Busca em profundidade

Um *algoritmo de busca* (ou *de varredura*) é qualquer algoritmo que visita todos os vértices de um grafo andando pelos arcos de um vértice a outro. Há muitas maneiras de fazer uma tal busca. Cada algoritmo de busca é caracterizado pela ordem em que visita os vértices.

Este capítulo introduz o poderoso algoritmo de *busca em profundidade* (= *depth-first search*), ou *busca DFS*. Trata-se de uma generalização do algoritmo que estudamos no capítulo *Acessibilidade*. Lá o objetivo era decidir se um vértice está ao alcance de outro. Aqui, é objetivo é visitar todos os vértices e *numerá-los* na ordem em que são descobertos.

A busca em profundidade não resolve um problema específico. Ela é apenas um arcabouço, ou pré-processamento, para a resolução eficiente de vários problemas concretos. A busca DFS nos ajuda a "compreender" o grafo com que estamos lidando, revelando sua "forma" e reunindo informações (representadas pela numeração dos vértices) que ajudam a responder perguntas sobre o grafo.

A busca em profundidade está relacionada com expressões como pré-ordem, exploração de labirintos, exploração Trémaux, fio de Ariadne (no mito de Teseu e o Minotauro), etc.

Sumário:

- Busca DFS
- Desempenho da busca DFS
- Pré-ordem
- Perguntas e respostas

Busca DFS

O algoritmo de busca DFS visita todos os vértices e todos os arcos do grafo numa determinada ordem e atribui um número a cada vértice: o *k-ésimo* vértice descoberto recebe o número k .

A função `GRAPHdfs()` abaixo é uma implementação do algoritmo. A busca poderia começar por qualquer vértice, mas é natural começá-la pelo vértice `0`. A numeração dos vértices é registrada em um vetor **`pre[]`** indexado pelos vértices.

Para simplificar o código, trataremos o vetor `pre[]` como variável global e suporemos que o número de vértices não passa de 1000. (Veja abaixo o exercício *Alocação dinâmica*.) Também trataremos como variável global o contador **`cnt`** usada para a numeração:

```
static int cnt;
int pre[1000];

/* A função GRAPHdfs() faz uma busca em profundidade no grafo G. Ela atribui
um número de ordem pre[x] a cada vértice x de modo que o k-ésimo vértice des-
coberto receba o número de ordem k.  (Código inspirado no programa 18.3 de
Sedgewick.) */
```

```
void GRAPHdfs( Graph G)
{
   cnt = 0;
   for (vertex v = 0; v < G->V; ++v)
      pre[v] = -1;
   for (vertex v = 0; v < G->V; ++v)
      if (pre[v] == -1)
         dfsR( G, v); // começa nova etapa
}
```

A função GRAPHdfs() é apenas um invólucro; a busca propriamente dita é realizada pela função [recursiva](#) dfsR(). Em geral, nem todos os vértices estão [ao alcance](#) do primeiro vértice visitado em GRAPHdfs(), e portanto a função dfsR() precisa ser invocada várias vezes por GRAPHdfs(). Cada uma dessas invocações define uma ⚠ *etapa* da busca.

```
/* A função dfsR() visita todos os vértices de G que podem ser alcançados a
partir do vértice v sem passar por vértices já descobertos. A função atribui
cnt+k a pre[x] se x é o k-ésimo vértice descoberto. (O código supõe que G é
representado por listas de adjacência.) */

static void dfsR( Graph G, vertex v)
{
   pre[v] = cnt++;
   for (link a = G->adj[v]; a != NULL; a = a->next) {
      vertex w = a->w;
      if (pre[w] == -1)
         dfsR( G, w);
   }
}
```

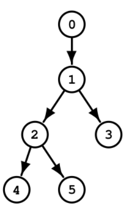
Convém esclarecer os termos "visitar" e "descobrir" que usamos até aqui de maneira informal. Dizemos que um vértice x é *visitado* toda vez que pre[x] é consultado, isto é, comparado com -1. Um vértice x é *descoberto* quando visitado pela primeira vez, ocasião em que pre[x], que valia -1, recebe o valor corrente de cnt. Um vértice x já foi visitado se e somente se pre[x] é diferente de -1.

