# Algoritmos e Estruturas de Dados II - SCE-183

## Grafos: Busca

Gustavo Batista

## Busca em Grafos

- Dado um grafo $G = (V, A)$ e um vértice $v \in V$, deseja-se encontrar todos os vértices em $G$ que estão conectados a $v$.
- Ou, dado um grafo $G = (V, A)$, deseja-se visitar todos os vértices de $G$.
- Serão vistas duas maneiras de realizar essas tarefas:
  - Busca em profundidade, e;
  - Busca em largura.

## Busca em Profundidade

- A **busca em profundidade** (*depth-first search*) é um algoritmo para caminhar no grafo.
- A estratégia é buscar o vértice mais profundo no grafo sempre que possível.
- As arestas são exploradas a partir do vértice $v$ mais recentemente descoberto que ainda possui arestas não exploradas saindo dele.

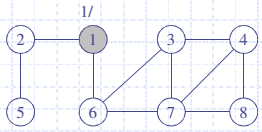
## Busca em Profundidade

- Quando todas as arestas adjacentes a $v$ tiverem sido exploradas a busca anda para trás para explorar vértices que saem do vértice do qual $v$ foi descoberto.
- O algoritmo é a base para muitos outros algoritmos importantes, tais como verificação de grafos acíclicos, ordenação topológica e componentes fortemente conectados.

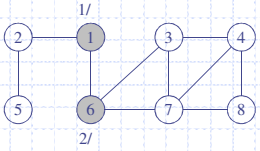
## Busca em Profundidade

- Para acompanhar o progresso do algoritmo cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicialmente brancos.
- Quando um vértice é "descoberto" pela primeira vez ele torna-se cinza, e torna-se preto quando seus adjacentes tenham sido completamente examinados.

