Busca em largura

Um algoritmo *de busca* é um algoritmo que percorre um grafo andando pelos arcos de um vértice a outro. Depois de visitar a ponta inicial de um arco, o algoritmo percorre o arco e visita sua ponta final. Cada arco é percorrido no máximo uma vez.

Há muitas maneiras de organizar uma busca. Cada estratégia de busca é caracterizada pela *ordem* em que os vértices são visitados. Este capítulo introduz a *busca em largura* (= *breadth-first search*), ou *busca BFS*. Essa estratégia está intimamente relacionada com os conceitos de *distância* e *caminho mínimo*.

Sumário:

Algoritmo de busca em largura

A busca em largura começa por um vértice, digamos s, especificado pelo usuário. O algoritmo visita s, depois visita todos os vizinhos de s, depois todos os vizinhos dos vizinhos, e assim por diante.

O algoritmo numera os vértices, em sequência, na ordem em que eles são descobertos (ou seja, visitados pela primeira vez). Para fazer isso, o algoritmo usa uma fila (= *queue*) de vértices. No começo de cada iteração, a fila contém vértices que já foram numerados mas têm vizinhos ainda não numerados. O processo iterativo consiste no seguinte:

```
enquanto a fila não estiver vazia
    retire um vértice v da fila
    para cada vizinho w de v
        se w não está numerado
            então numere w
                ponha w na fila
```

No começo da *primeira* iteração, a fila contém o vértice s, com número 0, e nada mais.

Exercícios 1

1. ★ Como começa cada iteração do algoritmo de busca em largura? (Cuidado! Não se trata de descrever as *ações* que ocorrem no começo da iteração. Trata-se de saber quais as informações disponíveis no início de uma iteração genérica, antes que a execução da iteração comece.)

2. ★ *BFS em árvore radicada.* Faça um rastreamento da busca em largura a partir do vértice `1` na árvore radicada definida pelos arcos `0-1 0-2 1-3 1-9 1-10 3-4 3-5 5-6 5-7 5-8 10-11 10-`

`12 12-13 12-14`. Observe que a busca em largura percorre a árvore "por níveis".

Implementação do algoritmo

A fila de vértices é manipulada pelas funções auxiliares `QUEUEinit()`, `QUEUEput()`, `QUEUEget()`, `QUEUEempty()` e `QUEUEfree()`. A primeira cria uma fila vazia, a segunda coloca um vértice na fila, a terceira tira um vértice da fila, a quarta verifica se a fila está vazia, e a última libera o espaço ocupado pela fila.

A numeração dos vértices é registrada num vetor `num[]` indexado pelos vértices. Se `v` é o [*k*-ésimo](#) vértice descoberto então `num[v]` recebe o valor *k*.

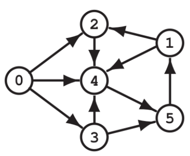
```
static int num[1000];

/* A função GRAPHbfs() implementa o algoritmo de busca em largura. Ela visita
todos os vértices do grafo G que estão ao alcance do vértice s e registra num
vetor num[] a ordem em que os vértices são descobertos. Esta versão da função
supõe que o grafo G é representado por listas de adjacência.  (Código inspi-
rado no programa 18.9 de Sedgewick.) */

void GRAPHbfs( Graph G, vertex s)
{
   int cnt = 0;
   for (vertex v = 0; v < G->V; ++v)
      num[v] = -1;
   QUEUEinit( G->V);
   num[s] = cnt++;
   QUEUEput( s);

   while (!QUEUEempty( )) {
      vertex v = QUEUEget( );
      for (link a = G->adj[v]; a != NULL; a = a->next)
         if (num[a->w] == -1) {
            num[a->w] = cnt++;
            QUEUEput( a->w);
         }
   }
   QUEUEfree( );
}
```

No início de cada iteração valem as seguinte propriedades:

1. todo vértice que está na fila já foi numerado;
2. se um vértice `v` já foi numerado mas algum de seus vizinhos ainda não foi numerado, então `v` está na fila.

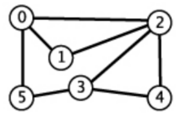
Observe que a fila foi dimensionada corretamente na linha "`QUEUEinit( G->V)`": cada vértice de `G` entra na fila no máximo uma vez e portanto a fila não precisa de mais do que `G->V` posições.