Cada [etapa da busca](#) pode ser resumida assim: (*a*) escolha um vértice não descoberto v e (*b*) visite todos os vértices que estão ao alcance de v mas ainda não foram visitados. No passo (*a*), qualquer vértice não descoberto pode ser escolhido para fazer o papel de v. A função GRAPHdfs() escolhe os vértices seguindo a ordem 0, 1, 2, etc., mas poderia ter usado qualquer outra ordem. Da mesma forma, a ordem em que os vizinhos de v são visitados na função dfsR() não é importante.

Qual a diferença entre a função GRAPHdfs() e a função [GRAPHreach()](#) que estudamos no capítulo [*Acessibilidade*](#)? Há apenas duas pequenas diferenças: (1) GRAPHdfs() visita *todos* os vértices do grafo, enquando GRAPHreach() visita apenas os que estão ao alcance de um dado vértice s e (2) GRAPHdfs() atribui um rótulo numérico diferente a cada vértice, enquanto GRAPHreach() usa apenas os rótulos 0 e 1.

Exemplo A. Considere o grafo G definido pelos arcos 0-1 1-2 1-3 2-4 2-5 . Veja a matriz de adjacências (com "-" no lugar de "0") e as listas de adjacência do grafo:

```
- 1 - - - -     0:  1
- - 1 1 - -     1:  2 3
- - - - 1 1     2:  4 5
- - - - - -     3:
- - - - - -     4:
- - - - - -     5:
```



Segue o rastreamento (= *trace*) de uma execução de `GRAPHdfs(G)`. As linhas da tabela que começam com `v-w` registram o momento em que a função `dfsR()` percorre o arco `v-w`, ou seja, o momento em que a função se depara com `w` ao examinar os vizinhos de `v`. Cada expressão da forma `dfsR(G,w)` registra uma invocação de `dfsR()`. Cada nova invocação de `dfsR()` é indicada por uma indentação apropriada da linha.

```
v-w dfsR(G,w)

0 dfsR(G,0)
0-1 dfsR(G,1)
  1-2 dfsR(G,2)
    2-4 dfsR(G,4)
      4
    2-5 dfsR(G,5)
      5
    2
  1-3 dfsR(G,3)
    3
  1
0
```

As linhas em que aparece apenas um vértice `v` representam o fim da execução de `dfsR(G,v)`, ou seja, a morte da encarnação `dfs(G,v)` de `dfsR()`. Para determinar o estado finaldo vetor `pre[]`, basta examinar a sequência de expressões da forma `dfsR(G,w)` no rastreamento acima:

```
     w   0 1 2 4 5 3
pre[w]   0 1 2 3 4 5
```

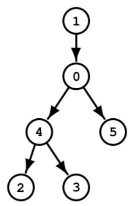
Podemos reescrever esse vetor colocando em ordem crescente a primeira linha da tabela:

```
     w   0 1 2 3 4 5
pre[w]   0 1 2 5 3 4
```

Nesse exemplo, a busca tem apenas uma etapa.

Exemplo B. Seja `G` o grafo definido pelos arcos `1-0 0-4 0-5 4-2 4-3`. Veja as listas de adjacências do grafo:

```
0:  4 5
1:  0
2:
3:
4:  2 3
5:
```



Segue o rastreamento de uma execução de `GRAPHdfs(G)`. As linhas da tabela que começam com `v-w` registram o momento em que a função `dfsR()` percorre o arco `v-w`. Se a ponta final do arco ainda não foi visitada, a linha da tabela também registra a invocação de `dfsR(G,w)`. Um vértice `v` isolado numa linha da tabela representa o fim da encarnação `dfsR(G,v)` de `dfsR()`.

```
0 dfsR(G,0)
0-4 dfsR(G,4)
  4-2 dfsR(G,2)
    2
  4-3 dfsR(G,3)
    3
  4
0-5 dfsR(G,5)
  5
0

1 dfsR(G,1)
1-0
```

```
    0
  1
```

