, v)$ para todo $v \in V$.

Base da indução: inclusão inicial de s na fila ...

Passo da indução: v é um vértice branco descoberto durante a exploração da adjacência do vértice u. Como a HI implica que $\texttt{dist}[u] \geq \delta(s, u)$, as operações das linhas 10 e 11 e o Lema 22.1 implicam que:

$$\texttt{dist}[v] = \texttt{dist}[u] + 1 \geq \delta(s, u) + 1 \geq \delta(s, v).$$

O valor de $\texttt{dist}[v]$ nunca mais é alterado pelo algoritmo ... □

Propriedades das buscas: BFS

Para provar que $\texttt{dist}[v] = \delta(s, v)$, é necessário entender melhor como opera a fila Q.

Lema 22.3: (Cormen, 2ed)

Suponha que numa iteração qualquer do algoritmo BFS, os vértices na fila Q sejam dados por $\{v_1, v_2, \ldots, v_r\}$, sendo v_1 a cabeça e v_r a cauda da fila. Então, $\texttt{dist}[v_r] \leq \texttt{dist}[v_1] + 1$ e $\texttt{dist}[v_i] \leq \texttt{dist}[v_{i+1}]$, para todo $i = 1, 2, \ldots, r - 1$.

Prova: indução no número de operações na fila (insere/remove).

Base da indução: a afirmação é evidente quando $Q = \{s\}$.

Passo da indução: dividido em duas partes.
A primeira trata da remoção de v_1, o que torna v_2 a nova *cabeça* de Q ...

Propriedades das buscas: BFS

Prova do Lema 22.3: (continuação)
A segunda trata de inclusão de um novo vértice v, tornando-o a nova *cauda* (v_{r+1}) de Q. Supõe-se que o vértice v foi descoberto na exploração da vizinhança de um vértice u. Pela HI e pelas operações das linhas 10 e 11 chega-se a:

$$\texttt{dist}[v_{r+1}] = \texttt{dist}[v] = \texttt{dist}[u] + 1 \leq \texttt{dist}[v_1] + 1.$$

A HI também implica que

$$\texttt{dist}[v_r] \leq \texttt{dist}[u] + 1 = \texttt{dist}[v] = \texttt{dist}[v_{r+1}],$$

e, além disso, as demais desigualdades ficam inalteradas. □

Propriedades das buscas: BFS

Corolário 22.4: (Cormen, 2ed)

Suponha que os vértices v_i e v_j são inseridos na fila Q durante a execução do algoritmo BFS nesta ordem. Então, $\texttt{dist}[v_i] \leq \texttt{dist}[v_j]$ a partir do momento em que v_j for inserido em Q.

Prova: Imediato. □

Teorema 22.5: corretude da BFS (Cormen, 2ed)

Durante a execução do algoritmo BFS, todos os vértices $v \in V$ alcançáveis a partir de s são descobertos e, ao término da execução, $\texttt{dist}[v] = \delta(s, v)$ para todo $v \in V$.

Além disso, para todo vértice $v \neq s$ alcançável a partir de s, um dos menores caminhos de s para v é composto por um menor caminho de s para $\texttt{pred}[v]$ seguido da aresta $(\texttt{pred}[v], v)$.

Propriedades das buscas: DFS

- Verifica-se agora algumas propriedades da busca em profundidade. Para tanto, usa-se a versão recursiva do algoritmo e para todo vértice u, denota-se por I_u o intervalo $[d[u], f[u]]$.

Teorema 22.7 (Parentização): (Cormen, 2ed)

Em qualquer busca em profundidade em um grafo $G = (V, E)$ (direcionado ou não), para quaisquer vértices u e v de V, **exatamente** uma das situações abaixo ocorre:

- $I_u \cap I_v = \{\}$ e u e v não são descendentes um do outro na floresta construída pela DFS, *ou*
- $I_u \subset I_v$ e o u é descendente de v em uma árvore da floresta construída pela DFS,
- $I_v \subset I_u$ e o v é descendente de u em uma árvore da floresta construída pela DFS.

Propriedades das buscas: DFS

Corolário 22.8 (aninhamento dos intervalos dos descententes): (Cormen, 2ed)

O vértice v é um descendente *próprio* do vértice u em uma árvore da floresta construída pela DFS *se e somente se* $d[u] < d[v] < f[v] < f[u]$.

Uma outra caracterização de descendência é dada pelo **Teorema do Caminho Branco** enunciado a seguir.

