Análise da complexidade temporal de BFS (1)

Grafo implementado através de listas de adjacências

BFS(G, s)

```
1  for each vertex u in G.V - {s} do
2      u.color <- WHITE
3      u.d <- INFINITY
4      u.p <- NIL
```

- Ciclo das linhas 1–4 é executado $|V| - 1$ vezes

```
5  s.color <- GREY
6  s.d <- 0
7  s.p <- NIL
8  Q <- EMPTY                        // fila (FIFO)
9  ENQUEUE(Q, s)
```

- Linhas 5–9 com custo constante

Análise da complexidade temporal de BFS (2)

- Ciclo das linhas 10–18 é executado $|V|$ vezes, no pior caso

```
10  while Q != EMPTY do
11      u <- DEQUEUE(Q)                 // próximo nó a explorar
12      for each vertex v in G.adj[u] do
13          if v.color = WHITE then
14              v.color <- GREY
15              v.d <- u.d + 1
16              v.p <- u
17              ENQUEUE(Q, v)
18      u.color <- BLACK                // u foi explorado
```

- Mas o ciclo das linhas 12–17 é executado, no pior caso

$$\sum_{u \in V} |\,\texttt{G.adj}[u]\,| = |E| \text{ (orientado) ou } 2 \times |E| \text{ (não orientado) vezes}$$

porque cada vértice só pode entrar na fila uma vez

Análise da complexidade temporal de BFS (3)

Considerando que todas as operações, incluindo a criação de uma fila vazia, ENQUEUE e DEQUEUE, têm custo $\Theta(1)$

- O ciclo das linhas 1–4 tem custo $\Theta(V)$
- Conjuntamente, os ciclos das linhas 10–18 e 12–17 têm custo $O(E)$ (pior caso)

Logo, a complexidade temporal de BFS é $O(V + E)$

Análise da complexidade temporal de BFS (4)

Grafo implementado através da matriz de adjacências

Na linha 12, é necessário percorrer uma linha da matriz, com $|V|$ elementos

```
12'     for each vertex v in G.V do
13'         if G.adjm[u,v] and v.color = WHITE then
```

Como o ciclo das linhas 10–18 é executado $|V|$ vezes, no pior caso, o custo combinado dos dois ciclos é $O(V^2)$

- Corresponde a aceder a todas as posições de uma matriz $|V| \times |V|$

Neste caso, a complexidade temporal de BFS será $O(V^2)$

Complexidade espacial de BFS

Memória usada pelo algoritmo

- 2 variáveis para vértices (`u` e `v`)

$$O(1)$$

- 3 valores escalares (`color`, `d` e `p`) por cada vértice

$$\Theta(V)$$

- Uma fila, que poderá ter, no pior caso, $|V| - 1$ vértices

$$O(V)$$

A complexidade espacial de BFS é $\Theta(V)$

BFS em Java (1)

```
public static final int INFINITY = Integer.MAX_VALUE;
public static final int NONE = -1;

private static enum Colour { WHITE, GREY, BLACK };

public int[] bfs(int s)
{
  Colour[] colour = new Colour[nodes];
  int[] d = new int[nodes];          // distância para S
  int[] p = new int[nodes];          // predecessor no caminho de S

  for (int u = 0; u < nodes; ++u)
    {
      colour[u] = Colour.WHITE;
      d[u] = INFINITY;
      p[u] = NONE;
    }

  colour[s] = Colour.GREY;
  d[s] = 0;

  Queue<Integer> Q = new LinkedList<>();

  Q.add(s);

  ...
```

BFS em Java (2)

```
  ...
  while (!Q.isEmpty())
    {
      int u = Q.remove();                          // visita nó U

      for (Integer v : adjacents[u])
        if (colour[v] == Colour.WHITE)
          {
            colour[v] = Colour.GREY;               // V é um novo nó
            d[v] = d[u] + 1;
            p[v] = u;

            Q.add(v);
          }

      colour[u] = Colour.BLACK;                    // nó U está tratado
    }

    return d ou p ou ...
}
```

Percurso em profundidade

DFS(G)

```
for each vertex u in G.V do
    u.color <- WHITE
    u.p <- NIL
time <- 0                               // variável global
for each vertex u in G.V do             // explora todos os nós
    if u.color = WHITE then
        DFS-VISIT(G, u)
```

DFS-VISIT(G, u)

```
time <- time + 1                   // instante da descoberta do
u.d <- time                        // vértice u
u.color <- GREY
for each vertex v in G.adj[u] do // explora arco (u, v)
    if v.color = WHITE then
        v.p <- u
        DFS-VISIT(G, v)
u.color <- BLACK                   // u foi explorado
time <- time + 1                   // instante em que termina
u.f <- time                        // a exploração de u
```

Percurso em profundidade

Depth-first search

Constrói a floresta da pesquisa em profundidade (linhas 3 [DFS] e 6 [DFS-VISIT])

Atributos dos vértices

`color`	`WHITE`	não descoberto
	`GREY`	descoberto e em processamento
	`BLACK`	processado
`d`	instante em que foi descoberto	
`f`	instante em que terminou de ser processado	
`p`	antecessor do nó no caminho que levou à sua descoberta	

Análise da complexidade temporal de DFS

O ciclo das linhas 1–3 [DFS] é executado $|V|$ vezes

DFS-VISIT é chamada para cada um dos $|V|$ vértices

Para cada vértice u (e considerando a implementação através de listas de adjacências), o ciclo das linhas 4–7 [DFS-VISIT] é executado

$$|\,\texttt{G.adj}[u]\,| \quad \text{vezes}$$

Tendo todas as operações custo constante, considerando todas as chamadas a DFS-VISIT, DFS corre em tempo

$$\Theta(V + \sum_{u \in V} |\,\texttt{G.adj}[u]\,|) = \Theta(V + E)$$

Complexidade espacial de DFS

Memória usada pelo algoritmo

- 4 valores escalares (`color`, `d`, `f` e `p`) por cada vértice

$$\Theta(V)$$

- Uma pilha, que poderá ter, no pior caso, $|V|$ vértices

$$O(V)$$

  A pilha será implícita, quando algoritmo for implementado recursivamente, ou explícita, quando for implementado iterativamente