O bloco de linhas que começa com `dfsR(G,0)` corresponde à primeira [etapa](#) da busca. O bloco que começa com `dfsR(G,1)` corresponde à segunda etapa. É fácil deduzir `pre[]` do rastreamento pois a função numera em sequência os vértices `w` que aparecem nas expressões da forma `dfsR(G,w)`. Assim, o estado final do vetor `pre[]` é

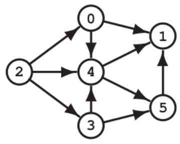
```
     w   0 4 2 3 5 1
pre[w]   0 1 2 3 4 5
```

Se preferir, a primeira linha da tabela pode ser colocada em ordem crescente:

```
     w   0 1 2 3 4 5
pre[w]   0 5 2 3 1 4
```

Exemplo C. Seja `G` o grafo definido pelos arcos `2-0 2-3 2-4 0-1 0-4 3-4 3-5 4-1 4-5 5-1`. Veja as listas de adjacência do grafo:

```
0:  1 4
1:
2:  0 3 4
3:  4 5
4:  1 5
5:  1
```



Segue o rastreamento da execução de `GRAPHdfs(G)`. (É fácil conferir o rastreamento pois os vizinhos de cada vértice aparecem em ordem crescente na lista de adjacência.)

```
0 dfsR(G,0)
0-1 dfsR(G,1)
  1
0-4 dfsR(G,4)
  4-1
  4-5 dfsR(G,5)
    5-1
    5
  4
0

2 dfsR(G,2)
2-0
2-3 dfsR(G,3)
  3-4
  3-5
  3
2-4
2
```

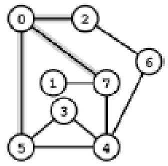
O bloco de linhas que começa com `dfsR(G,0)` corresponde à primeira etapa da busca. O bloco que começa com `dfsR(G,2)` corresponde à segunda etapa. Para deduzir o estado final do vetor `pre[]`, basta observar a sequência de valores de `w` nas expressões da forma `dfsR(G,w)`:

```
     w   0 1 4 5 2 3
pre[w]   0 1 2 3 4 5
```

Exemplo D. Este exemplo ilustra uma busca em profundidade num grafo [não-dirigido](#). O grafo `G` tem conjunto de arestas `0-2 0-5 0-7 1-7 2-6 3-4 3-5 4-5 4-6 4-7` e portanto suas listas de adjacências são

```
0: 2 5 7
1: 7
```

```
2: 0 6
3: 4 5
4: 3 5 6 7
5: 0 3 4
6: 2 4
7: 0 1 4
```



Segue o rastreamento de uma execução de `GRAPHdfs(G)`. As linhas da tabela que começam com `v-w` registram o momento em que a função `dfsR()` percorre o arco `v-w` (esse arco é uma "metade" de uma aresta do grafo).

```
0 dfsR(G,0)
0-2 dfsR(G,2)
. 2-0
. 2-6 dfsR(G,6)
. . 6-2
. . 6-4 dfsR(G,4)
. . . 4-3 dfsR(G,3)
. . . . 3-4
. . . . 3-5 dfsR(G,5)
. . . . . 5-0
. . . . . 5-3
. . . . . 5-4
. . . . . 5
. . . . 3
. . . 4-5
. . . 4-6
. . . 4-7 dfsR(G,7)
. . . . 7-0
. . . . 7-1 dfsR(G,1)
. . . . . 1-7
. . . . . 1
. . . . 7-4
. . . . 7
. . . 4
. . 6
. 2
0-5
0-7
0
```

Nesse exemplo, a busca tem uma só etapa. (Os pontos servem apenas para deixar o alinhamento vertical mais visível.) Como vimos nos exemplos anteriores, é fácil deduzir do rastreamento o estado final do vetor `pre[]`:

```
     v   0 2 6 4 3 5 7 1
pre[v]   0 1 2 3 4 5 6 7
```

A tabela pode ser reescrita com a primeira linha em ordem crescente:

```
     v   0 1 2 3 4 5 6 7
pre[v]   0 7 1 4 3 5 2 6
```

Exercícios 1

1. *Instâncias extremas.* Qual o resultado de `GRAPHdfs( G)` quando `G->A` vale `0`? E quando `G->V` vale `1`?
2. *Base da recursão.* Qual o base da recursão na função `dfsR()`?
3. Suponha que uma etapa de `GRAPHdfs()` começa com um certo vértice `v`. É verdade que o conjunto de vértices descobertos por `dfsR(G, v)` é `v v+1 v+2 ...` ?
4. A seguinte afirmação está correta? "A função `dfsR()` visita todos os vértices que podem ser alcançados a partir do vértice `v`."