Exemplo A. Considere o grafo `G` definido pelos arcos `0-2 0-3 0-4 1-2 1-4 2-4 3-4 3-5 4-5 5-1`. Suponha que os vértices estão em ordem crescente de nomes em cada lista de adjacência. Submeta `G` à função `GRAPHbfs()` com segundo argumento `0`. Eis o estado da fila (coluna esquerda) e o estado do vetor `num[]` (coluna direita) *no início* de cada iteração:



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

Basta examinar o vetor `num[]` para deduzir que os vértices foram descobertos (e portanto numerados) [na ordem](#) `0 2 3 4 5 1`.



Exemplo B. Neste exemplo, `G` é o grafo [não-dirigido](#) definido pelas arestas `0-1 0-2 0-5 2-1 2-3 2-4 3-4 3-5`. Suponha que os vértices estão em ordem crescente de nomes em todas as listas de adjacência e submeta o grafo à função `GRAPHbfs()` com segundo argumento `0`. Eis o estado da fila e o estado do vetor `num[]` *no início* de cada iteração:

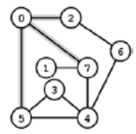
```
queue               0 1 2 3 4 5
0                   0 - - - - -
  1 2 5             0 1 2 - - 3
    2 5             0 1 2 - - 3
      5 3 4         0 1 2 4 5 3
        3 4         0 1 2 4 5 3
          4         0 1 2 4 5 3
                    0 1 2 4 5 3
```

Os vértices foram descobertos (e portanto numerados) na ordem `0 1 2 5 3 4`.

Exercícios 2

1. Faça uma busca em largura a partir do vértice `0` no grafo não-dirigido definido pelas arestas `0-1 1-2 1-4 2-3 2-4 2-9 3-4 4-5 4-6 4-7 5-6 7-8 7-9`. Imagine que o grafo é representado por sua matriz de adjacências e portanto os vizinhos de cada vértice estão em ordem crescente de nomes. Exiba o vetor `num[]` calculado pela busca. Diga em que ordem os vértices foram descobertos.

2. Faça uma busca em largura a partir do vértice `0` do grafo não-dirigido definido pelas arestas `0-2 2-6 6-4 4-5 5-0 0-7 7-1 7-4 3-4 3-5`. Suponha que o grafo é representado por sua matriz de adjacências. Repita a busca começando pelo vértice `4`.



3. *Código de manipulação da fila.* Escreva o código das funções `QUEUEinit()`, `QUEUEput()`, `QUEUEget()`, `QUEUEempty()` e `QUEUEfree()` de manipulação da fila de vértices. Coloque essas funções num [módulo](#) `QUEUE.c` e prepare um arquivo-interface `QUEUE.h`. [[Solução](#)]

4. ★ [Sedgewick 18.52] *Código de manipulação da fila incorporado.* Reescreva a função `GRAPHbfs()` substituindo as invocações de `QUEUEinit()`, `QUEUEput()`, `QUEUEget()`, `QUEUEempty()` e `QUEUEfree()` pelo código apropriado.

5. Mostre um exemplo em que a fila de vértices chega a conter quase todos os vértices do grafo.

6. ★ Depois de executar a função `GRAPHbfs()` com argumentos G e `s`, seja X o conjunto dos vértices `v` para os quais `num[v]` é diferente de `-1`. Descreva os [leques de entrada e de saída](#) de X.

7. Escreva uma versão da função `GRAPHbfs()` para grafos representados por [matriz de adjacências](#).

Árvore de busca em largura

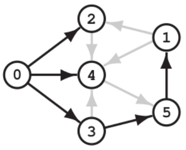
A busca em largura a partir de um vértice `s` constrói uma árvore radicada com raiz `s`: cada arco `v-w` percorrido até um vértice `w` não numerado é acrescentado à árvore. Essa árvore radicada é conhecida como *árvore de busca em largura*, ou *árvore BFS* (= *BFS tree*). Podemos representar essa árvore explicitamente por um vetor de pais `pa[]`. Basta acrescentar algumas linhas ao código de `GRAPHbfs()`:

```
static int num[1000];
static vertex pa[1000];

void GRAPHbfs( Graph G, vertex s)
{
   int cnt = 0;
   for (vertex v = 0; v < G->V; ++v)
      num[v] = pa[v] = -1;
   QUEUEinit( G->V);
   num[s] = cnt++;
   pa[s] = s;
   QUEUEput( s);

   while (!QUEUEempty( )) {
      vertex v = QUEUEget( );
      for (link a = G->adj[v]; a != NULL; a = a->next)
         if (num[a->w] == -1) {
            num[a->w] = cnt++;
            pa[a->w] = v;
            QUEUEput( a->w);
         }
   }
   QUEUEfree( );
}
```