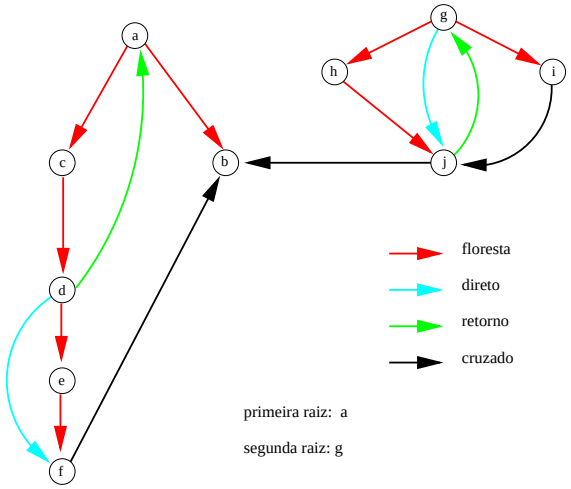
Teorema 22.9: (Cormen, 2ed)

Em uma floresta construída pela DFS, um vértice v é descendente de um vértice u *se e somente se* no *tempo* $d[u]$ em que a busca **descobre** u, o vértice v é alcançável a partir de u por um caminho *inteiramente* composto por vértices de **cor branca**.

Propriedades das buscas: Classificação de arestas

- O algoritmo DFS permite classificar arestas do grafo e esta classificação dá acesso a importantes informações sobre o mesmo (p.ex., a existência de ciclos em grafos direcionados).
- Considere a floresta $G_{\texttt{pred}}$ construída pela DFS. As arestas podem se classificar como sendo:
  - **Arestas da floresta:** aquelas em $G_{\texttt{pred}}$.
  - **Arestas de retorno (*back edges*):** arestas (u, v) conectando o vértice u a um vértice v que é seu **ancestral** em $G_{\texttt{pred}}$.
    Em grafos direcionados, os auto-laços são considerados arcos de retorno.
  - **Arestas diretas (*forward edges*):** arestas (u, v) que não estão em $G_{\texttt{pred}}$ onde v é descendente de u em uma árvore construída pela DFS.
  - **Arestas cruzadas (*cross edges*):** todas as arestas restantes.
    Elas podem ligar vértices de uma mesma árvore de $G_{\texttt{pred}}$, desde que nenhuma das extremidades seja

Classificação de arestas: exemplo



Classificação de arestas: modificações no algoritmo

- O algoritmo DFS pode ser modificado para ir classificando as arestas a medida que elas são encontradas.
- A idéia é que a aresta (u, v) pode ser classificada de acordo co a cor do vértice v no momento que a aresta é "explorada" pela primeira vez:
  1. `branco`: indica que (u, v) é um aresta da floresta (está em $G_{\texttt{pred}}$).
  2. `cinza`: indica que (u, v) é um aresta de retorno.
  3. `preto`: indica que (u, v) é um aresta direta ou uma aresta cruzada.
     Pode-se mostrar (exercício) que ela será direta se $d[u] < d[v]$, caso contrário será cruzada.

Classificação de arestas: modificações no algoritmo

- Para grafos não direcionados, a classificação pode apresentar ambigüidades já que (u, v) e (v, u) são de fato a mesma aresta.
- O algoritmo pode ser ajustado para manter a primeira classificação em que a aresta puder ser categorizada.
- Arestas diretas e cruzadas *nunca* ocorrem em grafos não-direcionados.

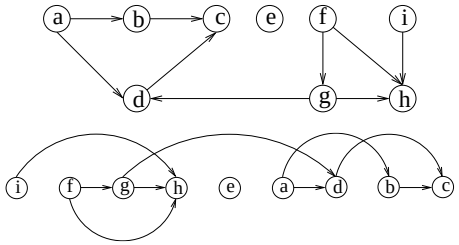
Teorema 22.10: (Cormen, 2ed)

Em uma DFS de um grafo **não direcionado** $G = (V, E)$, toda aresta é da floresta ou é de retorno.

Ordenação topológica

Ordenação topológica

- A **ordenação topológica** de um grafo direcionado acíclico (**DAG**) $G = (V, E)$ é uma ordem linear dos vértices tal que, para todo arco (u, v) em E, u *antecede* v na ordem.
- *Todo DAG admite uma ordenação topológica !*
- Graficamente, isto significa que G pode ser desenhado de modo que todos seus vértices fiquem dispostos em uma linha horizontal e os seus arcos fiquem todos orientados *da esquerda para direita*.