5. ★ Considere a árvore radicada cujos arcos são `1-0 0-2 0-3 3-4 3-5 1-6 6-7 6-8 6-9 1-10 10-11`. (Dê uma numeração topológica dos vértices, mas evite fazer uma figura do grafo.) Submeta a árvore radicada à função `GRAPHdfs()` e mostre o resultado.

6. ★ Aplique a função `GRAPHdfs()` ao grafo definido pelos arcos `0-1 1-2 1-3 3-4 3-5 1-6 0-7 7-8 7-9 8-6`. Em que ordem os vértices são descobertos?

7. Execute uma busca em profundidade no grafo dado pelas listas de adjacência a seguir. Faça o rastreamento da busca.

```
 0: 1 4
 1: 2 5
 2: 3
 3: 7
 4: 8
 5: 4
 6: 5 10 2
 7: 11 6
 8: 9
 9: 5 8
10: 9
11: 10
```

8. [Sedgewick 18.4] Aplique a função `GRAPHdfs()` ao grafo não-dirigido definido pelas arestas `0-2 0-5 1-2 3-4 4-5 3-5` e faça o rastreamento da execução da função.

9. [Sedgewick 18.7] Faça o rastreamento da execução da função `GRAPHdfs()` sobre o grafo não-dirigido definido pelas arestas `3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4`.

10. ★ [Sedgewick 17.85] *Imprima rastreamento.* Modifique as funções `GRAPHdfs()` e `dfsR()` de modo que elas imprimam um rastreamento como o dos exemplos acima. Para fazer a indentação, use uma variável global `indent` que é incrementada quando a execução entra em `dfsR()` e decrementada quando a execução sai de `dfsR()`. Para testar suas funções, prepare um programa que aceite como entrada um grafo representado por um arquivo de texto.

11. Qual a principal diferença entre a busca em profundidade, executada pela função `GRAPHdfs()`, e a busca por um caminho a partir de um dado vértice, executada pela função `GRAPHreach()`?

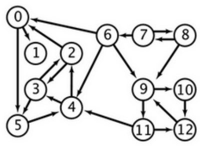
12. *Alocação dinâmica.* No código da função `GRAPHdfs()`, o vetor `pre[]` foi alocado estaticamente. Reescreva o código fazendo alocação dinâmica do vetor.

Exercícios 2 (matriz de adjacências)

1. [Sedgewick 17.89] *DFS com matriz de adjacências.* Reescreva a função `dfsR()` supondo que o grafo é representado por uma matriz de adjacências.

2. [Sedgewick 18.5] *Vértices examinados em ordem inversa.* Reescreva a função `dfsR()` para grafos representados por matriz de adjacências de modo que cada linha da matriz seja percorrida "ao contrário", isto é, de `V-1` para `0`. Agora refaça o rastreamento dos exemplos A, B, C e D.

3. Faça uma busca em profundidade no grafo da figura. Suponha que o grafo é representado por sua matriz de adjacências.



Desempenho da busca DFS

A função `GRAPHdfs()` examina o leque de saída de cada vértice uma só vez. Portanto, cada arco é examinado uma só vez. Assim, se o grafo tem V vértices e A arcos, `GRAPHdfs()` consome tempo proporcional a

$$V + A .$$

(Esse consumo é proporcional ao tamanho do grafo e portanto também ao tempo necessário para ler todas as listas de adjacência.) No caso de grafos representados por *matriz* de adjacências, a função `GRAPHdfs()`, combinada com a versão apropriada de `dfsR()`, consome tempo proporcional a

$$V^2$$

quando aplicada a um grafo com V vértices. (Esse consumo é proporcional ao tempo necessário para ler a matriz de adjacências.) Se o grafo é esparso, esta segunda versão é mais lenta que a primeira.

