ACH2024 –

Algoritmos e Estruturas de Dados II

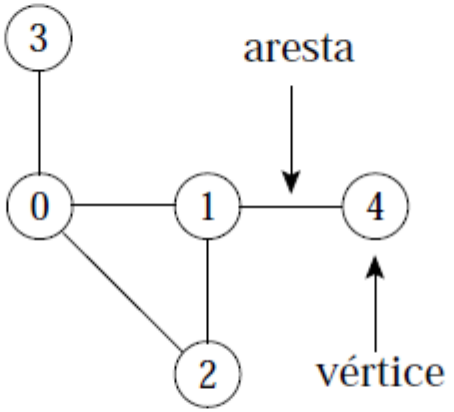
Prof. Helton Hideraldo Biscaro
heltonhb@usp.br

Parte 1 da Disciplina:

GRAFOS

Grafos

- **Grafo**: conjunto de vértices e arestas.
- **Vértice**: objeto simples que pode ter nome e outros atributos.
- **Aresta**: conexão entre dois vértices.



- Notação: $G = (V, A)$
  - G: grafo
  - V: conjunto de vértices
  - A: conjunto de arestas

Grafos

Tipo Abstrato de dados Grafo

- Importante considerar os algoritmos em grafos como **tipos abstratos de dados**.
- Conjunto de operações associado a uma estrutura de dados.
- Independência de implementação para as operações.

Operadores do TAD Grafo

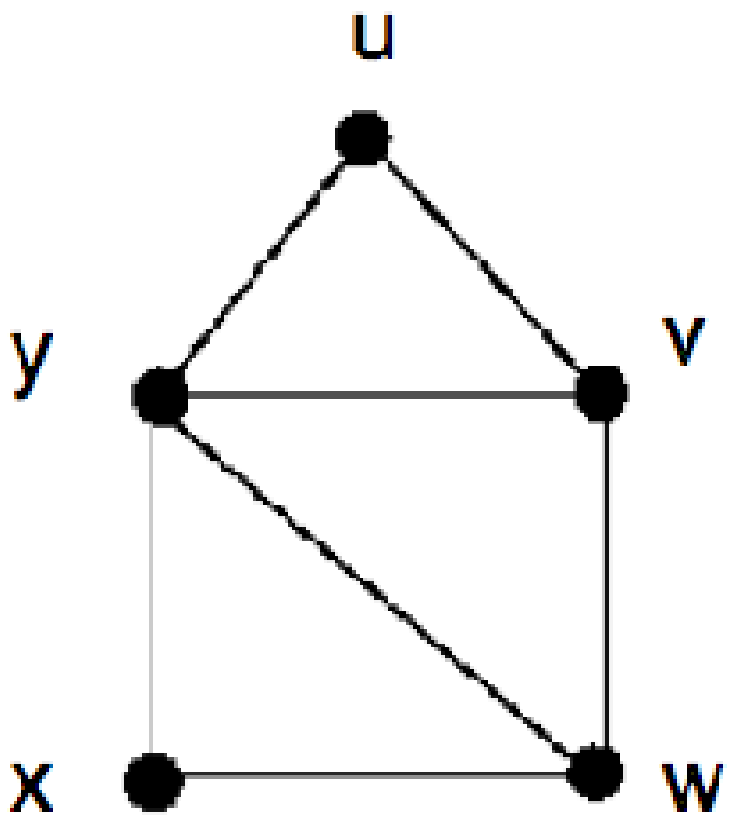
1. *FGVazio(Grafo)*: Cria um grafo vazio.
2. *InsereAresta(V1,V2,Peso, Grafo)*: Insere uma aresta no grafo.
3. *ExisteAresta(V1,V2,Grafo)*: Verifica se existe uma determinada aresta.
4. Obtem a lista de vértices adjacentes a um determinado vértice (tratada a seguir).
5. *RetiraAresta(V1,V2,Peso, Grafo)*: Retira uma aresta do grafo.
6. *LiberaGrafo(Grafo)*: Liberar o espaço ocupado por um grafo.
7. *ImprimeGrafo(Grafo)*: Imprime um grafo.
8. *GrafoTransposto(Grafo,GrafoT)*: Obtém o transposto de um grafo direcionado.
9. *RetiraMin(A)*: Obtém a aresta de menor peso de um grafo com peso nas arestas.

Representação de Grafos

1) Listas de Adjacências

1) Matriz de Adjacências

Listas de Adjacências

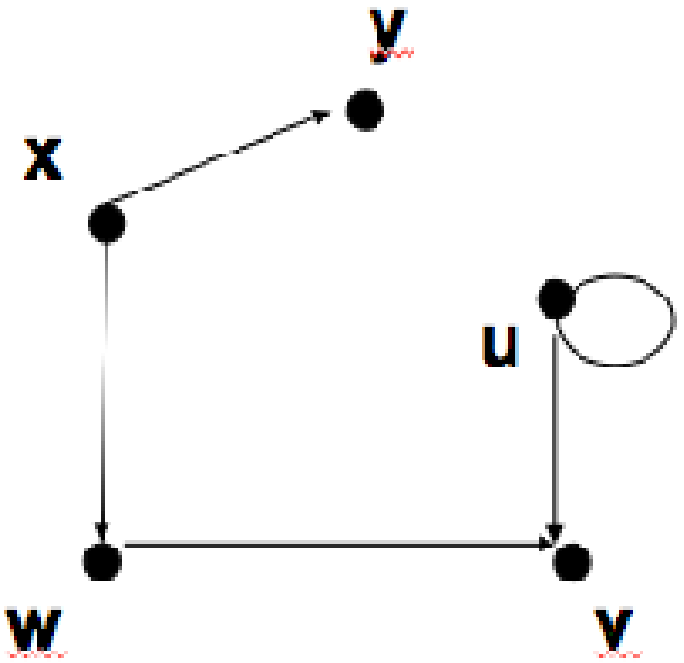


u	v, y
v	u, y, w
w	v, x, y
x	w, y
y	u, v, w, x

Listas de Adjacências

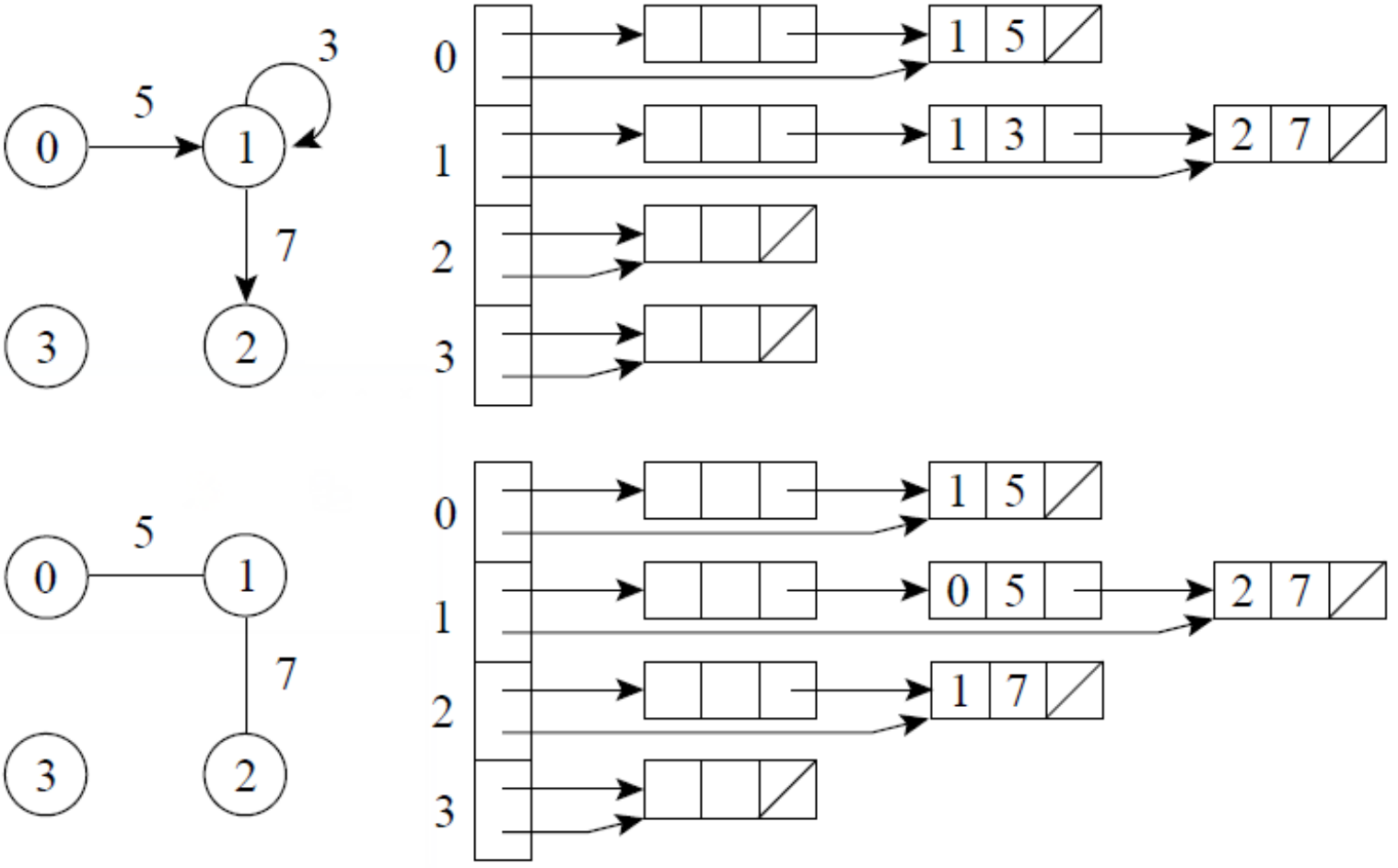
Grafos Direcionados:

Tabela com vértices iniciais e finais (terminais)



Iniciais	Terminais
u	u,v
v	
w	v
x	w,y
y	

Listas de Adjacências



Listas de Adjacências

- Os vértices de uma lista de adjacência são em geral armazenados em uma ordem arbitrária.
- Possui uma complexidade de espaço $O(|V| + |A|)$
- Indicada para grafos **esparsos**, onde $|A|$ é muito menor do que $|V|^2$.
- É compacta e usualmente utilizada na maioria das aplicações.
- A principal desvantagem é que ela pode ter tempo $O(|V|)$ para determinar se existe uma aresta entre o vértice i e o vértice j, pois podem existir $O(|V|)$ vértices na lista de adjacentes do vértice i.