Ordenação topológica

- Grafos direcionados acíclicos são muito usados para representar situações onde haja relações de precedência entre eventos. Exemplo: tarefas necessárias para construir uma casa.
- Em muitas aplicações a ordenação topológica é usada como um pré-processamento que conduz a um algoritmo *eficiente* para resolver um problema definido sobre grafos.
- A ordenação topológica pode ser obtida através de uma simples adaptação do algoritmo de busca em profundidade mostrada a seguir.

Ordenação topológica: algoritmo

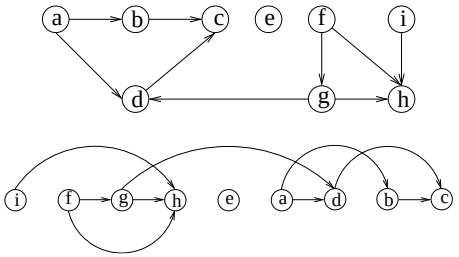
TOPOLOGICAL-SORT(G)

▷ **Entrada:** grafo direcionado acíclico G.
▷ **Saída:** lista ligada L com ordem topológica dos vértices de G.
1. InicializaLista(L); `tempo` $\leftarrow 0$;
2. **para** cada vértice $u \in V$ **faça** `cor`$[u] \leftarrow$ `branco`;
3. **para** cada vértice $u \in V$ **faça**
4. **se** `cor`$[u] =$ `branco` **então** TOP-SORT-AUX(u)
5. **retorne** L.

TOP-SORT-AUX(u) ▷ u acaba de ser descoberto
1. `cor`$[u] \leftarrow$ `cinza`; `tempo` $\leftarrow$ `tempo` $+ 1$; $d[u] \leftarrow$ `tempo`;
2. **para** $v \in$ `Adj`$[u]$ **faça** ▷ explora aresta (u, v)
3. **se** `cor`$[v] =$ `branco` **então** TOP-SORT-AUX(v);
4. `cor`$[u] \leftarrow$ `preto`; ▷ u foi explorado
5. InsereInicioLista(L, u);
6. `tempo` $\leftarrow$ `tempo` $+ 1$; $f[u] \leftarrow$ `tempo`;

Ordenação topológica

- Vê-se que os vértices são armazenados na lista ligada L na *ordem inversa* (decrescente) dos valores de $f[.]$ (a inserção se dá sempre no inicio da lista).
- A *complexidade* deste algoritmo é obviamente igual àquela da DFS, ou seja, $O(|V| + |E|)$.
- **Exemplo:** raízes a, e, f e i.



Ordenação topológica: corretude

Lema 22.11 (Cormen 2ed):

Um grafo direcionado G é acíclico *se e somente se* uma busca em profundidade em G não classifica nenhum arco como sendo de retorno.

Prova: ($\Longrightarrow$) por contradição. Supor que (u, v) é um arco de retorno. O único caminho em $G_{\texttt{pred}}$ de v para u juntamente com o arco(u, v) forma um ciclo.

($\Longleftarrow$) por contradição. Supor que existe um ciclo C em G. Seja v o primeiro vértice de C descoberto pela busca e (u, v) o arco precedendo v em C. No instante $d[v]$, os arcos de $C - \{(u, v)\}$ formam um **caminho branco** ligando v a u. Logo, u será um descendente de v e, então, (u, v) é um arco de retorno. □

Ordenação topológica: corretude

Teorema 22.12 (Cormen 2ed):

O algoritmo `TOPOLOGICAL-SORT`(G) produz uma ordenação topológica correta de um grafo direcionado acíclico.

Prova: suponha que DFS tenha sido executado sobre o DAG G. Seja (u, v) um arco de G e considere o instante onde este arco está sendo explorado. A cor de v não pode ser `cinza`, pois (u, v) seria arco de retorno, contrariando o Lema 22.11. Se $\texttt{cor}[v] = \texttt{branco}$, v é descendente de u e, assim, $f[v] < f[u]$. Se $\texttt{cor}[v] = \texttt{preto}$, $f[v]$ já foi fixado. Como (u, v) está sendo explorado, $\texttt{cor}[u] = \texttt{cinza}$ e $f[u]$ ainda não foi fixado. Portanto, $f[v] < f[u]$.

Conclusão: quando (u, v) está em G, $f[u] > f[v]$, ou seja, u aparecerá antes de v na ordenação dada pelo algoritmo, como requerido. □