O subgrafo de `G` representado pelo vetor `pa[]` é, de fato, uma árvore radicada pois (1) o vetor `num[]` é uma numeração topológica do subgrafo e (2) no máximo um arco do subgrafo entra em cada vértice. (Veja abaixo o exercício *Numeração topológica da árvore BFS*.)

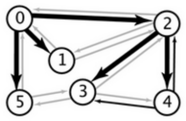
No início de cada iteração, cada vértice na fila é uma folha da árvore radicada representada por `pa[]`.

Exemplo C. Aplique a função `GRAPHbfs()` ao grafo do exemplo A. No fim da execução da função, o vetor `pa[]` estará no seguinte estado:




```
    v    0 1 2 3 4 5
pa[v]    0 5 0 0 0 3
```

Exemplo D. Aplique a função `GRAPHbfs()` ao grafo não-dirigido do exemplo B. No fim da execução da função, o vetor `pa[]` estará no seguinte estado:

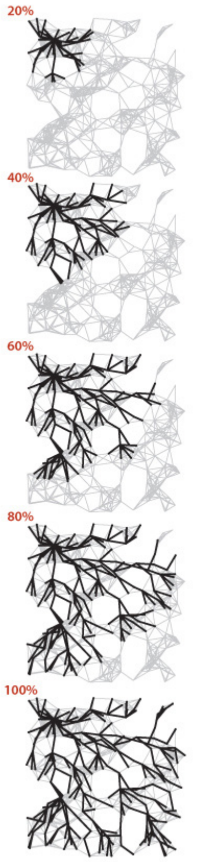



```
    v    0 1 2 3 4 5
pa[v]    0 0 0 2 2 0
```

Exercícios 3

1. Estude a figura ao lado (copiada do livro de Sedgwewick e Wayne). Ela mostra a construção de uma árvore de busca em largura num grafo não-dirigido aleatório com 250 vértices. Note como a árvore radicada cresce "em largura", atingindo gradualmente vértices cada vez mais distantes da raiz.

2. [Sedgewick 18.50, modificado] Faça uma busca em largura a partir do vértice `0` no grafo não-dirigido definido pelas arestas `8-9 3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4`. Suponha que o grafo é representado por sua matriz de adjacências. Faça uma figura da árvore de busca. Repita o exercício depois de representar o grafo por listas de adjacência e inserir as arestas, na ordem dada, num grafo inicialmente vazio.



BFS for shortest paths (250 vertices)

3. Uma árvore de busca em largura é sempre um subgrafo gerador do grafo submetido à busca?

4. ★ *Numeração topológica da árvore BFS.* Mostre que no fim da execução de `GRAPHbfs()` o vetor `num[]` é uma numeração topológica da árvore BFS representada pelo vetor `pa[]`. Para fazer isso, prove os seguintes invariantes: no início de cada iteração (1) `num[q]` ≠ `-1` para cada vértice `q` da fila e (2) `num[pa[w]] < num[w]` para cada vértice `w` tal que `pa[w]` ≠ `-1`.

5. *Propriedades da fila.* No início de uma iteração qualquer de `GRAPHbfs()`, seja q_0, q_1, … , q_k a sequência dos vértices que estão na fila. Mostre que existe um número positivo `L` tal que `num[`q_i`]` ≡ `L+i` para cada `i`. Agora considere a árvore radicada *T* representada pelo vetor `pa[]` e mostre que `num[u]` ≤ `num[q]` para cada `u` em *T* que não está na fila. Deduza daí que, no início de cada iteração, `num[v]` < `num[w]` para cada arco `v-w` tal que `v` ≡ `pa[w]`.