Lista de Adjacências – Ponteiros

```
#define MAXNUMVERTICES   100
#define MAXNUMARESTAS   4500
typedef int TipoValorVertice;
typedef int TipoPeso;
typedef struct TipoItem {
   TipoValorVertice Vertice;
   TipoPeso Peso;
} TipoItem;
typedef struct TipoCelula *TipoApontador;
struct TipoCelula {
   TipoItem Item;
   TipoApontador Prox;
} TipoCelula;
```

Lista de Adjacências – Ponteiros

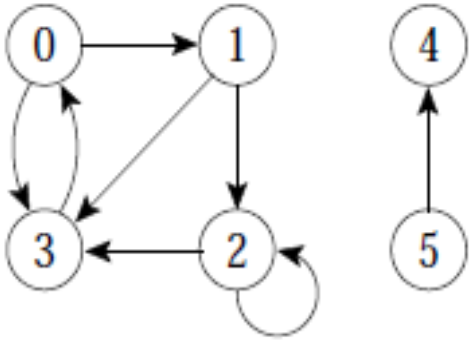
```
typedef struct TipoLista {
   TipoApontador Primeiro, Ultimo;
} TipoLista;
typedef struct TipoGrafo {
   TipoLista Adj[MAXNUMVERTICES + 1];
   TipoValorVertice NumVertices;
   short NumArestas;
} TipoGrafo;
```

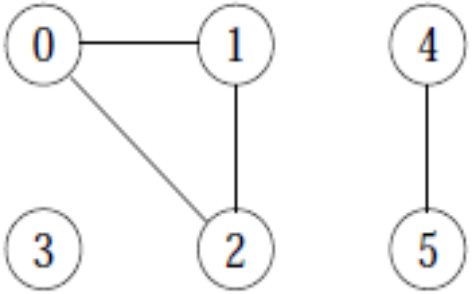
- No uso de apontadores a lista é constituída de células, onde cada célula contém um item da lista e um apontador para a célula seguinte.

Matriz de Adjacências

- A matriz de adjacência de um grafo $G = (V, A)$ contendo n vértices é uma matriz $n \times n$ de *bits*, onde $A[i, j]$ é 1 (ou verdadeiro) se e somente se existe um arco do vértice i para o vértice j.
- Para grafos ponderados $A[i, j]$ contém o rótulo ou peso associado com a aresta e, neste caso, a matriz não é de *bits*.
- Se não existir uma aresta de i para j então é necessário utilizar um valor que não possa ser usado como rótulo ou peso.

Matriz de Adjacências - Exemplo





	0	1	2	3	4	5
0		1		1		
1			1	1		
2			1	1		
3	1					
4						
5						

(a)

	0	1	2	3	4	5
0		1	1			
1	1		1			
2	1	1				
3						
4						
5						

(b)

Matriz de Adjacências - Análise

- Deve ser utilizada para grafos **densos**, onde $|A|$ é próximo de $|V|^2$.
- O tempo necessário para acessar um elemento é independente de $|V|$ ou $|A|$.
- É muito útil para algoritmos em que necessitamos saber com rapidez se existe uma aresta ligando dois vértices.
- A maior desvantagem é que a matriz necessita $\Omega(|V|^2)$ de espaço. Ler ou examinar a matriz tem complexidade de tempo $O(|V|^2)$.

Matriz de Adjacências - Operadores

```
void FGVazio(TipoGrafo *Grafo)
{ short i, j;
  for (i = 0; i <= Grafo->NumVertices; i++)
    { for (j = 0; j <=Grafo->NumVertices; j++) Grafo->Mat[i][j] = 0; }
}

void InsereAresta(TipoValorVertice *V1, TipoValorVertice *V2,
                  TipoPeso *Peso, TipoGrafo *Grafo)
{ Grafo->Mat[*V1][*V2] = *Peso; }

short ExisteAresta(TipoValorVertice Vertice1,
                   TipoValorVertice Vertice2, TipoGrafo *Grafo)
{ return (Grafo->Mat[Vertice1][Vertice2] > 0); }
```

Matriz de Adjacências – Operadores

```
void RetiraAresta(TipoValorVertice *V1, TipoValorVertice *V2,
                  TipoPeso *Peso, TipoGrafo *Grafo)
{ if (Grafo->Mat[*V1][*V2] == 0) printf("Aresta nao existe\n");
  else { *Peso = Grafo->Mat[*V1][*V2]; Grafo->Mat[*V1][*V2] = 0; }
}
void LiberaGrafo(TipoGrafo *Grafo)
{  /* Nao faz nada no caso de matrizes de adjacencia */ }
void ImprimeGrafo(TipoGrafo *Grafo)
{ short i, j; printf("    ");
  for (i = 0; i <= Grafo->NumVertices - 1; i++) printf("%3d", i);
  printf("\n");
  for (i = 0; i <=  Grafo->NumVertices - 1; i++)
    { printf("%3d", i);
      for (j = 0; j <=Grafo->NumVertices - 1; j++)
        printf("%3d", Grafo->Mat[i][j]);
      printf("\n");
    }
}
```

Exercício:

Escrever um algoritmo em c que verifica se existe um caminho do vértice x até o vértice y, utilizando a representação de Matriz de Adjacências.

Busca em Profundidade

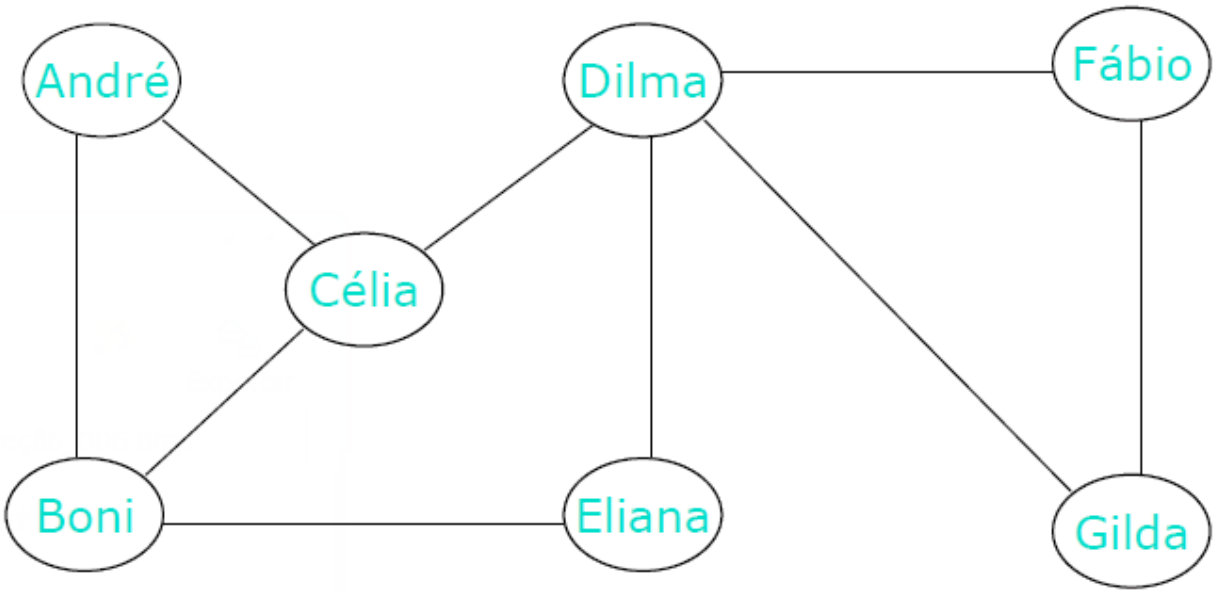
- A busca em profundidade, do inglês *depth-first search*), é um algoritmo para caminhar no grafo.
- A estratégia é buscar o mais profundo no grafo sempre que possível.
- As arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda possui arestas não exploradas saindo dele.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

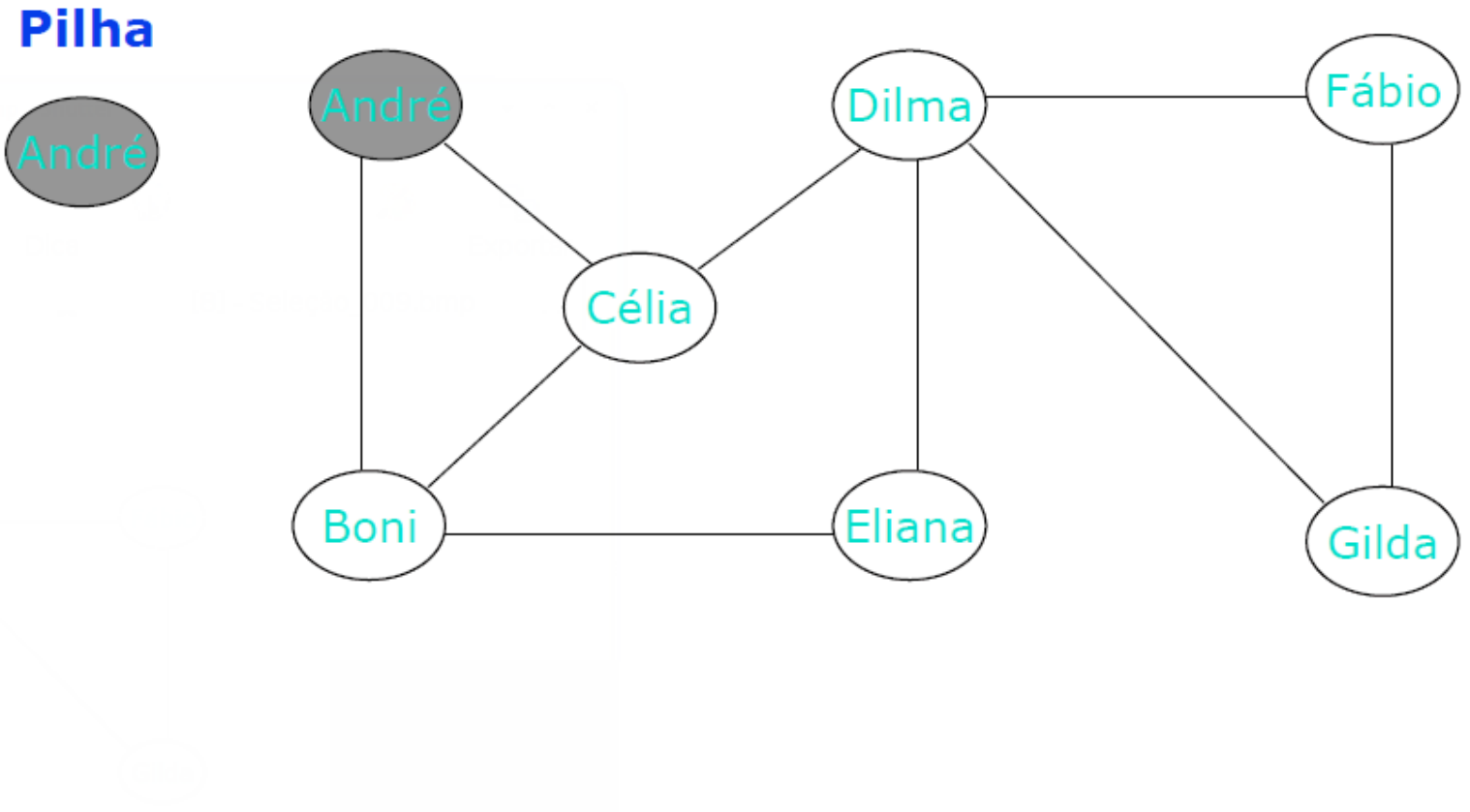
- Para acompanhar o progresso do algoritmo cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é *descoberto* pela primeira vez ele torna-se cinza, e é tornado preto quando sua lista de adjacentes tenha sido completamente examinada.