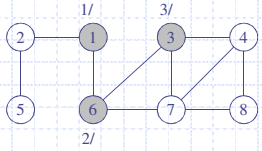
## Busca em Profundidade



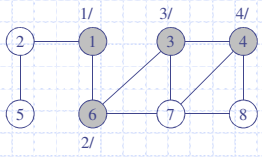

## Busca em Profundidade



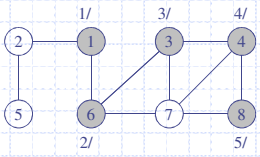

## Busca em Profundidade



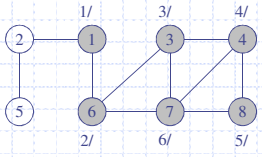
## Busca em Profundidade



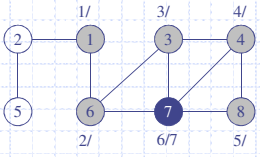

## Busca em Profundidade



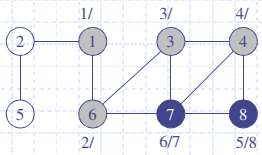

## Busca em Profundidade



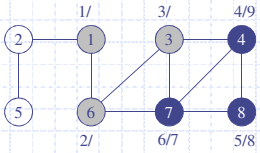

## Busca em Profundidade



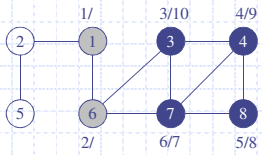
## Busca em Profundidade



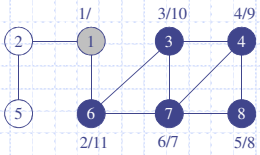

## Busca em Profundidade



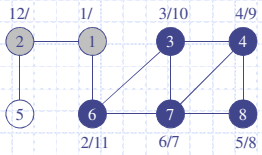

## Busca em Profundidade



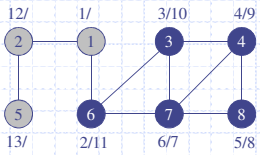

## Busca em Profundidade



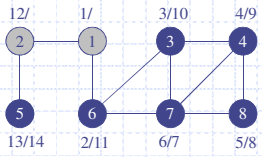
## Busca em Profundidade



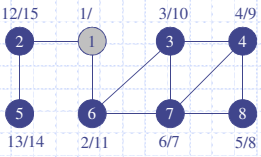

## Busca em Profundidade



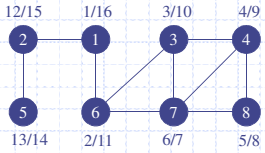

## Busca em Profundidade




## Busca em Profundidade



## Busca em Profundidade




## Busca em Profundidade

```
void busca_profundidade(tgrafo *grafo) {
    tvertice v;
    int cor[MAXNUMVERTICES];

    for (v = 0; v < grafo->num_vertices; v++)
        cor[v] = BRANCO;
    for (v = 0; v < grafo->num_vertices; v++)
        if (cor[v] == BRANCO)
            visita_dfs(v, cor, grafo);
}
```

```
#define BRANCO 0
#define CINZA 1
#define PRETO 2
```

## Busca em Profundidade

```
void visita_dfs(tvertice v, int cor[], tgrafo *grafo) {
    tvertice w;
    tapontador p;
    tpeso peso;

    cor[v] = CINZA;
    p = primeiro_adj(v, grafo);
    while (p != NULO) {
        recupera_adj(v, p, &w, &peso, grafo);
        if (cor[w] == BRANCO)
            visita_dfs(w, cor, grafo);
        p = proximo_adj(v, p, grafo);
    }
    cor[v] = PRETO;
}
```

## Busca em Profundidade - Complexidade

- O procedimento BuscaProfundidade requer $O(|V|)$ para inicializar o vetor cor.
- Quando uma matriz de adjacências é utilizada, o procedimento VisitaDfs requer $O(|V|^2)$. Logo a busca em profundidade requer $O(|V|+|V|^2)$.
- Quando uma lista de adjacências é utilizada, VisitaDfs requer $O(|A|)$ e a busca profundidade requer $O(|V|+|A|)$.

## Busca em Largura

- A busca em largura (*breadth-first search*) expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a um distância *k* do vértice origem antes de descobrir qualquer vértice a uma distância *k+1*.

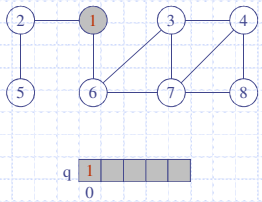
## Busca em Largura

- Cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicialmente brancos.
- Quando um vértice é "descoberto" pela primeira vez ele torna-se cinza.
- Vértices cinza e preto já foram "descobertos", mas são distinguidos para assegurar que a busca ocorra em largura.

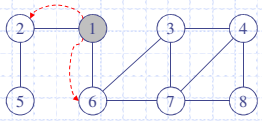
## Busca em Largura

- Se $(u,v) \in A$ e o vértice $u$ é preto, então o vértice $v$ tem que ser cinza ou preto.
- Vértices cinza podem ter alguns vértices adjacentes brancos, e eles representam a fronteira entre vértices "descobertos" e não "descobertos".