Pré-ordem

A ordem em que a função `GRAPHdfs()` descobre os vértices do grafo é chamada *pré-ordem* (= *preorder*). Para obter a permutação dos vértices em pré-ordem basta inverter o vetor `pre[]`:

```
vertex vv[1000];
for (vertex v = 0; v < G->V; ++v)
   vv[pre[v]] = v;
for (int i = 0; i < G->V; ++i)
   printf( "%d ", vv[i]);
```

Para ilustrar, considere o exemplo D acima. O estado final de `pre[]` informa que os vértice foram descobertos na ordem `0 2 6 4 3 5 7 1`. Esta é, então, a permutação dos vértices em pré-ordem.

É claro que a pré-ordem depende da sequência em que os vértices aparecem nas listas de adjacência. Se alterarmos essa sequência, a pré-ordem também se altera, ⚠ mas continua merecendo o nome *pré-ordem*. Essas variações da pré-ordem são especialmente evidentes quando o grafo é representado por sua matriz de adjacências, pois nesse caso `dfsR()` pode percorrer as linhas da matriz em diferentes ordens (como da direita para a esquerda, por exemplo).

Resumindo, dizemos que qualquer implementação do algoritmo de busca em profundidade descobre os vértices do grafo *em pré-ordem*.

Se o grafo for uma árvore radicada, a permutação dos vértices em pré-ordem pode ser descrita recursivamente: visite a raiz; depois, para cada vizinho `w` da raiz, visite, em pré-ordem, a subárvore que tem raiz `w`.

Exercícios 3

1. [Sedgewick 18.8] Faça uma busca em profundidade no grafo não-dirigido definido pelas arestas `3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4` e exiba o rastreamento da busca. Suponha que o grafo é dado por suas listas de adjacência e que as listas são construídas inserindo as arestas uma a uma, na ordem dada, num grafo inicialmente vazio.
2. Refaça o exemplo D acima depois de alterar a ordem em que os vértices aparecem na lista de adjacência de cada vértice.
3. ★ Faça uma figura do grafo definido pelos arcos `a-c a-d a-g b-d b-f g-c d-c d-e f-e c-h e-h`. Faça uma busca em profundidade no grafo percorrendo os vértices em ordem alfabética de nomes. Suponha que o grafo é representado por listas de adjacência. Comece por escrever as listas de modo que estejam em ordem alfabética. Deduza do resultado da busca a permutação dos vértices em pré-ordem.
4. *Permutação versus numeração.* Imagine um grafo com vértices `0 1 2 3 4 5 6 7 8 9`. Suponha que uma execução de `GRAPHdfs()` descobre os vértices na ordem `2 8 7 1 3 4 5 9 0 6`. Exiba o estado do vetor `pre[]` no fim da execução da função.

5. *Numeração versus permutação.* Imagine um grafo com vértices `0 1 2 3 4 5 6 7 8 9`. Suponha que uma execução de `GRAPHdfs()` produz o vetor `pre[]` abaixo. Em que ordem os vértices foram descobertos?

```
     v    0 1 2 3 4 5 6 7 8 9
pre[v]    9 8 3 4 5 2 0 7 1 6
```

6. Um grafo não-dirigido tem conjunto de arestas `0-1 0-2 1-3 1-4 1-5 3-6 3-7 4-7 5-7`. Suponha que uma busca em profundidade descobre os vértices na ordem `0 1 5 7 4 3 6 2`. Faça uma figura das listas de adjacência do grafo que seja compatível com essa pré-ordem. Repita o exercício supondo que os vértices são descobertos na ordem `0 1 4 3 2 5 6 7`.

7. *Nem tudo é pré-ordem.* Mostre que nem toda permutação do conjunto de vértices de um grafo é uma pré-ordem.

8. ★ *É pré-ordem?* É possível representar o grafo do exemplo A por listas de adjacência de modo que a permutação dos vértices em pré-ordem seja `3 1 2 5 4 0`? Repita o exercício com `3 1 4 2 5 0`?

9. ★ *É pré-ordem?* Considere a permutação `0 1 4 2 3 5` dos vértices do grafo do exemplo C. Existe alguma representação do grafo para a qual essa permutação está em pré-ordem? Repita o exercício com a permutação `0 4 5 1 2 3`.