Busca em Profundidade

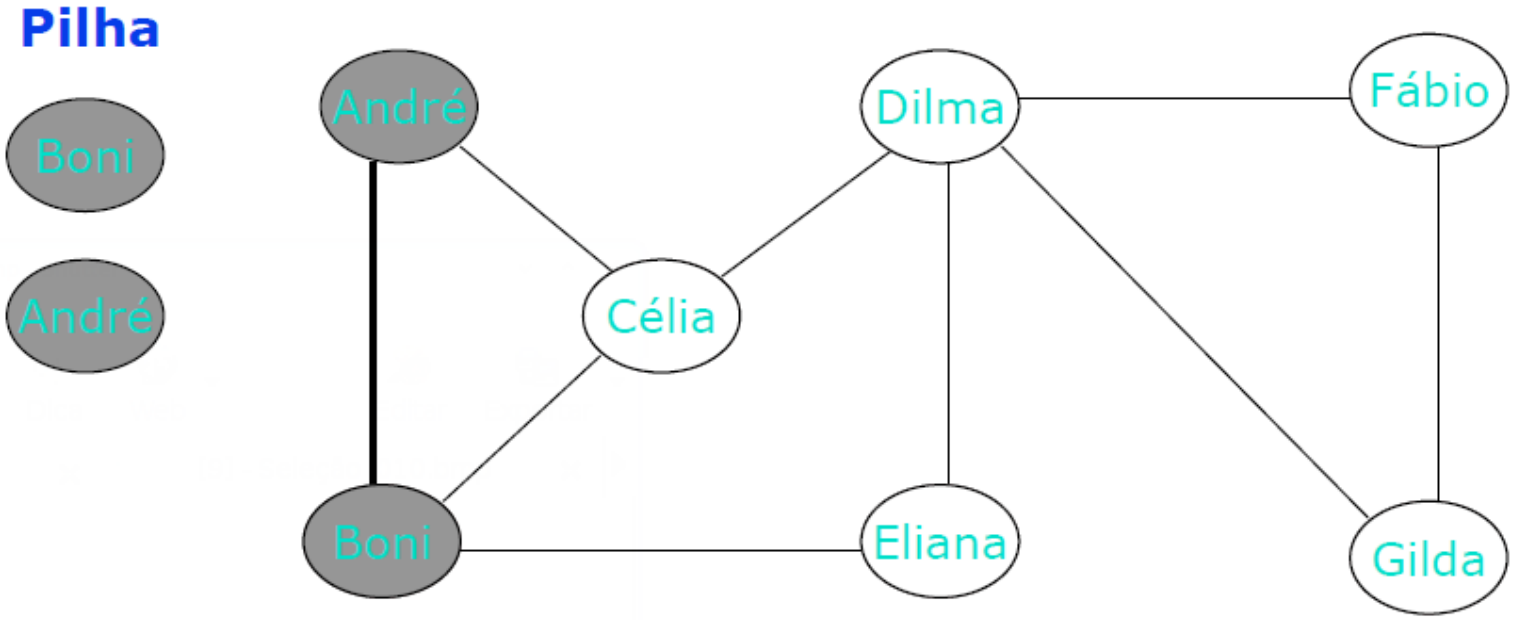
Pilha



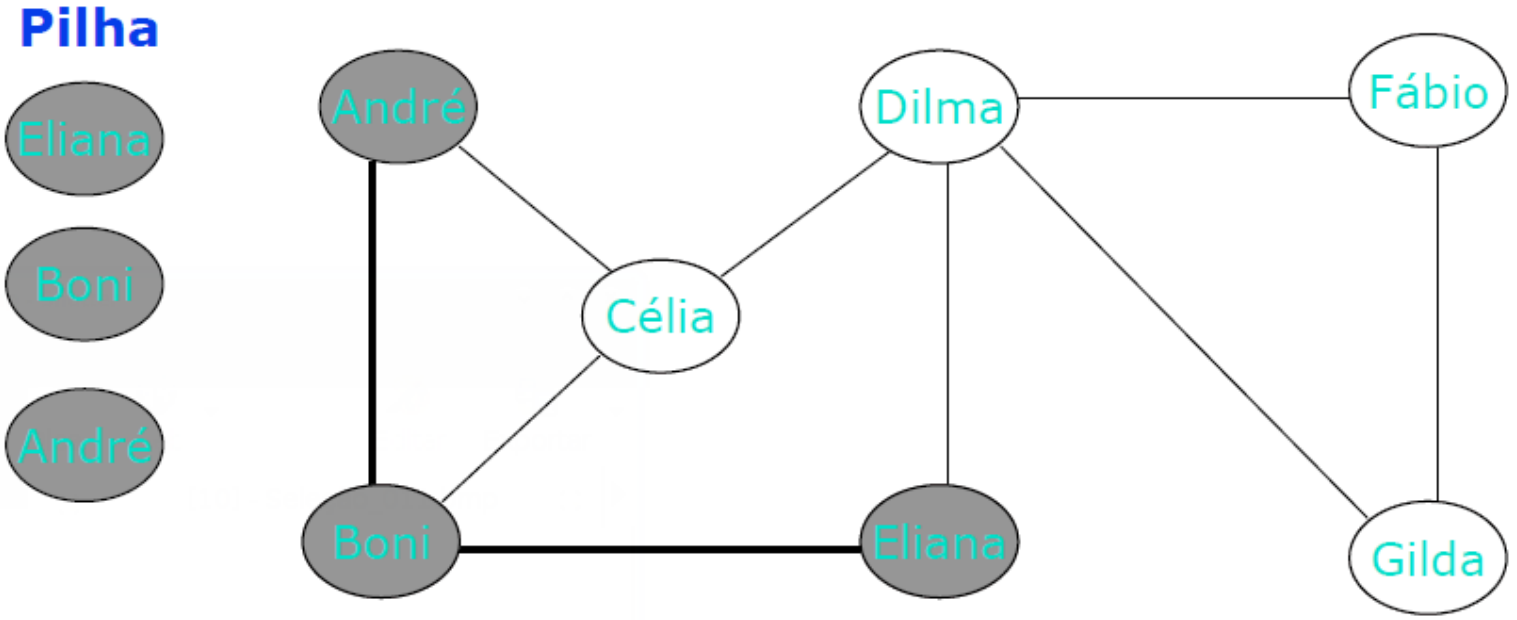
Busca em Profundidade



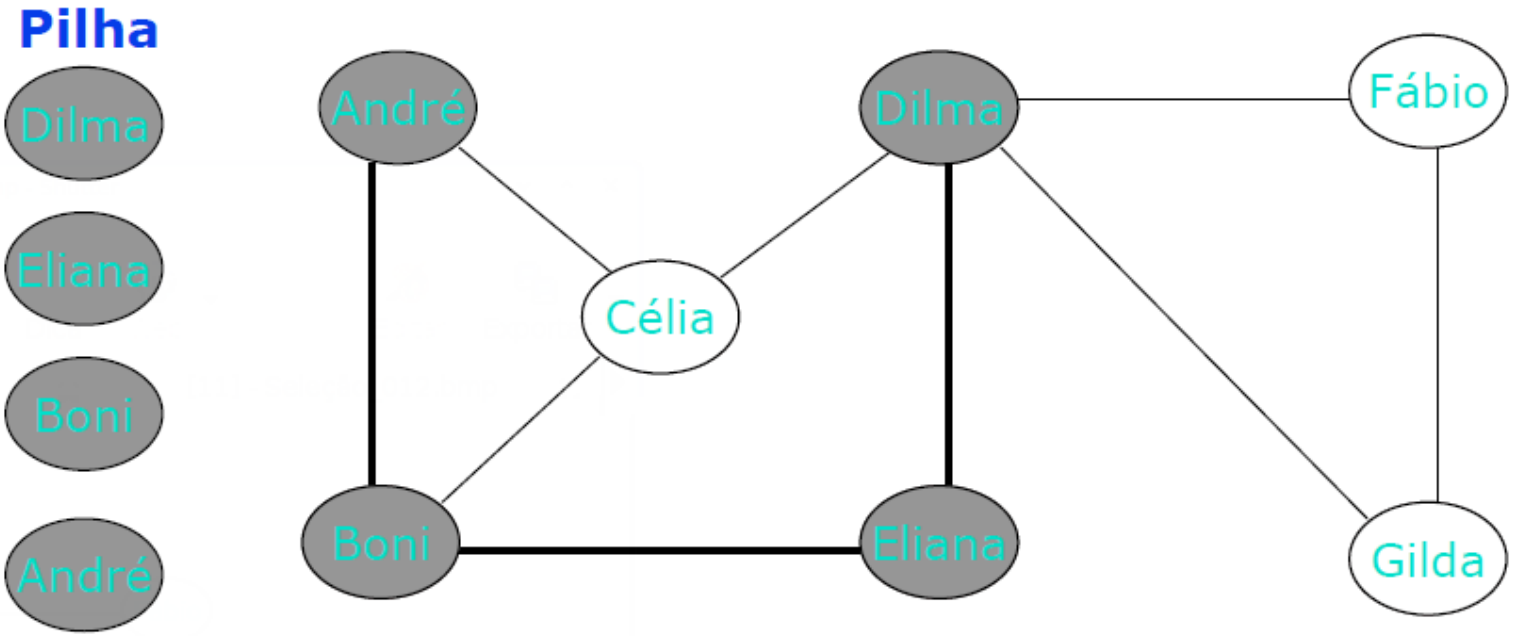
Busca em Profundidade



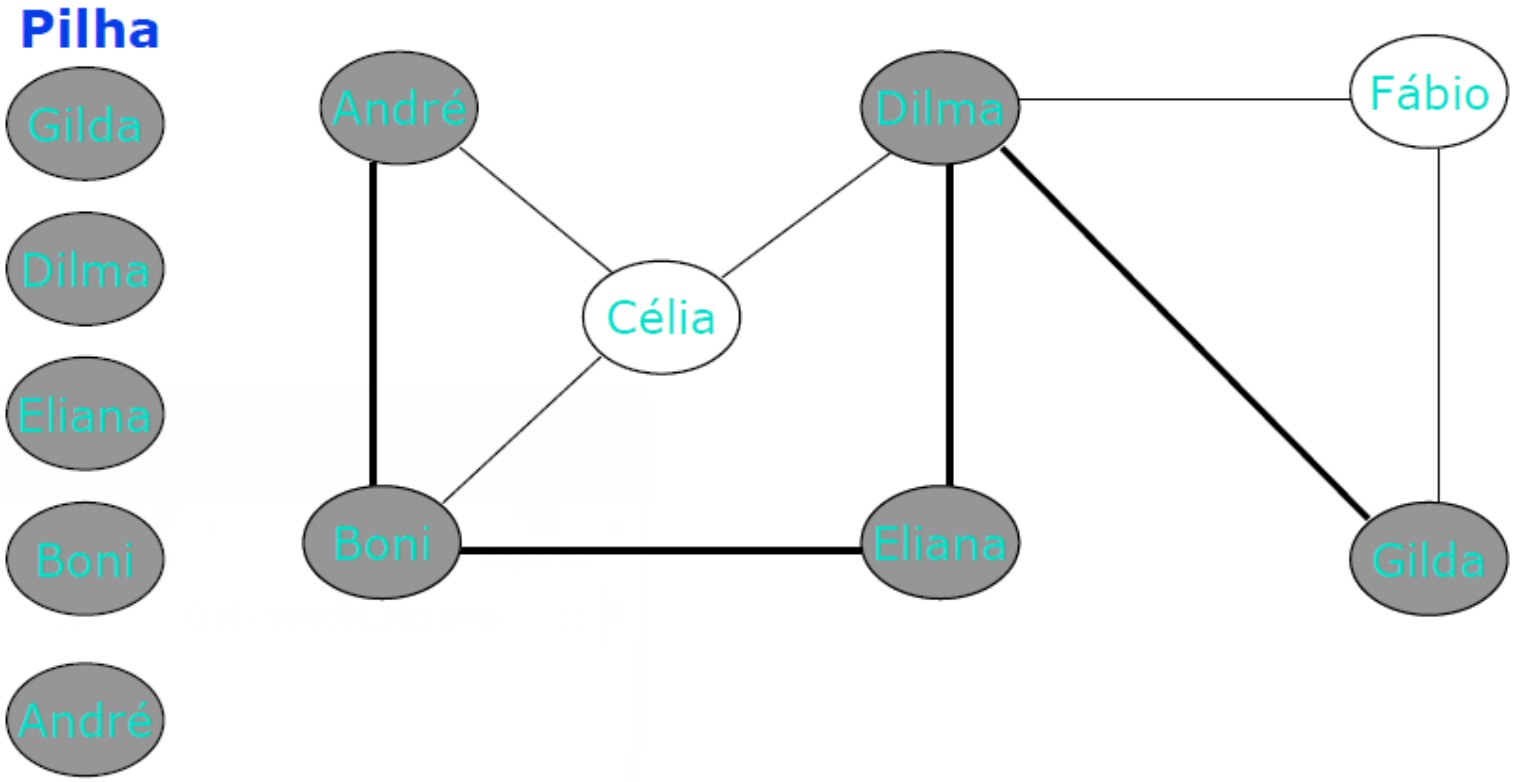
Busca em Profundidade



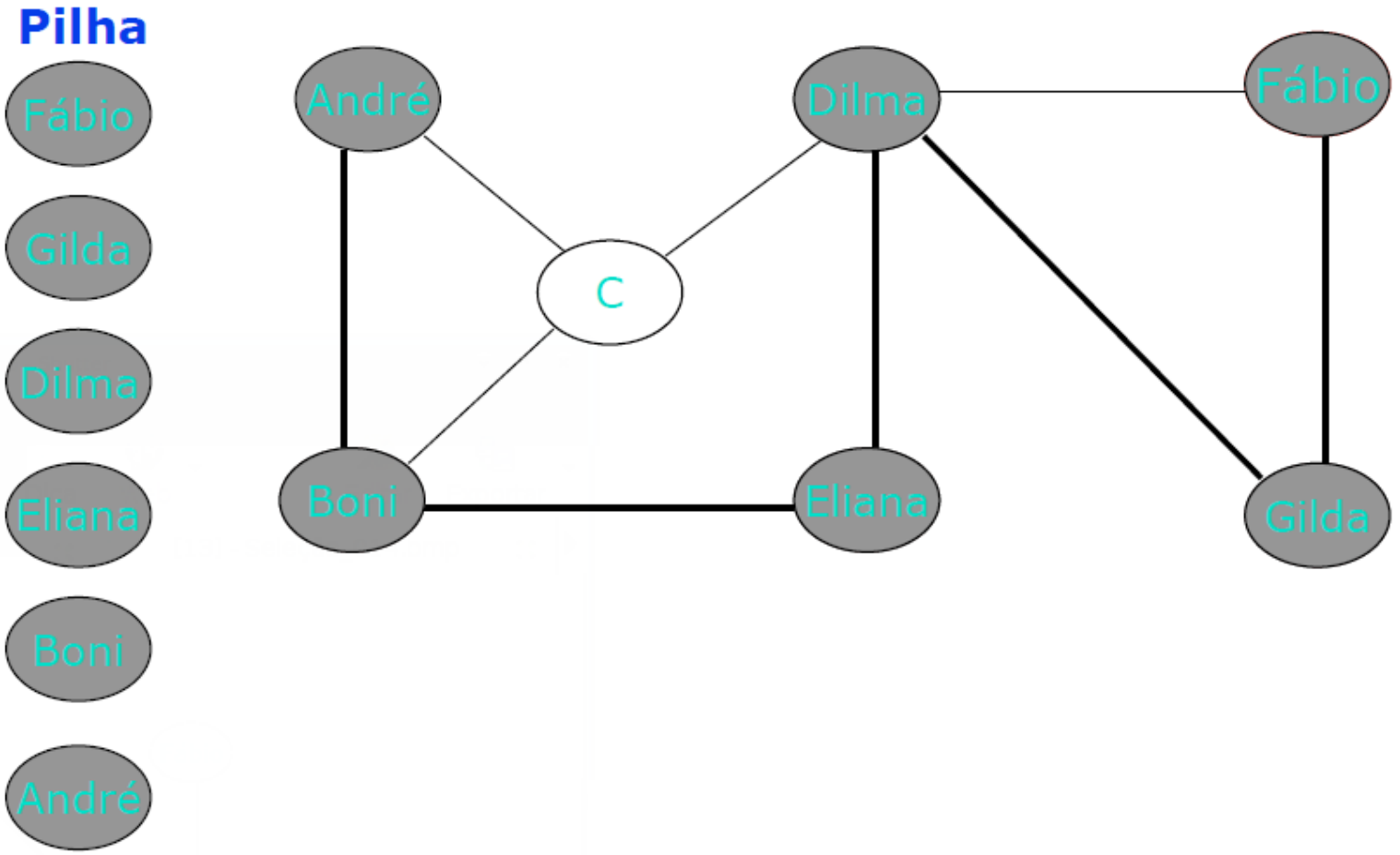
Busca em Profundidade



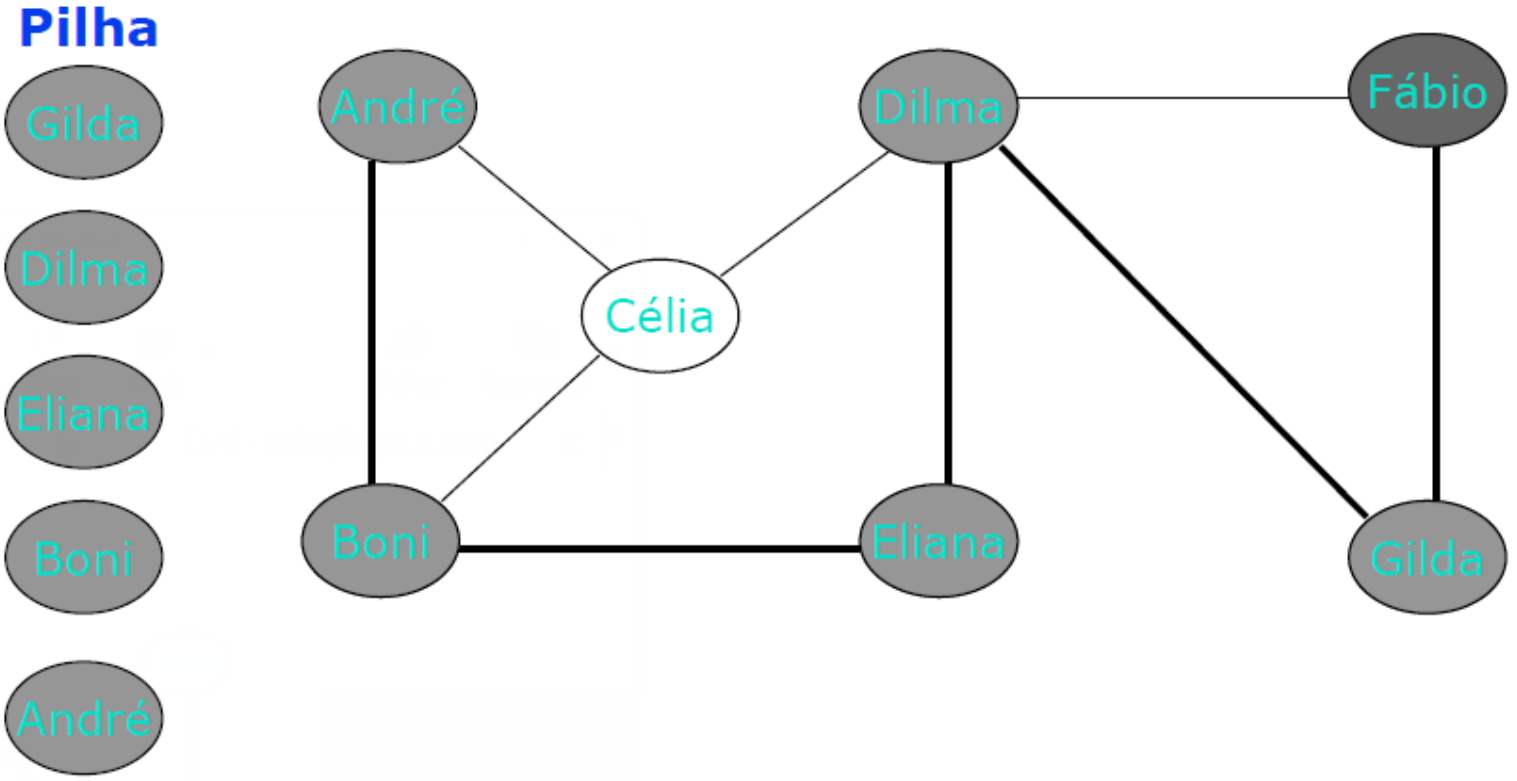
Busca em Profundidade