A complexidade espacial de DFS é $\Theta(V)$

Complexidades dos percursos

Resumo

$G = (V, E)$

	Complexidade Temporal		Complexidade Espacial
Percurso em largura	$O(V + E)$	$O(V^2)$	$\Theta(V)$
Percurso em profundidade	$\Theta(V + E)$	$\Theta(V^2)$	$\Theta(V)$
	Listas de adjacências	Matriz de adjacências	

Se, no percurso em largura, for percorrido todo o grafo (como é feito no percurso em profundidade), as complexidades temporais também serão $\Theta(V + E)$ e $\Theta(V^2)$, respectivamente

Se o percurso em profundidade for feito a partir de um único nó (como no percurso em largura), as complexidades temporais também serão $O(V + E)$ e $O(V^2)$, respectivamente

Ordenação topológica

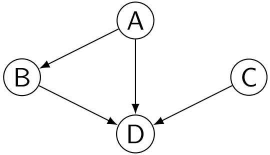
Seja $G = (V, E)$ um grafo orientado acíclico (DAG, de *directed acyclic graph*)

Ordem topológica

Se existe um arco de u para v, u está antes de v na ordem dos vértices

$$(u, v) \in E \;\Rightarrow\; u < v$$

Exemplo



Ordem

$A < B \quad A < D \quad B < D \quad C < D$

Ordenações possíveis

- A B C D
- A C B D
- C A B D

Ordenação topológica

Um algoritmo baseado no percurso em profundidade

Aplicação do percurso em profundidade

- Se não há caminho de u para nenhum outro vértice, u poderá ser o último vértice da ordenação topológica
- Se há um caminho de u para outro vértice v, então u estará antes de v, em qualquer ordenação topológica

TOPOLOGICAL-SORT(G)

1. Aplicar DFS(G)
2. Durante o percurso, quando termina o processamento de um vértice, inseri-lo à cabeça de uma lista
3. Devolver a lista, que contém os vértices por (alguma) ordem topológica

Ordenação topológica

Adaptação de DFS

$G = (V, E)$ – grafo orientado acíclico (DAG)

TOPOLOGICAL-SORT(G)

```
for each vertex u in G.V do
    u.color <- WHITE
L <- EMPTY                              // lista, global
for each vertex u in G.V do
    if u.color = WHITE then
        DFS-VISIT'(G, u)
return L
```

DFS-VISIT'(G, u)

```
u.color <- GREY
for each vertex v in G.adj[u] do
    if v.color = WHITE then
        DFS-VISIT'(G, v)
u.color <- BLACK
LIST-INSERT-HEAD(L, u)
```

Ordenação topológica
Outro algoritmo (1)

- Se u não é o destino de nenhum arco, u pode ser o primeiro nó da ordenação topológica
- Uma vez ordenados todos os vértices u tais que $(u, v) \in E$, o vértice v pode ser colocado a seguir

Ordenação topológica

Outro algoritmo (2)

TOPOLOGICAL-SORT'(G)

```
for each vertex u in G.V do
    u.i <- 0
for each edge (u,v) in G.E do
    v.i <- v.i + 1               // arcos incidentes em v
L <- EMPTY                       // lista
S <- EMPTY                       // conjunto
for each vertex u in G.V do
    if u.i = 0 then
        SET-INSERT(S, u)
while S != EMPTY do
    u <- SET-DELETE(S)           // retira um nó de S
    for each vertex v in G.adj[u] do
        v.i <- v.i - 1
        if v.i = 0 then
            SET-INSERT(S, v)
    LIST-INSERT-TAIL(L, u)
return L
```

Complexidade dos algoritmos

$G = (V, E)$

	Compl. Temporal
Percurso em largura	$O(V + E)$
Percurso em profundidade	$\Theta(V + E)$
Ordenação topológica (ambos os algoritmos)	$\Theta(V + E)$

Pressupostos

Grafo representado através de listas de adjacências

Se, no percurso em largura, for percorrido todo o grafo (como é feito no percurso em profundidade), a complexidade temporal também será $\Theta(V + E)$

Conectividade (1)

Seja $G = (V, E)$ um grafo não orientado

G é conexo se existe algum caminho entre quaisquer dois nós:

$u, v \in V \Rightarrow$ existe (pelo menos) um caminho $v_0 v_1 \ldots v_k$, $k \geq 0$,
com $v_0 = u$ e $v_k = v$

$V' \subseteq V$ é uma componente conexa de G se

- existe algum caminho entre quaisquer dois nós de V' e
- não existe qualquer caminho entre algum nó de V' e algum nó de $V \setminus V'$

Conectividade (2)

Seja $G = (V, E)$ um grafo orientado

G é fortemente conexo se existe algum caminho de qualquer nó para qualquer outro nó

$V' \subseteq V$ é uma componente fortemente conexa de G se

- existe algum caminho de qualquer nó de V' para qualquer outro nó de V' e
- se, qualquer que seja o nó $u \in V \setminus V'$
  - não existe qualquer caminho de algum nó de V' para u ou
  - não existe qualquer caminho de u para algum nó de V'

G é simplesmente conexo se, substituindo todos os arcos por arcos não orientados, se obtém um grafo conexo