## Busca em Largura

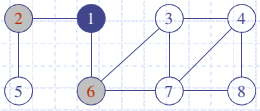


## Busca em Largura



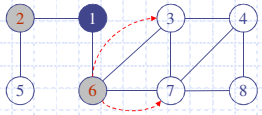

q [1]
0

## Busca em Largura



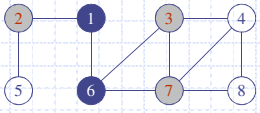

q [6] [2]
1 1

## Busca em Largura



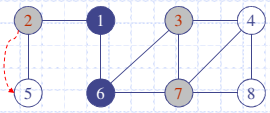

q [6] [2]
1 1

## Busca em Largura



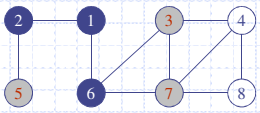

q [2] [3] [7]
1 2 2

## Busca em Largura



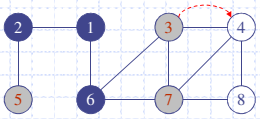

q: 2 3 7
1 2 2

## Busca em Largura



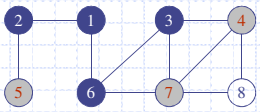

q: 3 7 5
2 2 2

## Busca em Largura



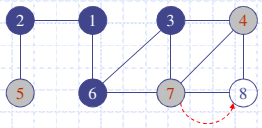

q: 3 7 5
2 2 2

## Busca em Largura




q: 7 5 4
2 2 3

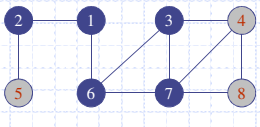
## Busca em Largura




q: ~~2~~ 5 4

2 2 3

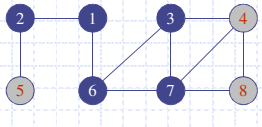
## Busca em Largura




q: 5 4 8

2 3 3

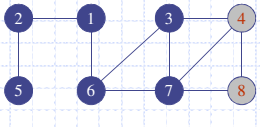
## Busca em Largura




q: ~~5~~ 4 8

2 3 3

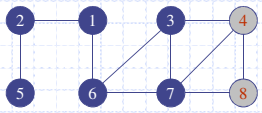
## Busca em Largura




q: 4 8

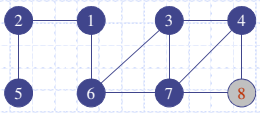
3 3

## Busca em Largura



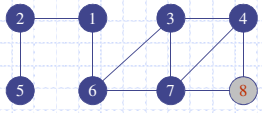

q: ~~4~~ 8
3 3

## Busca em Largura



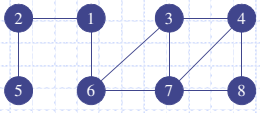

q: 8
3

## Busca em Largura




q: ~~8~~
3

## Busca em Largura




q:

## Busca em Largura

```
void busca_largura(tgrafo *grafo) {
    tvertice v;
    int cor[MAXNUMVERTICES];

    for (v = 0; v < grafo->num_vertices; v++)
        cor[v] = BRANCO;
    for (v = 0; v < grafo->num_vertices; v++)
        if (cor[v] == BRANCO)
            visita_bfs(v, cor, grafo);
}
```

## Busca em Largura

```
void visita_bfs(tvertice v, int cor[], tgrafo *grafo) {
    tvertice w;
    tapontador p;
    tpeso peso;
    std::queue<tvertice> q;

    cor[v] = CINZA;
    q.push(v);
    while (!q.empty()) {
        v = q.front(); q.pop();
        p = primeiro_adj(v, grafo);
        while (p != NULO) {
            recupera_adj(v, p, &w, &peso, grafo);
            if (cor[w] == BRANCO) {
                cor[w] = CINZA;
                q.push(w);
            }
            p = proximo_adj(v, p, grafo);
        }
        cor[v] = PRETO;
    }
}
```

## Busca em Largura - Complexidade

- Colorir todos os vértices de branco é $O(|V|)$.
- Cada vértice entra na fila $q$ exatamente uma vez, portanto o laço **enquanto** faz $|V|$ iterações.
- Se uma matriz de adjacências é utilizada, é necessário $O(|V|)$ para cada nó visitado. O tempo total é $O(|V|+|V|^2)$.
- Se lista de adjacências é utilizada, o custo do laço é $d_1+...+d_n = O(|V|+|A|)$.

## Exercício

- Implementar a busca em profundidade e largura na linguagem de programação de sua preferência utilizando as operações do TAD Grafo implementadas.
- Implementar a busca em profundidade iterativa utilizando o algoritmo de busca em largura com uma pilha ao invés de fila. Comente a mudança na coloração dos vértices.