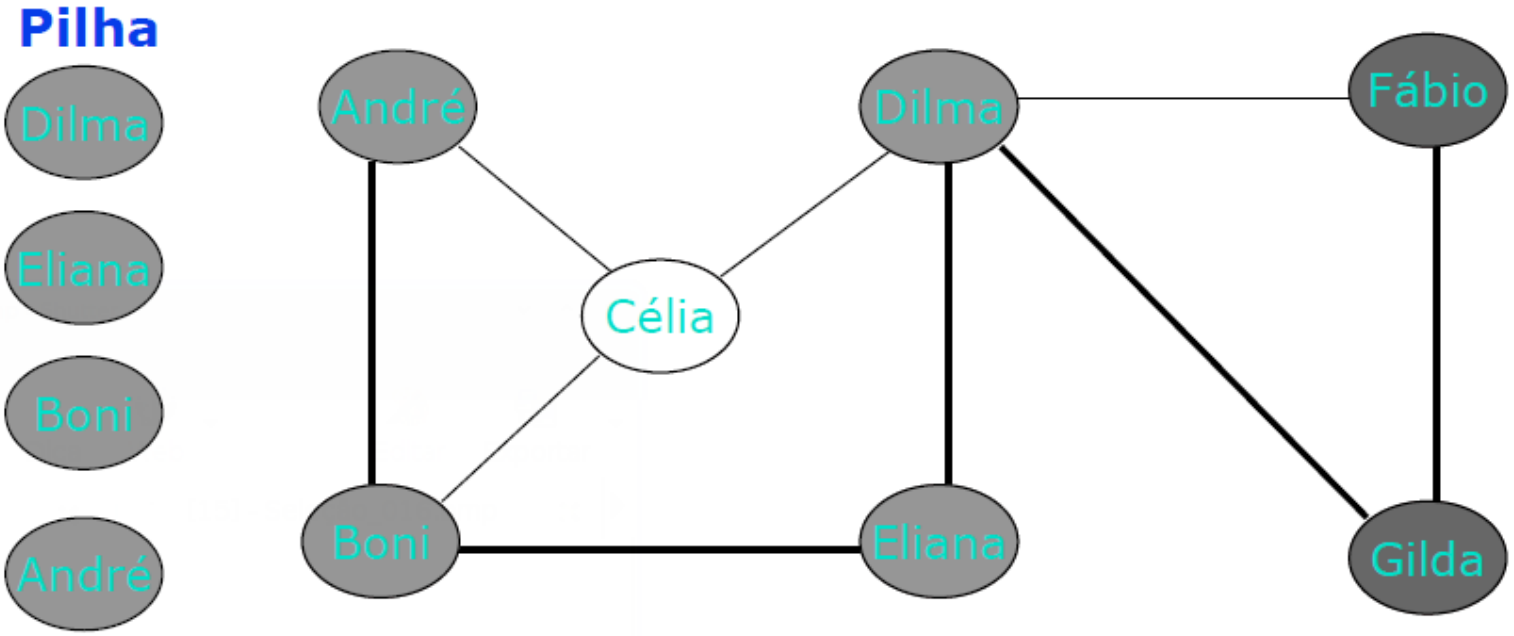
Busca em Profundidade



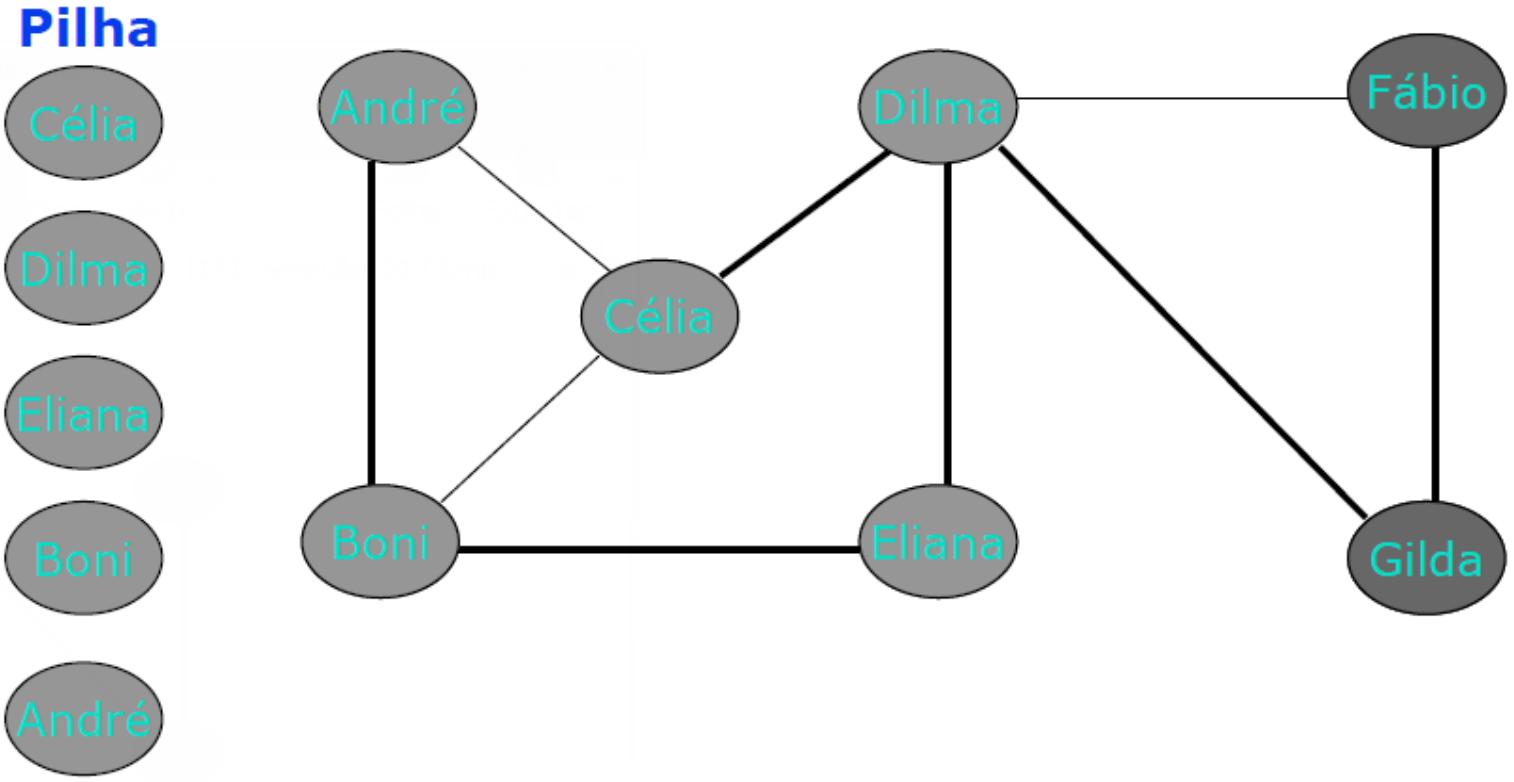
Busca em Profundidade



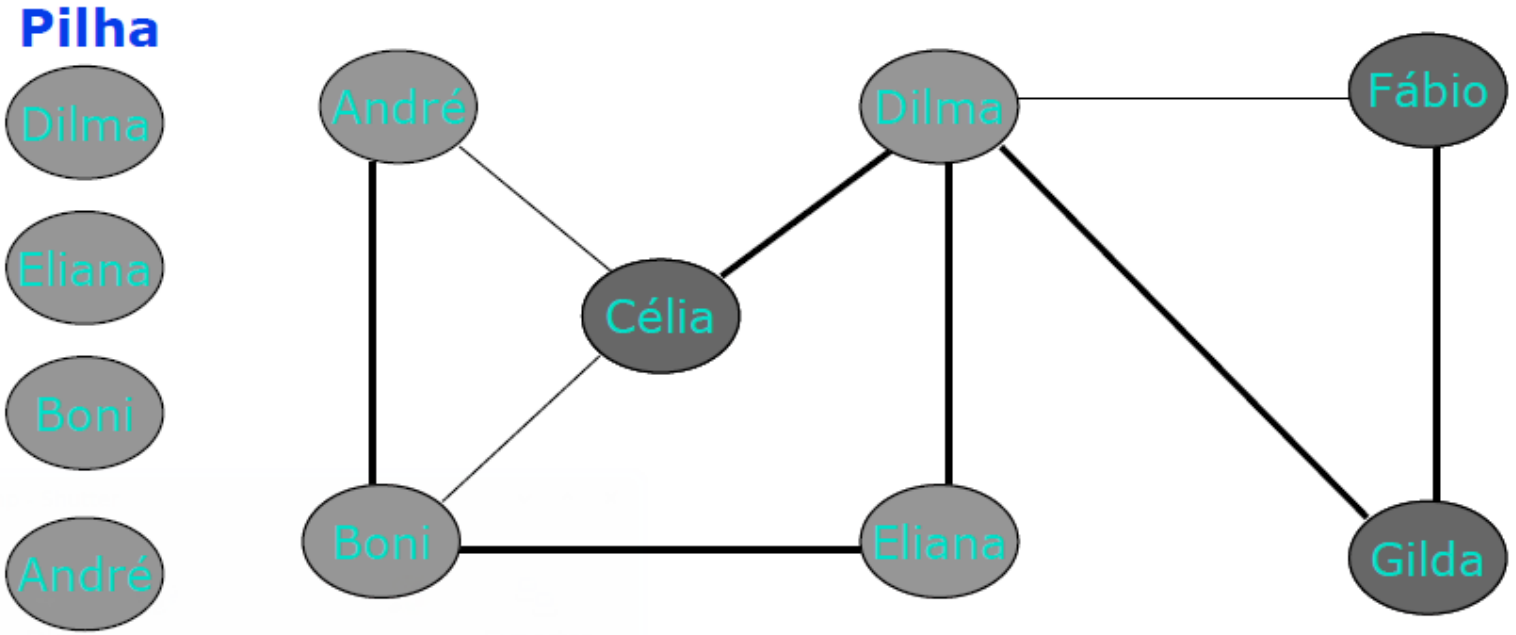
Busca em Profundidade



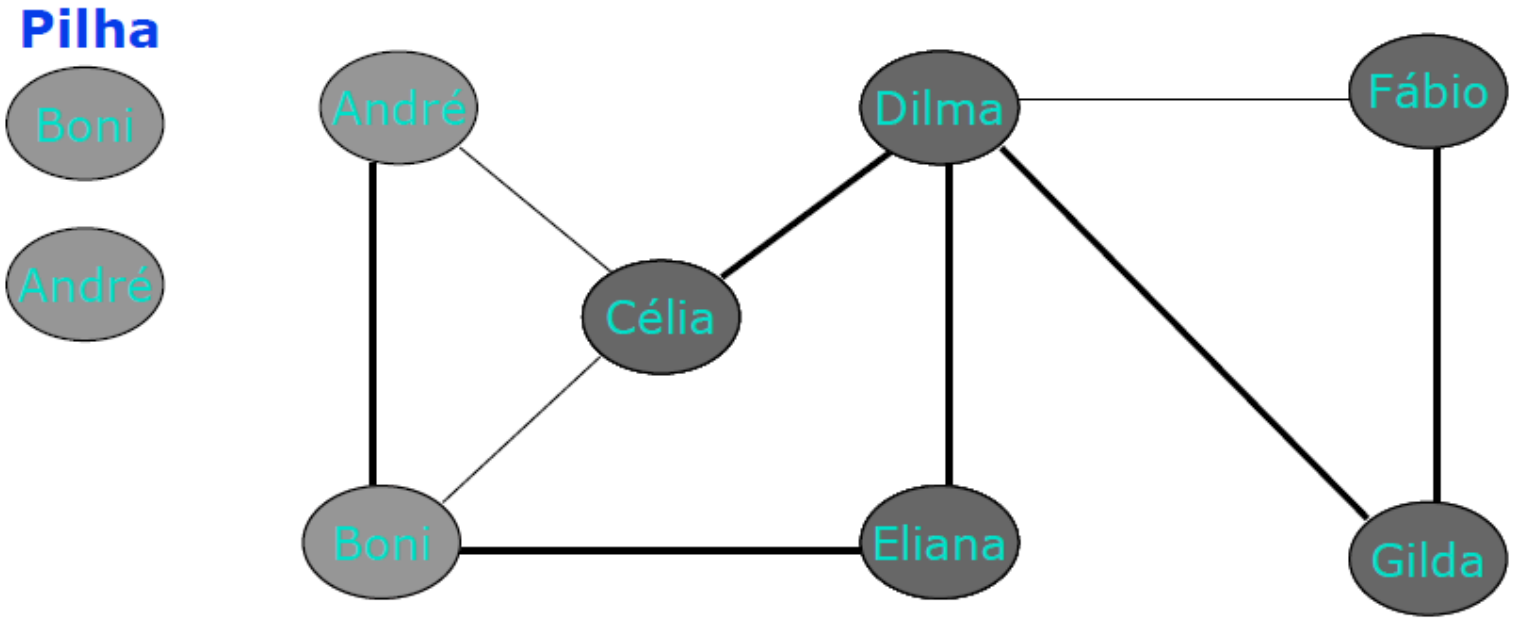
Busca em Profundidade



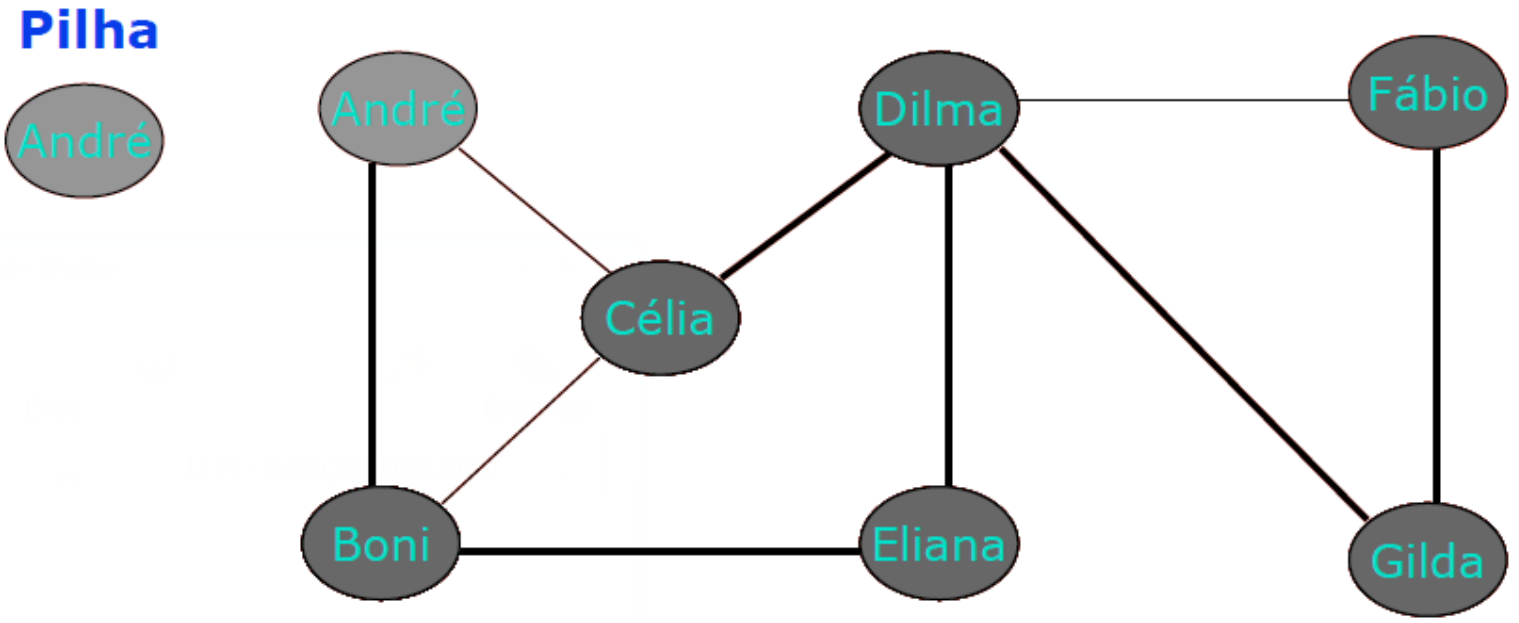
Busca em Profundidade



Busca em Profundidade

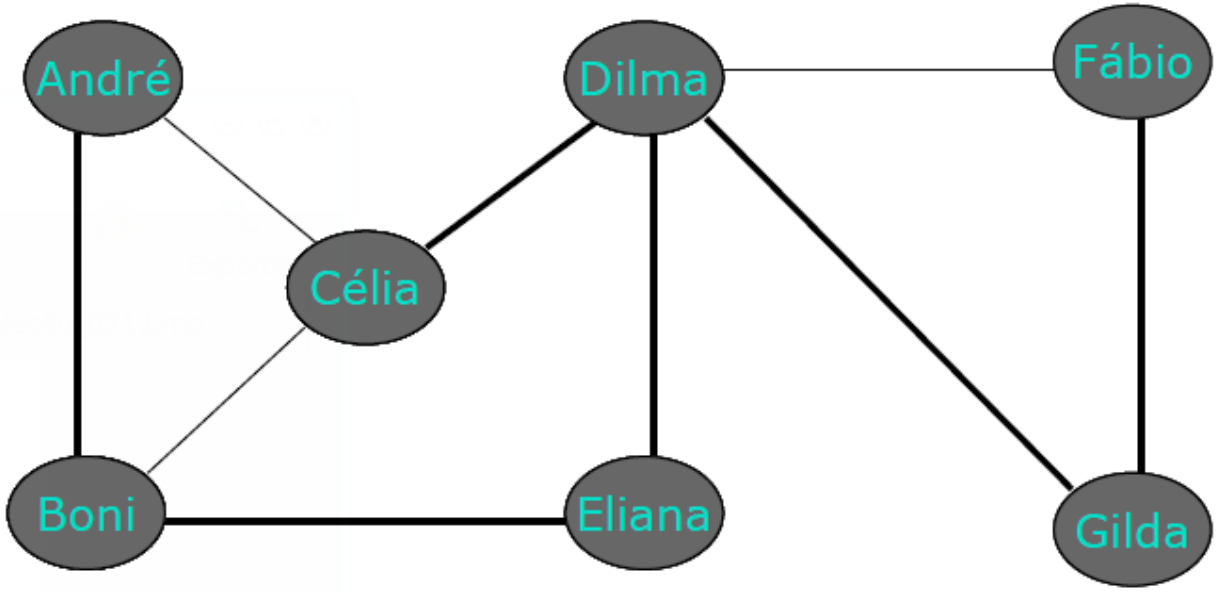


Busca em Profundidade