10. ★ *É pré-ordem?* Considere o grafo do exemplo D. Existe alguma representação do grafo para a qual a permutação `0 5 4 6 2 3 7 1` dos vértices está em pré-ordem? Repita o exercício com a permutação `0 2 6 4 3 5 7 1`.

11. Considere a árvore radicada definida pelos arcos `0-1 1-2 1-3 0-4 4-5 4-6`. (Dê uma numeração topológica dos vértices.) Qual das duas permutações de vértices a seguir está em pré-ordem?

```
0  1  2  3  5  6  4
0  3  6  5  1  2  4
```

12. [Sedgewick 18.11] Considere o grafo não-dirigido definido pelas arestas `0-1 0-9 1-4 1-9 2-7 2-10 2-12 3-12 5-12 6-10 6-12 7-10 8-11`. Existem 13! diferentes permutações dos vértices desse grafo. Quantas dessas permutações estão em pré-ordem supondo que o grafo é representado por listas de adjacência? (Sugestão: Considere todas as ordens em que os vértices podem aparecer nas listas de adjacência.)

13. ★ *Vértices examinados em ordem arbitrária.* Em geral, a ordem em que os vértices são examinados para determinar o início de uma nova etapa da busca — linha "`for (v = 0; v < G->V; ++v)`" de `GRAPHdfs()` — não é importante. Entretanto, em certos algoritmos (como o das componentes fortes), é necessário examinar os vértices *numa ordem específica*. Escreva uma variante de `GRAPHdfs()` que examine os vértices na ordem dada por uma permutação `vv[0..V-1]` dos vértices. (O vetor `vv[]` pode ser tratado como uma variável global ou como argumento da função.)

Exercícios 4 (versões iterativas)

1. O seguinte algoritmo iterativo executa uma busca em profundidade? As funções com prefixo `STACK` manipulam uma pilha de vértices: `STACKinit()` cria uma pilha vazia, `STACKpush()` coloca um vértice na pilha, `STACKpop()` retira um vértice da pilha e `STACKempty()` devolve `true` se a pilha está vazia. (Compare o código com a implementação da *busca em largura* a ser estudada adiante.)

```
#include "STACK.h"
static int cnt, pre[1000];
void GRAPHdfs?( Graph G) {
   STACKinit( G->V);
   cnt = 0;
   for (vertex v = 0; v < G->V; ++v) pre[v] = -1;
   for (vertex v = 0; v < G->V; ++v)
      if (pre[v] == -1)
         dfs?( G, v);
}
void dfs?( Graph G, vertex v) {
   pre[v] = cnt++;
```

```
      STACKpush( v);
      while (!STACKempty( )) {
         v = STACKpop( );
         for (vertex w = 0; w < G->V; ++w)
            if (G->adj[v][w] == 1)
               if (pre[w] == -1) {
                  pre[w] = cnt++;
                  STACKpush( w);
               }
      }
   }
```

2. ★ [Sedgewick 18.61] *DFS iterativa para listas de adjacência.* Escreva uma versão iterativa da função `dfsR()` para grafos representados por listas de adjacência. (Faça uma adaptação simples do exercício anterior.)

3. ★ [Sedgewick 18.61] *DFS iterativa para matriz de adjacências.* Escreva uma versão iterativa da função `dfsR()` para grafos representados por matriz de adjacências.

Perguntas e respostas

- PERGUNTA: Qual a vantagem de trocar V^2 por $V+A$ na análise de desempenho? Afinal, A não passa de V^2 e portanto $V+A$ é da mesma ordem que V^2.

  RESPOSTA: Mais ou menos. Se A for bem menor que V^2 (como acontece com grafos esparsos) então $V+A$ é bem menor que V^2.

- PERGUNTA: Por que não escrever "A" no lugar de "$V+A$" na análise de desempenho? Afinal, A é maior que V e portanto $V+A$ é da ordem de A.

  RESPOSTA: Não é bem assim. O número de arcos A pode ser muito menor que V. Pode até ser zero!

---

www.ime.usp.br/~pf/algoritmos_para_grafos/
Atualizado em 2019-04-08
Paulo Feofiloff
IME-USP