6. Seja `v-w` uma aresta de um grafo não-dirigido *G*. Submeta *G* a uma busca em largura e suponha que `v` é descoberto (e numerado) antes de `w`. É verdade que o arco `v-w` passa a fazer parte da árvore BFS?

7. ★ *BFS não tem arcos de avanço.* Seja *T* uma árvore BFS de um grafo *G*. Mostre que *G* não tem arcos de avanço em relação a *T*. (Embora o conceito de arco de avanço tenha sido definido no contexto da busca DFS, ele faz sentido em relação a qualquer árvore ou floresta radicada de *G*.) Mais precisamente, mostre que se `v-w` é um arco tal que `w` é descendente de `v` em *T* então `v-w` é um arco de *T*.

8. ★ *BFS não-dirigida não tem arcos de retorno.* Seja *T* uma árvore BFS de um grafo *não-dirigido G*. Mostre que todo arco de retorno em relação a *T* é trivial. Mais precisamente, mostre que se `v-w` é um arco tal que `w` é ancestral de `v` em *T* então o arco antiparalelo `w-v` pertence a *T*.

9. ★ *Arcos cruzados em BFS.* Seja *T* uma árvore BFS de um grafo *G*. O grafo pode arcos cruzados em relação a *T*? E se *G* for não-dirigido?

10. *Altura da árvore.* Escreva uma variante da função `GRAPHbfs()` que calcule a altura da árvore de busca em largura.

Desempenho

Quanto tempo a função `GRAPHbfs()` consome para processar um grafo com *V* vértices e *A* arcos? Cada iteração em "`while (!QUEUEempty())`" tira um vértice `v` da fila e percorre os arcos do leque de saída de `v`. Como cada vértice entra na fila e sai da fila no máximo uma vez, ⚠ cada arco do grafo é percorrido no máximo uma vez durante a execução do `while`. Assim, a execução de todas as iteração consome tempo proporcional a *A* no pior caso. O resto do código consome tempo proporcional a *V*. Portanto, o consumo total de tempo é proporcional a

$$V + A$$

no pior caso, supondo que o grafo é representado por um listas de adjacência. (Se $A \geq V$, como acontece em muitas aplicações, podemos dizer que o consumo de tempo é proporcional a *A*.) Esse tempo é proporcional ao tamanho do grafo e portanto podemos dizer que `GRAPHbfs()` é linear.

Se o grafo for representado por matriz de adjacências, uma análise semelhante mostra que o consumo de tempo é proporcional a

$$V^2$$

no pior caso. Se nos restrigirmos a grafos densos, esse consumo é linear.

Exercícios 4

1. ★ *BFS com múltiplas origens.* Dado um conjunto S de vértices de um grafo, faça uma busca em largura *a partir de* S. (A busca visita todos os vértices que estão ao alcance de S, isto é, todos os vértices t que estão na ponta final de algum caminho que começa em S.) Sua função deve também construir a *floresta* da busca.
2. Escreva uma versão da função `GRAPHbfs()` em que a fila armazena *arcos* e não vértices.
3. Escreva uma versão de `GRAPHbfs()` sem o vetor `num[]`: o vetor `pa[]` é suficiente para controlar a lógica da função.
4. Escreva uma versão recursiva da busca em largura. (Um tanto ridículo, mas pode ser um bom exercício.)

BFS versus DFS

Apesar da semelhança entre a siglas, a busca BFS e a busca DFS são muito diferentes e têm aplicações muito diferentes.

A diferença mais marcante entre as duas buscas está nas estruturas de dados auxiliares empregadas pelas duas estratégias. A BFS usa uma fila (de vértices), enquanto a DFS usa uma pilha. (Na versão recursiva da DFS, a pilha não aparece explicitamente porque é administrada pelo mecanismo de recursão.)

Outras diferenças entre os dois algoritmos são mais superficiais:

- na BFS, o usuário escolhe o vértice inicial; na DFS o próprio algoritmo escolhe o vértice inicial de cada etapa;
- a DFS visita todos os vértices do grafo, enquanto a BFS visita apenas os vértices que estão ao alcance do vértice inicial;
- em geral, a DFS é descrita em estilo recursivo enquanto a BFS é descrita em estilo iterativo.

---

www.ime.usp.br/~pf/algoritmos_para_grafos/
Atualizado em 2019-08-22
Paulo Feofiloff
IME-USP