Busca em Profundidade

Pilha



Busca em Profundidade – Pseudocódigo

$DFS(G)$

1 *para cada vértice* $u \in V[G]$

2 $\quad cor[u] \leftarrow BRANCO$

3 $tempo \leftarrow 0$

4 *para cada vértice* $u \in V[G]$

5 $\quad$ *se* $cor[u] = BRANCO$

6 $\quad\quad DFS - VISIT(u)$

Busca em Profundidade – Pseudocódigo

$DFS-VISIT(u)$

1 $cor[u] \leftarrow CINZA$

2 $tempo \leftarrow tempo + 1$

3 $d[u] \leftarrow tempo$

4 *para cada vértice* $v \in Adj(u)$

5 *se* $cor[v] = BRANCO$

6 $DFS-VISIT(v)$

7 $cor[u] \leftarrow PRETO$

8 $f[u] \leftarrow tempo \leftarrow (tempo + 1)$

Busca em Profundidade

Complexidade do algoritmo:

Exercício:

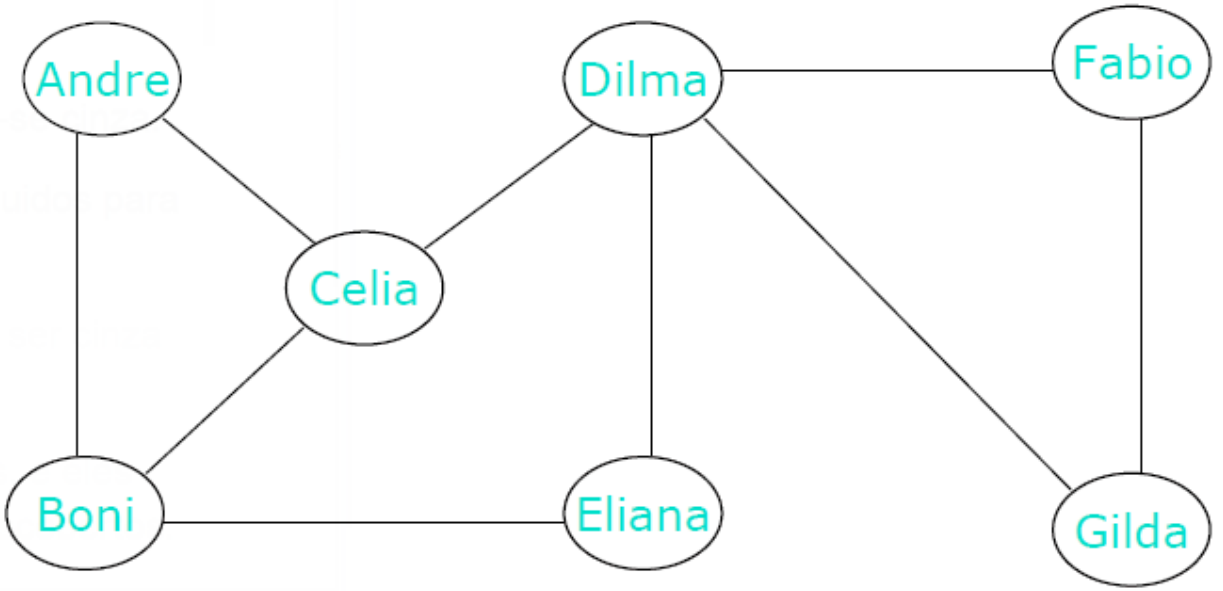
Busca em Largura

- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$.
- O grafo $G(V, A)$ pode ser direcionado ou não direcionado.

Busca em Largura

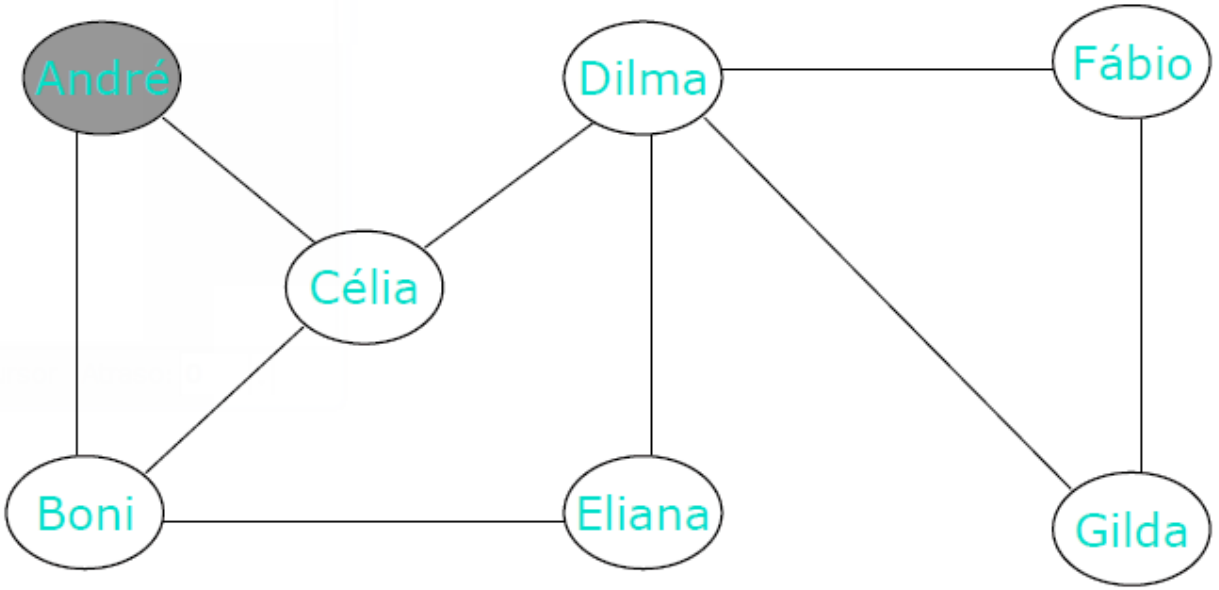
- Cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é descoberto pela primeira vez ele torna-se cinza.
- Vértices cinza e preto já foram descobertos, mas são distinguidos para assegurar que a busca ocorra em largura.
- Se $(u, v) \in A$ 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



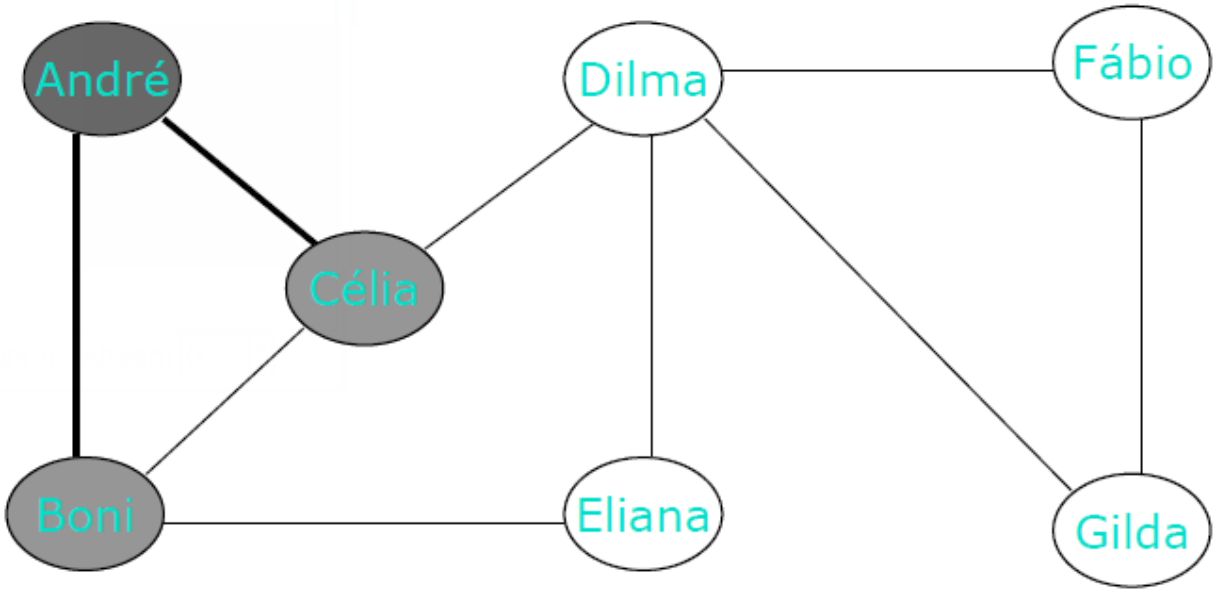
Fila:

Busca em Largura



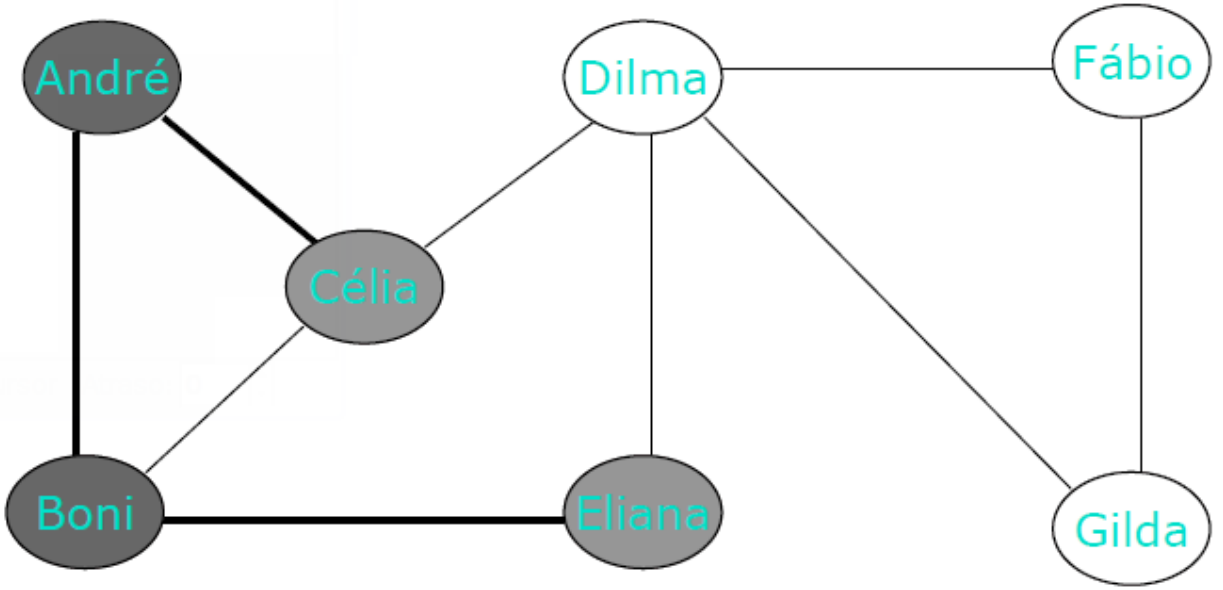
Fila:

Busca em Largura



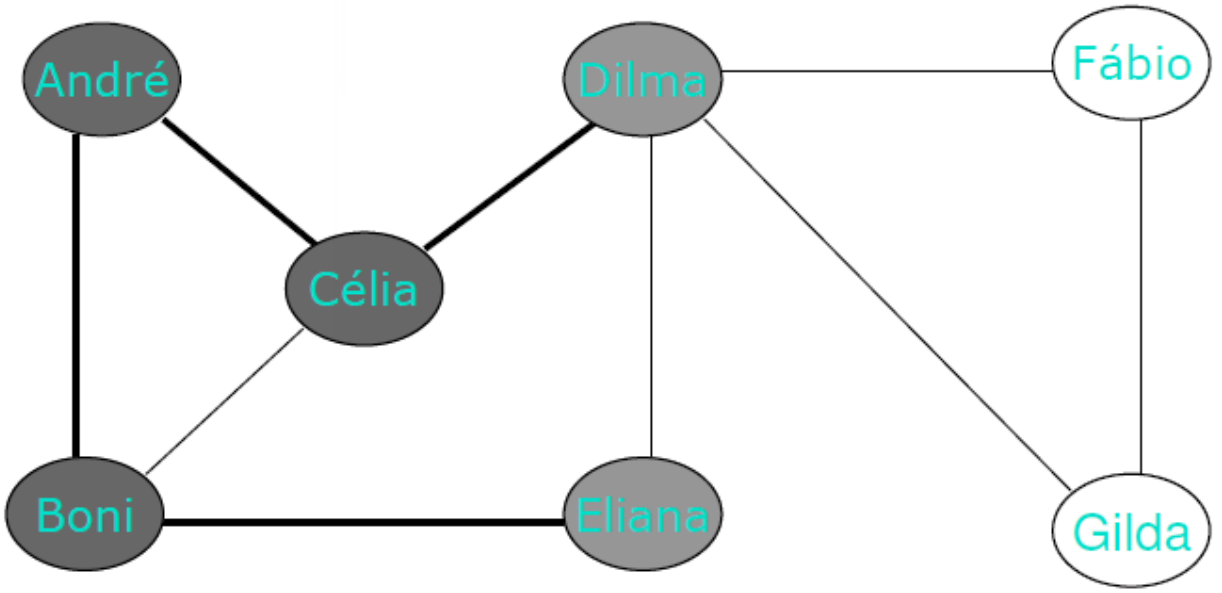
Fila:

Busca em Largura



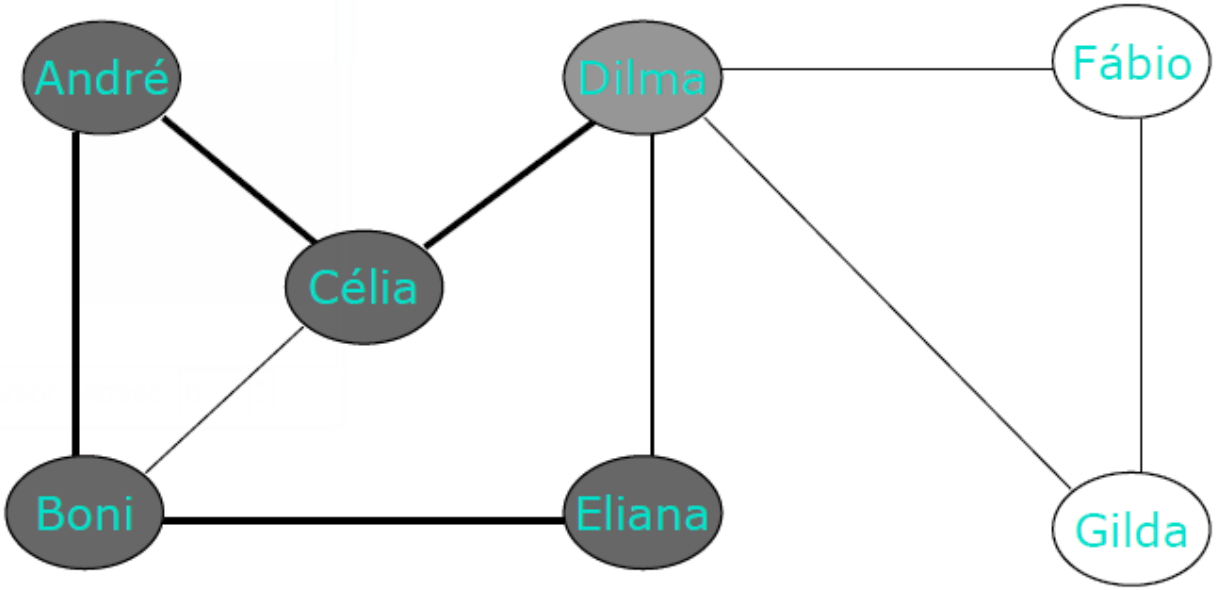
Fila:

Busca em Largura



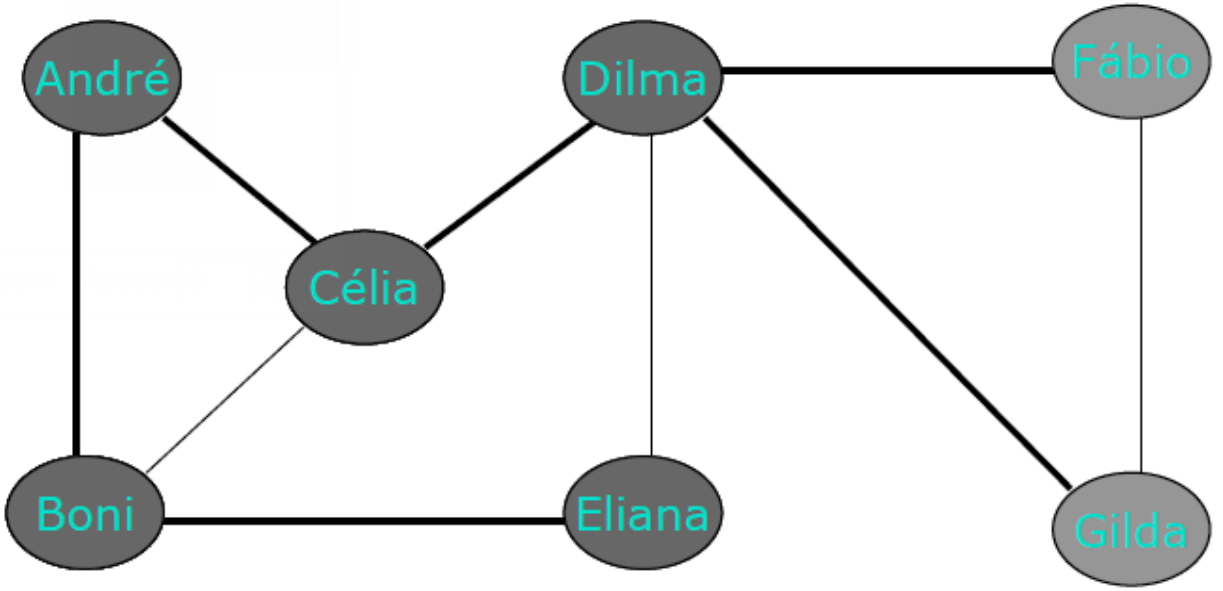
Fila:

Busca em Largura



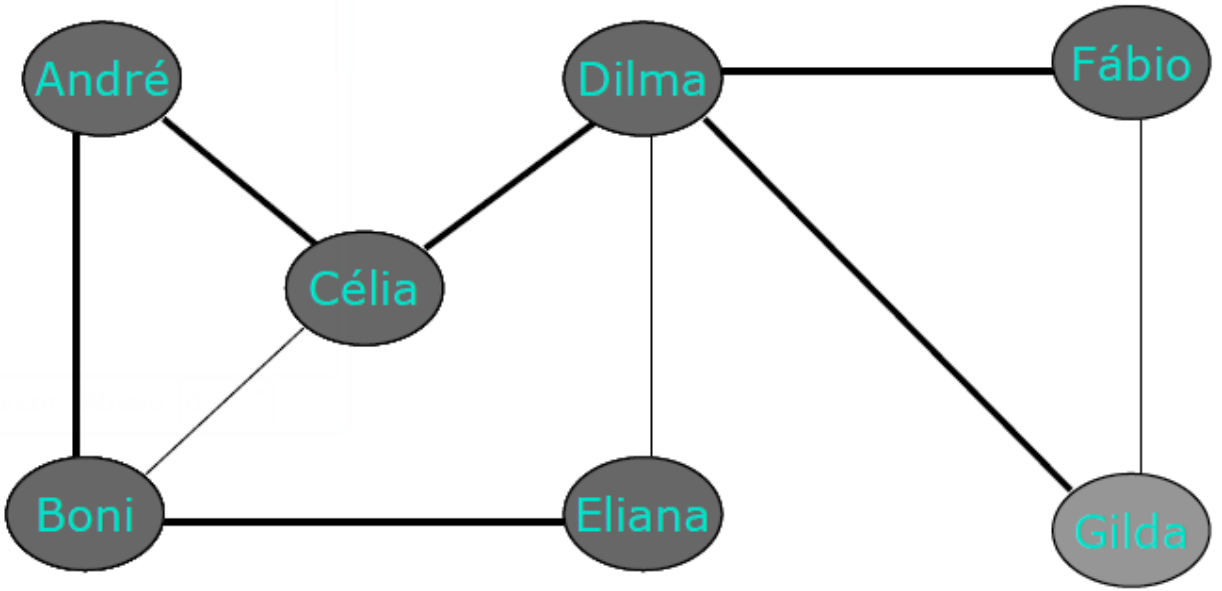
Fila:

Busca em Largura



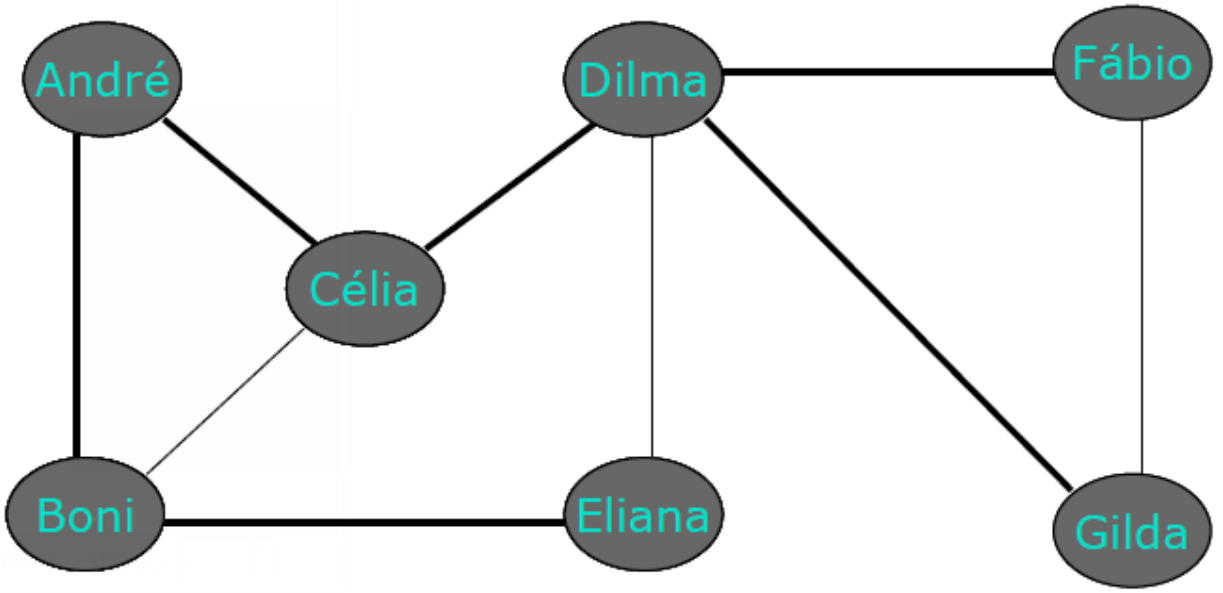
Fila:

Busca em Largura



Fila:

Busca em Largura



Fila:

Busca em Largura - Pseudocódigo

```
BFS(G, s)
1 para cada vértice u ∈ V[G] − {s}
2     cor[u] ← BRANCO
3     d[u] ← ∞
4     π[u] ← NULL
5 cor[s] ← CINZA
6 d[s] ← 0
7 π[s] ← NULL
8 Q ← novaFila()
9 ENFILEIRA(Q, s)
```

Legenda:

cor[u]; *//indicativo de atingibilidade;*
π[u]; *//indica o vértice predecessor de u(pai);*
d[u]; *//indica a distância desde a origem d(s,u) - em arestas;*
Q; *//indica a fila (FIFO) – ponto chave do algoritmo.*

Busca em Largura - Pseudocódigo

```
10 enquanto !vazia(Q)
11     u ← DESENFILEIRA(Q)
12     para cada v ∈ Adj[u]
13         se cor[v] = BRANCO
14             cor[v] ← CINZA
15             d[v] = d[u]+1
16             π[v] ← u
17             ENFILEIRA(Q,v)
18     cor[u] ← PRETO
```

Busca em Largura

Complexidade do algoritmo:

Exercício: