Grafos e Árvores

Travessia e Diâmetro

Prof. Edson Alves

2018

Faculdade UnB Gama

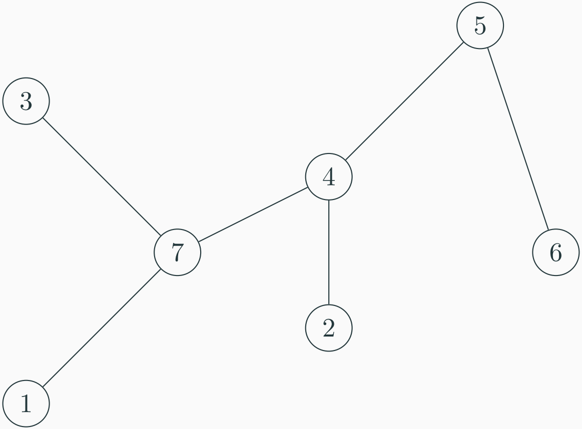
Sumário

Fundamentos

Definição de árvore

- Uma árvore é um grafo não direcionado, conectado e acíclico com N vértices e $N - 1$ arestas
- A remoção de qualquer uma das arestas divide a árvore em dois componentes
- A adição de uma aresta cria um ciclo, descaracterizando a árvore
- Para quaisquer vértices u e v da árvore existe um caminho único de u a v

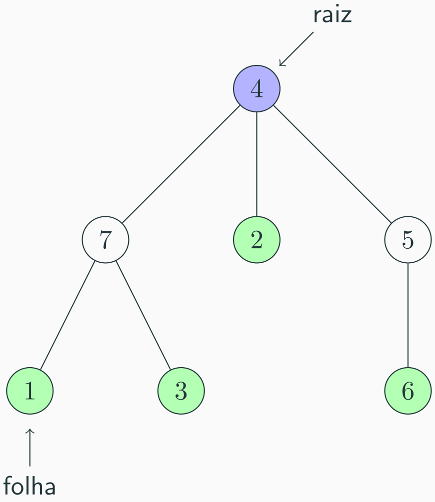
Visualização de uma árvore



Folhas e raiz

- Uma folha é um nó com apenas um vizinho (no exemplo anterior, os nós com rótulos 1, 2, 3 e 6 são folhas da árvore)
- Em uma árvore com raiz, um dos nós é escolhido para ser a raiz da árvore
- Os demais nós são organizados em níveis, de acordo com sua distância até à raiz
- Esta organização é implícita: não é preciso alterar a estrutura da árvore (no máximo, deixar indicada qual é a raiz)
- Em uma árvore com raiz, os filhos são os vizinhos que estão no nível inferior em relação ao nó
- Cada nó tem um único pai, o qual é o nó que está no nível imediatamente acima
- A estrutura de uma árvore é recursiva: cada nó pode ser interpretado como raiz de uma subárvore

Visualização de uma árvore com raiz



Travessias em árvores

DFS em árvores

- Embora a DFS e a BFS possa ser utilizadas em árvores sem nenhuma alteração, a estrutura simplificada da árvore permite implementações mais simples e com menor complexidade de memória
- Por conta da ausência de ciclos, a implementação da DFS em árvores dispensa o vetor que mantém o registro dos vértices já visitados
- Ele pode ser substituído por um parâmetro extra, que mantém o registro do nó p que antecede u na travessia
- Assim, a complexidade de memória reduz de $O(V)$ para $O(1)$
- Na primeira chamada da DFS, o parâmetro p deve ser igual a zero (ou qualquer valor sentinela que não seja um rótulo de um dos vértices do grafo)

Implementação da DFS em árvores

```
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;

const int MAX { 1010 };
vector<int> adj[MAX];

void dfs(int u, int p)
{
    cout << u << " ";

    for (const auto& v : adj[u])
        if (v != p)
            dfs(v, u);
}

```

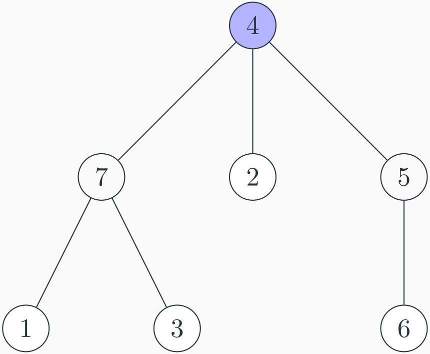
Implementação da DFS em árvores

```cpp
int main()
{
    vector<ii> edges { ii(1, 7), ii(3, 7), ii(7, 4), ii(4, 2),
        ii(4, 5), ii(5, 6) };

    for (const auto& [u, v] : edges) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    // 4 7 1 3 2 5 6
    dfs(4, 0);
    cout << endl;

    return 0;
}
```

Números de nós na subárvore

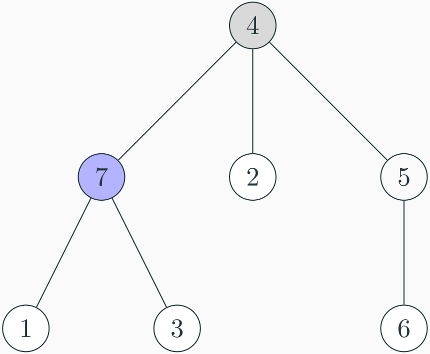
- A DFS, em conjunto com técnicas de programação dinâmica, permite computar em $O(N)$ algumas características da árvore
- Um primeiro exemplo seria o número de nós `nodes[u]` da subárvore cuja raiz é o nó u
- Se u é uma folha, então `nodes[u]` $= 1$ (apenas u faz parte da subárvore)
- Caso contrário, `nodes[u]` $= 1 + \sum_v$ `nodes[v]`, onde v é um filho de u

Visualização do algoritmo que computa o número de nós



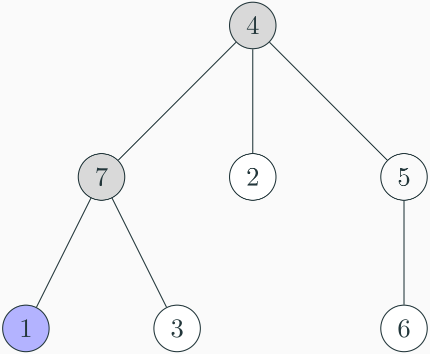
	1	2	3	4	5	6	7
nodes =	0	0	0	1	0	0	0

Visualização do algoritmo que computa o número de nós



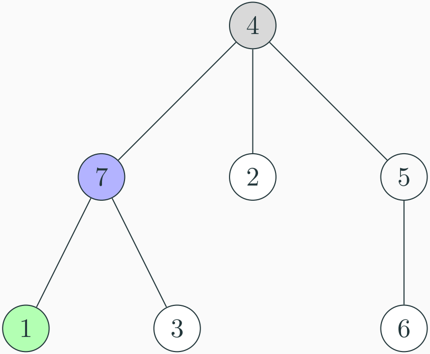
	1	2	3	4	5	6	7
nodes =	0	0	0	1	0	0	1

Visualização do algoritmo que computa o número de nós



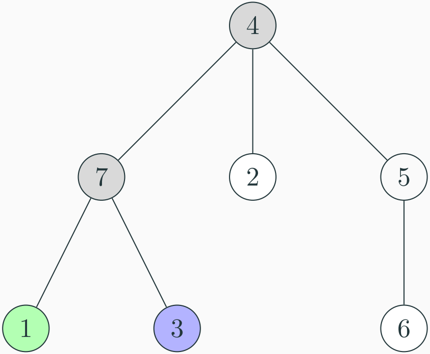
	1	2	3	4	5	6	7
nodes =	1	0	0	1	0	0	1

Visualização do algoritmo que computa o número de nós



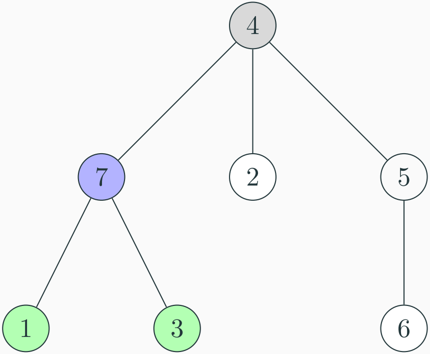
	1	2	3	4	5	6	7
nodes =	1	0	0	1	0	0	2

Visualização do algoritmo que computa o número de nós



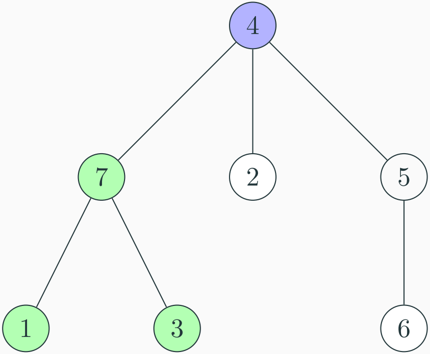
	1	2	3	4	5	6	7
nodes =	1	0	1	1	0	0	2

Visualização do algoritmo que computa o número de nós



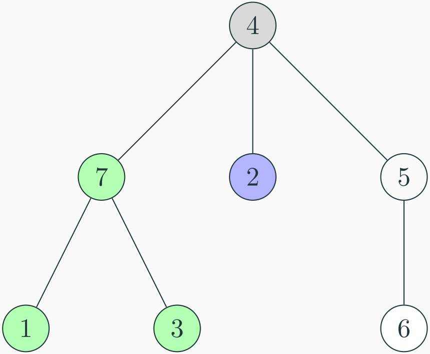
	1	2	3	4	5	6	7
nodes =	1	0	1	1	0	0	3

Visualização do algoritmo que computa o número de nós



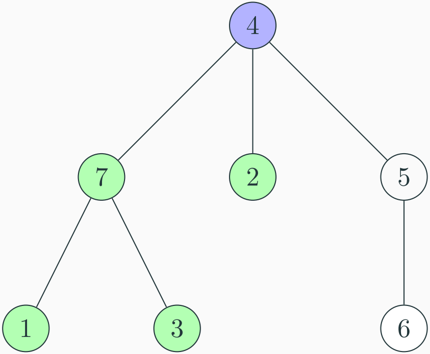
	1	2	3	4	5	6	7
nodes =	1	0	1	4	0	0	3

Visualização do algoritmo que computa o número de nós



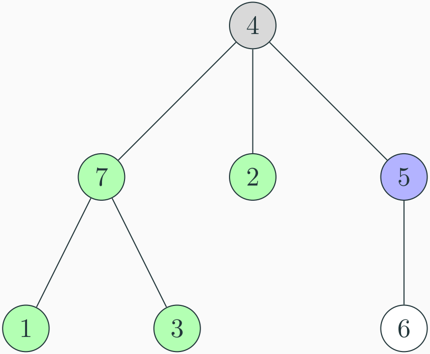
	1	2	3	4	5	6	7
nodes =	1	1	1	4	0	0	3

Visualização do algoritmo que computa o número de nós



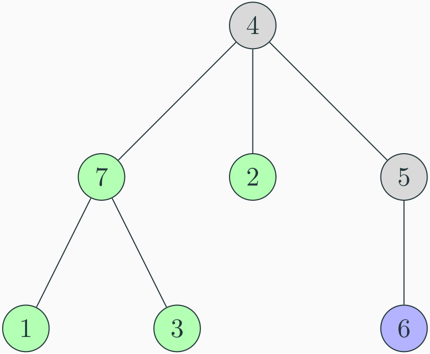
	1	2	3	4	5	6	7
nodes =	1	1	1	5	0	0	3

Visualização do algoritmo que computa o número de nós



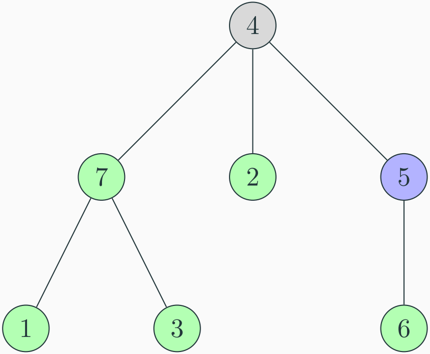
	1	2	3	4	5	6	7
nodes =	1	1	1	5	1	0	3

Visualização do algoritmo que computa o número de nós



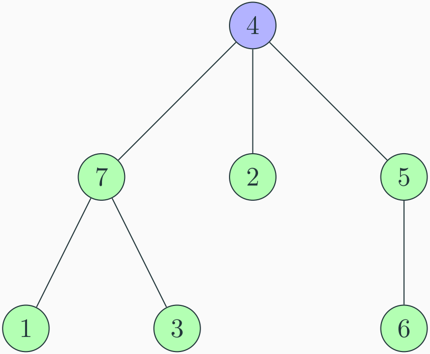
	1	2	3	4	5	6	7
nodes =	1	1	1	5	1	1	3

Visualização do algoritmo que computa o número de nós



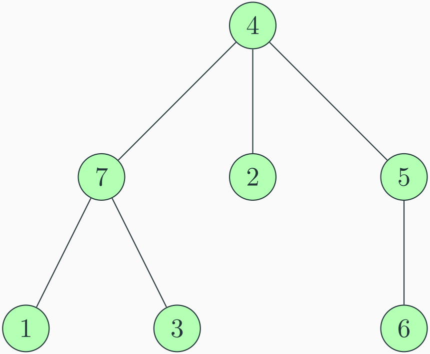
	1	2	3	4	5	6	7
nodes =	1	1	1	5	2	1	3

Visualização do algoritmo que computa o número de nós



	1	2	3	4	5	6	7
nodes =	1	1	1	7	2	1	3

Visualização do algoritmo que computa o número de nós



	1	2	3	4	5	6	7
nodes =	1	1	1	7	2	1	3

Implementação da rotina que computa nodes[u]

```cpp
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;

const int MAX { 1010 };
vector<int> adj[MAX];
int nodes[MAX];

void dfs(int u, int p)
{
    nodes[u] = 1;

    for (const auto& v : adj[u])
    {
        if (v == p) continue;

        dfs(v, u);
        nodes[u] += nodes[v];
    }
}
```

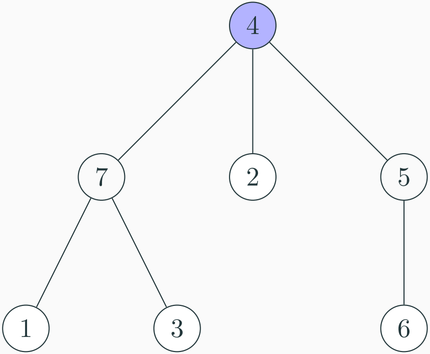
Implementação da rotina que computa nodes[u]

```cpp

int main()
{
    vector<ii> edges { ii(1, 7), ii(3, 7), ii(7, 4), ii(4, 2),
        ii(4, 5), ii(5, 6) };

    for (const auto& [u, v] : edges) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    dfs(4, 0);

    // 1 1 1 7 2 1 3
    for (int u = 1; u <= 7; ++u)
        cout << nodes[u] << (u == 7 ? '\n': ' ');

    return 0;
}
```

Maior caminho até uma folha

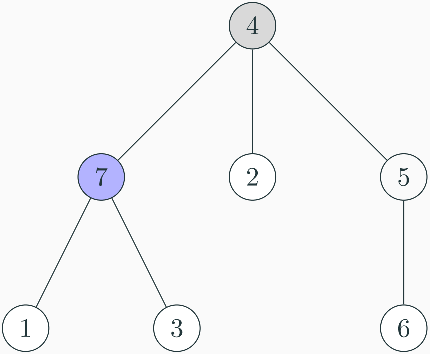
- Outro exemplo de DFS com DP é o cálculo do tamanho (em número de arestas) do maior caminho `to_leaf[u]` de u até uma folha
- Se u for uma folha então $\texttt{to\_leaf[u]} = 0$
- Caso contrário, $\texttt{to\_leaf[u]} = 1 + \max\{\ \texttt{to\_leaf}[v_i]\ \}$, onde v_i são os filhos de u
- Esta rotina pode ser facilmente adaptada para retornar o tamanho como a soma dos pesos das arestas

Visualização do algoritmo que computa to_leaf



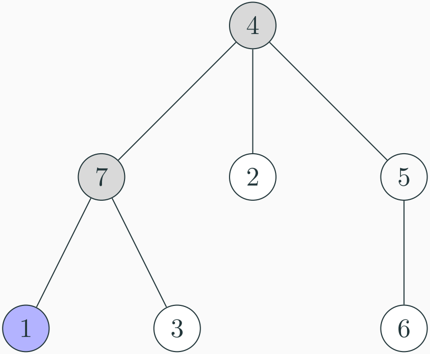
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	0

Visualização do algoritmo que computa to_leaf



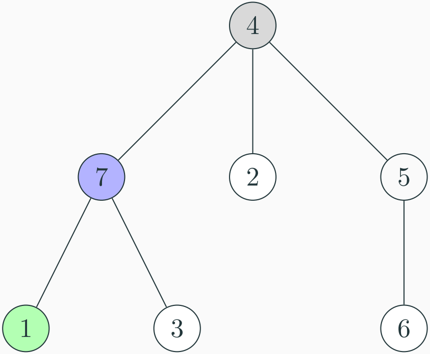
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	0

Visualização do algoritmo que computa to_leaf



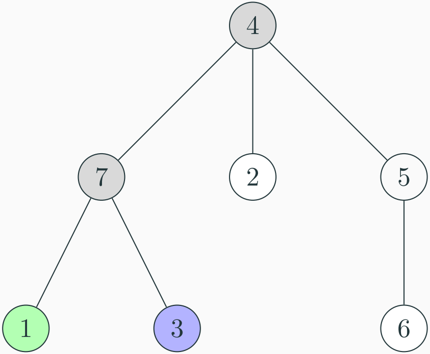
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	0

Visualização do algoritmo que computa to_leaf



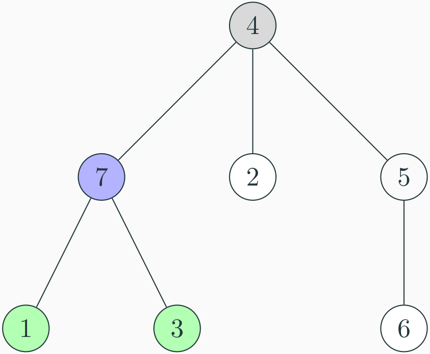
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	0

Visualização do algoritmo que computa to_leaf



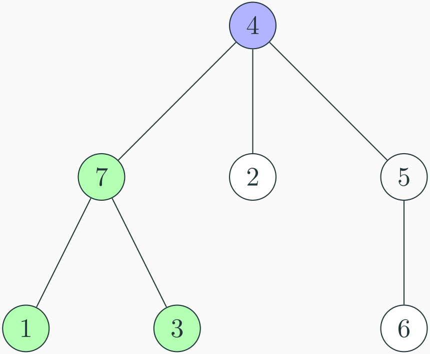
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	0

Visualização do algoritmo que computa to_leaf



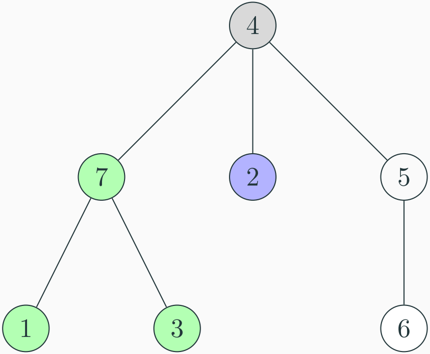
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1

Visualização do algoritmo que computa to_leaf



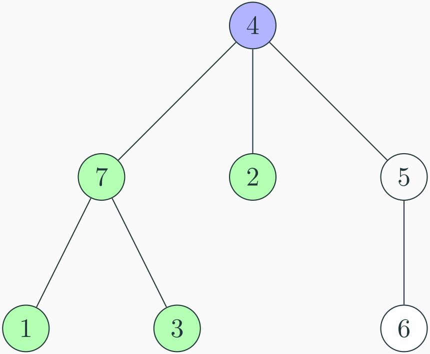
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1

Visualização do algoritmo que computa to_leaf



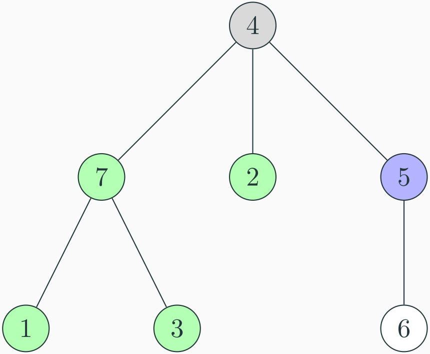
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1

Visualização do algoritmo que computa to_leaf



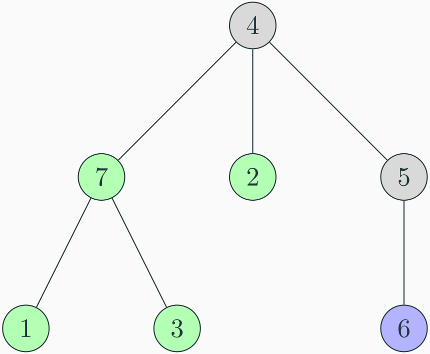
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1

Visualização do algoritmo que computa to_leaf



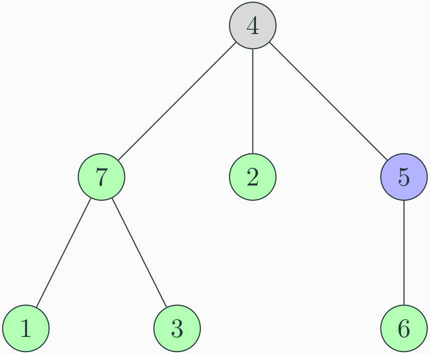
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1

Visualização do algoritmo que computa to_leaf



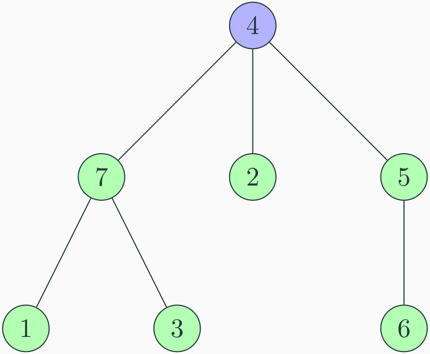
	1	2	3	4	5	6	7
`to_leaf =`	0	0	0	0	0	0	1

Visualização do algoritmo que computa to_leaf



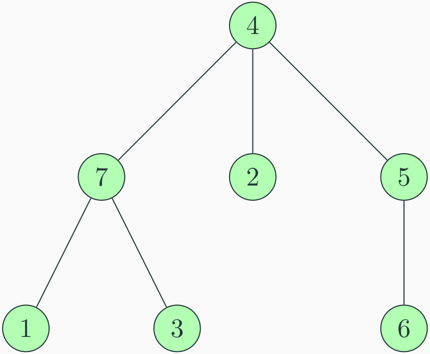
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	1	0	1

Visualização do algoritmo que computa `to_leaf`



	1	2	3	4	5	6	7
`to_leaf =`	0	0	0	2	1	0	1

Visualização do algoritmo que computa `to_leaf`



	1	2	3	4	5	6	7
`to_leaf =`	0	0	0	2	1	0	1

Implementação da rotina que computa to_leaf[u]

```
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;

const int MAX { 1010 };
vector<int> adj[MAX];
int to_leaf[MAX];

void dfs(int u, int p) {
    int m = -1;

    for (const auto& v : adj[u]) {
        if (v == p) continue;

        dfs(v, u);
        m = max(m, to_leaf[v]);
    }

    to_leaf[u] = 1 + m;
}
```

Implementação da rotina que computa to_leaf[u]

```cpp

int main()
{
    vector<ii> edges { ii(1, 7), ii(3, 7), ii(7, 4), ii(4, 2),
        ii(4, 5), ii(5, 6) };

    for (const auto& [u, v] : edges) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    dfs(4, 0);

    // 0 0 0 2 1 0 1
    for (int u = 1; u <= 7; ++u)
        cout << to_leaf[u] << (u == 7 ? '\n': ' ');

    return 0;
}
```

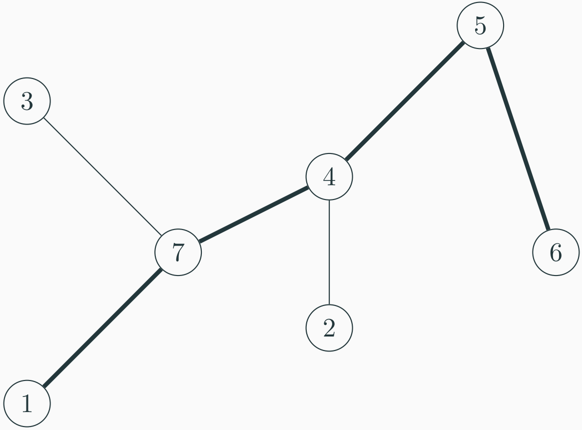
Diâmetro

Diâmetro de uma árvore

- O diâmetro de uma árvore é igual ao maior dentre todos os tamanhos dos caminhos entre os pares de vértices u e v do grafo
- O maior caminho que produz o diâmetro não é, necessariamente, único
- Computar estas distâncias utilizando o algoritmo de Floyd-Warshall em $O(V^3)$ e, em seguida, determinar a maior dentre estas distâncias em $O(V^2)$ produziria o resultado correto
- Porém é possível chegar ao mesmo resultado de duas maneiras: com programação dinâmica ou com duas DFS
- Em ambos casos, a complexidade é $O(V)$

Visualização do diâmetro de uma árvore

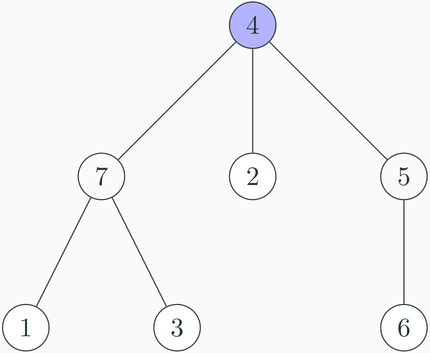
Diâmetro: 4



Diâmetro com programação dinâmica

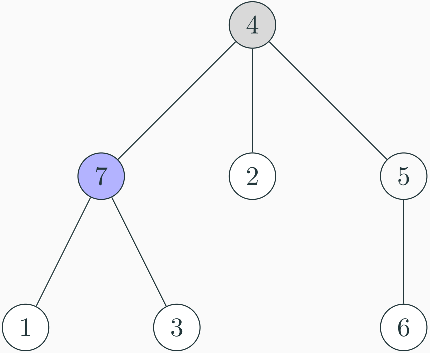
- Para computar o diâmetro com programação dinâmica é preciso observar que, em uma árvore com raiz, cada caminho possui um pico: o nó que está posicionado no nível mais alto da árvore
- Assim, para cada nó u da árvore, será computado `max_length[u]`: o tamanho do maior caminho que tem u como pico
- Dentre todos estes caminhos, um deles fornecerá o diâmetro
- Para computar `max_length[u]`, basta recorrer à rotina que computa `to_leaf[v]` para todos os filhos v de u
- Se u não tem filhos, `max_length[u]` $= 0$
- Se u tem apenas um filho, `max_length[u]` $=$ `to_leaf[v]` $+ 1$
- Se u tem dois ou mais filhos, `max_length[u]` $=$ `to_leaf[v]` $+$ `to_leaf[w]` $+ 2$, onde v e w são dois filhos distintos com os maiores valores `to_leaf` dentre todos os filhos de u

Visualização do algoritmo que computa o diâmetro com DP



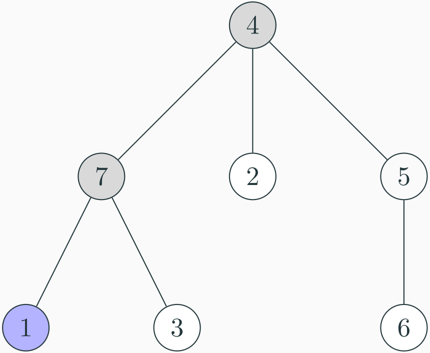
	1	2	3	4	5	6	7
`to_leaf =`	0	0	0	0	0	0	0
`max_length =`	0	0	0	0	0	0	0

Visualização do algoritmo que computa o diâmetro com DP



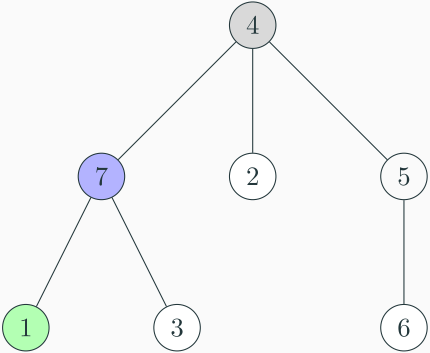
	1	2	3	4	5	6	7
`to_leaf =`	0	0	0	0	0	0	0
`max_length =`	0	0	0	0	0	0	0

Visualização do algoritmo que computa o diâmetro com DP



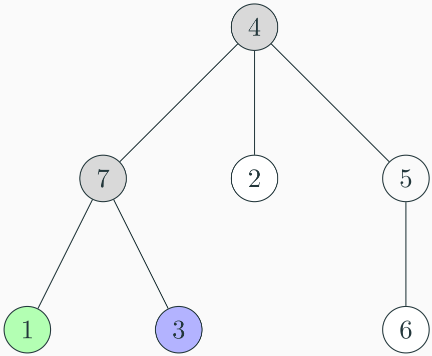
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	0
max_length =	0	0	0	0	0	0	0

Visualização do algoritmo que computa o diâmetro com DP



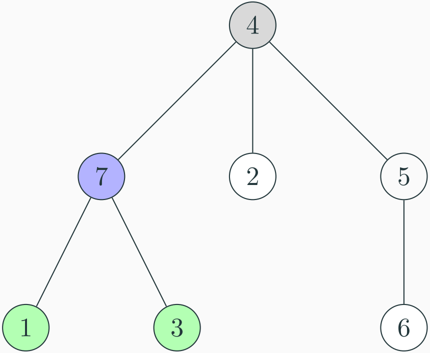
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	0
max_length =	0	0	0	0	0	0	0

Visualização do algoritmo que computa o diâmetro com DP



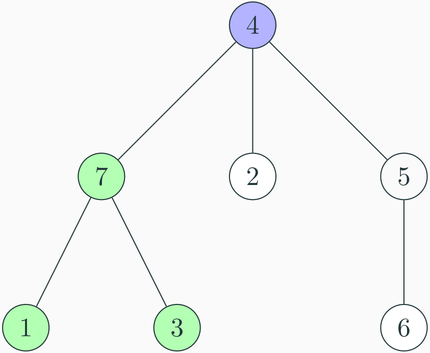
	1	2	3	4	5	6	7
`to_leaf =`	0	0	0	0	0	0	0
`max_length =`	0	0	0	0	0	0	0

Visualização do algoritmo que computa o diâmetro com DP



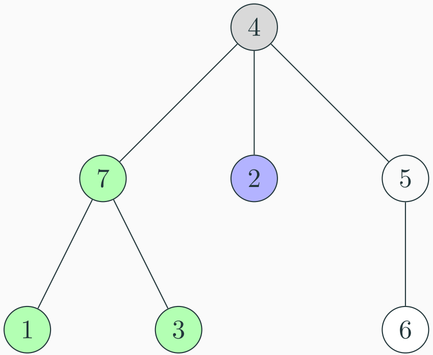
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1
max_length =	0	0	0	0	0	0	2

Visualização do algoritmo que computa o diâmetro com DP



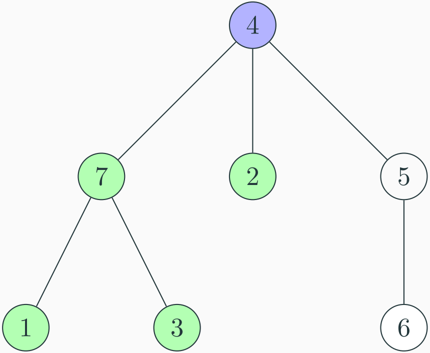
	1	2	3	4	5	6	7
`to_leaf =`	0	0	0	0	0	0	1
`max_length =`	0	0	0	0	0	0	2

Visualização do algoritmo que computa o diâmetro com DP



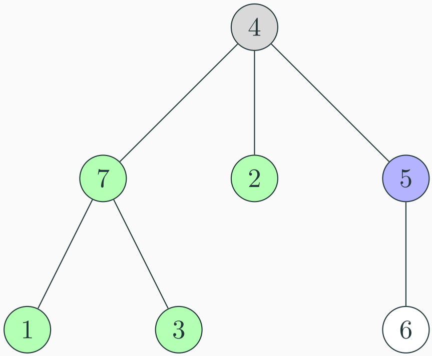
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1
max_length =	0	0	0	0	0	0	2

Visualização do algoritmo que computa o diâmetro com DP



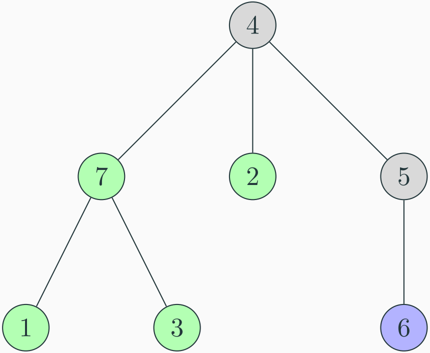
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1
max_length =	0	0	0	0	0	0	2

Visualização do algoritmo que computa o diâmetro com DP



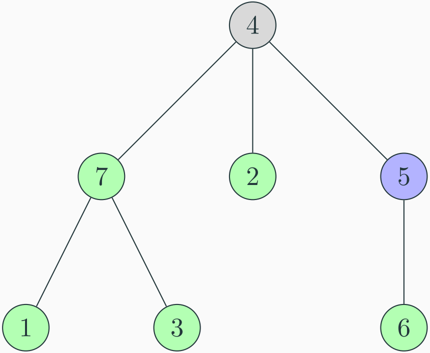
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1
max_length =	0	0	0	0	0	0	2

Visualização do algoritmo que computa o diâmetro com DP



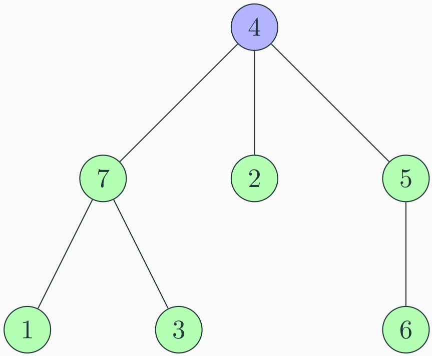
	1	2	3	4	5	6	7
to_leaf =	0	0	0	0	0	0	1
max_length =	0	0	0	0	0	0	2

Visualização do algoritmo que computa o diâmetro com DP



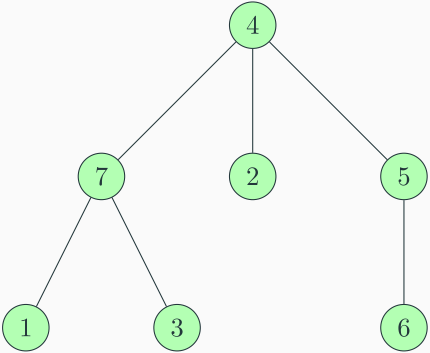
	1	2	3	4	5	6	7
`to_leaf =`	0	0	0	0	1	0	1
`max_length =`	0	0	0	0	1	0	2

Visualização do algoritmo que computa o diâmetro com DP



	1	2	3	4	5	6	7
to_leaf =	0	0	0	2	1	0	1
max_length =	0	0	0	4	1	0	2

Visualização do algoritmo que computa o diâmetro com DP



	1	2	3	4	5	6	7
`to_leaf =`	0	0	0	2	1	0	1
`max_length =`	0	0	0	4	1	0	2

Implementação da rotina que computa o diâmetro com DP

```cpp
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;

const int MAX { 1010 };
vector<int> adj[MAX];
int to_leaf[MAX], max_length[MAX];

void dfs(int u, int p) {

    vector<int> ds;

    for (const auto& v : adj[u]) {
        if (v == p) continue;

        dfs(v, u);
        ds.push_back(to_leaf[v]);
    }

    sort(ds.begin(), ds.end());
```

Implementação da rotina que computa o diâmetro com DP

```

    to_leaf[u] = ds.empty() ? 0 : ds.back() + 1;

    int N = ds.size();

    switch (N) {
    case 0:
        max_length[u] = 0;
        break;

    case 1:
        max_length[u] = ds.back() + 1;
        break;

    default:
        max_length[u] = ds[N - 1] + ds[N - 2] + 2;
    }
}

```

Implementação da rotina que computa o diâmetro com DP

```cpp
int diameter(int root, int N)
{
    dfs(root, 0);

    int d = 0;

    for (int u = 1; u <= N; ++u)
        d = max(d, max_length[u]);

    return d;
}

int main()
{
    vector<ii> edges { ii(1, 7), ii(3, 7), ii(7, 4), ii(4, 2),
        ii(4, 5), ii(5, 6) };

    for (const auto& [u, v] : edges) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
```

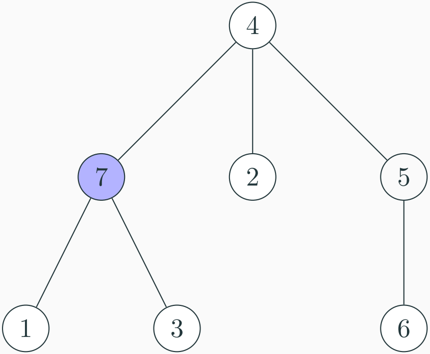
Implementação da rotina que computa o diâmetro com DP

```
    // 4
    cout << diameter(4, 7) << endl;

    // 0 0 0 2 1 0 1
    for (int u = 1; u <= 7; ++u)
        cout << to_leaf[u] << (u == 7 ? '\n' : ' ');

    // 0 0 0 4 1 0 2
    for (int u = 1; u <= 7; ++u)
        cout << max_length[u] << (u == 7 ? '\n' : ' ');

    return 0;
}
```

Diâmetro com DFS

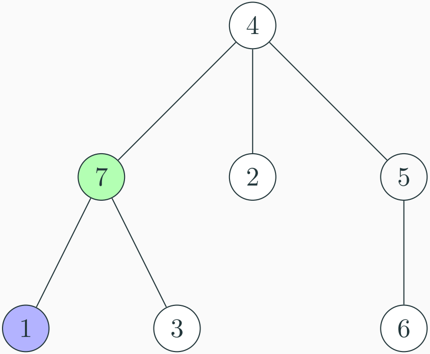
- Computar o diâmetro usando DFS é simples de implementar, mas a corretude do algoritmo não é trivial
- Basta escolher dois vértices distintos u e v quaisquer
- Em seguida, identifique o vértice w mais distante de u
- Agora, compute o vértice t mais distante de w
- O diâmetro será a distância entre w e t
- A corretude é provada em duas etapas
- Primeiro, mostre que ao menos um vértice x do caminho de u a w deve fazer parte de um caminho cujo tamanho é o diâmetro
- Em seguida, use este fato para provar que w deve ser o extremo de um caminho máximo
- Ambos fatos podem ser demonstrados por contradição

Visualização do algoritmo que computa o diâmetro com DFS



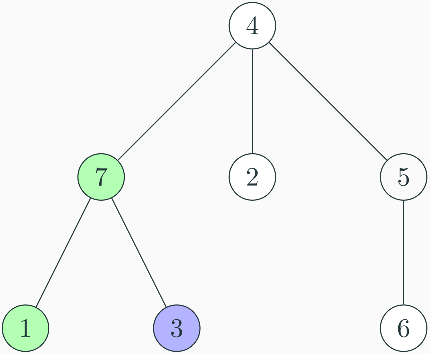
	1	2	3	4	5	6	7
`dist to 7 =`	∞	∞	∞	∞	∞	∞	0

Visualização do algoritmo que computa o diâmetro com DFS



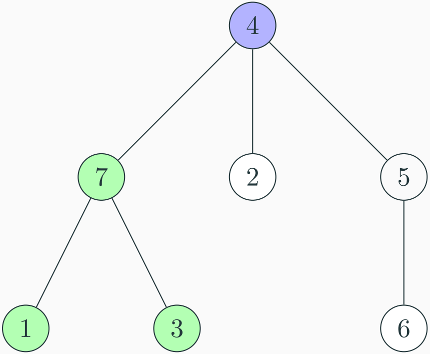
	1	2	3	4	5	6	7
`dist to 7 =`	1	∞	∞	∞	∞	∞	0

Visualização do algoritmo que computa o diâmetro com DFS



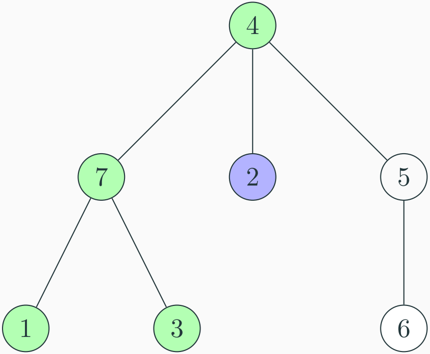
	1	2	3	4	5	6	7
`dist to 7 =`	1	∞	1	∞	∞	∞	0

Visualização do algoritmo que computa o diâmetro com DFS



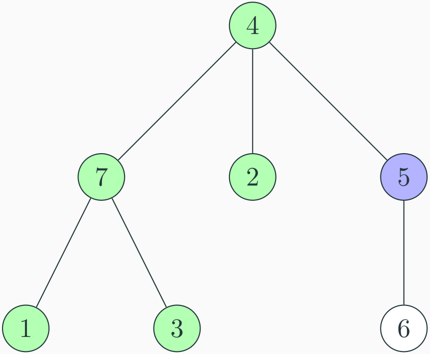
	1	2	3	4	5	6	7
`dist to 7 =`	1	∞	1	1	∞	∞	0

Visualização do algoritmo que computa o diâmetro com DFS



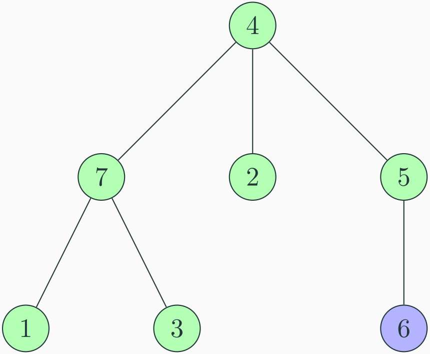
	1	2	3	4	5	6	7
`dist to 7 =`	1	2	1	1	∞	∞	0

Visualização do algoritmo que computa o diâmetro com DFS



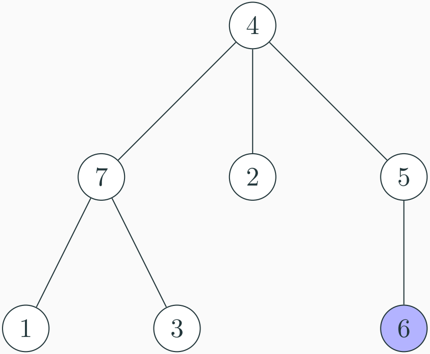
	1	2	3	4	5	6	7
`dist to 7 =`	1	2	1	1	2	∞	0

Visualização do algoritmo que computa o diâmetro com DFS



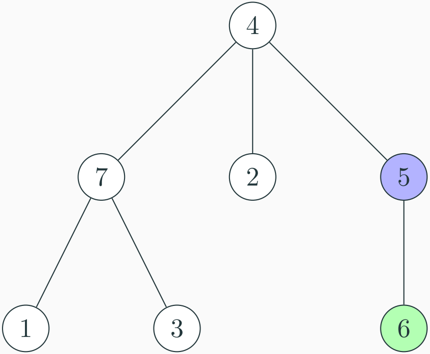
	1	2	3	4	5	6	7
`dist to 7 =`	1	2	1	1	2	3	0

Visualização do algoritmo que computa o diâmetro com DFS



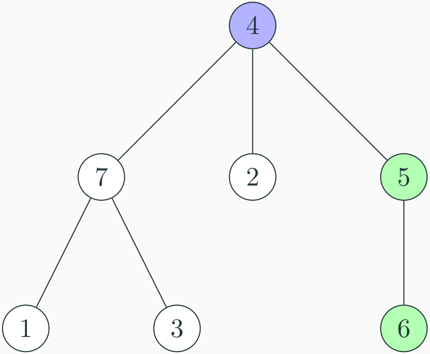
	1	2	3	4	5	6	7
`dist to 6 =`	∞	∞	∞	∞	∞	0	∞

Visualização do algoritmo que computa o diâmetro com DFS



	1	2	3	4	5	6	7
`dist to 6 =`	∞	∞	∞	∞	1	0	∞

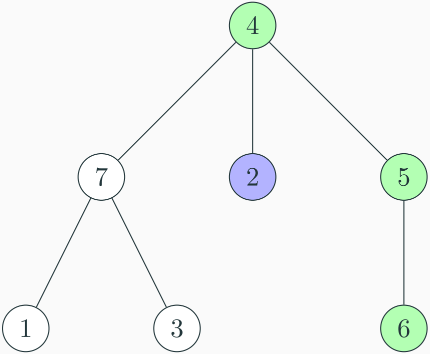
Visualização do algoritmo que computa o diâmetro com DFS



dist to 6 =

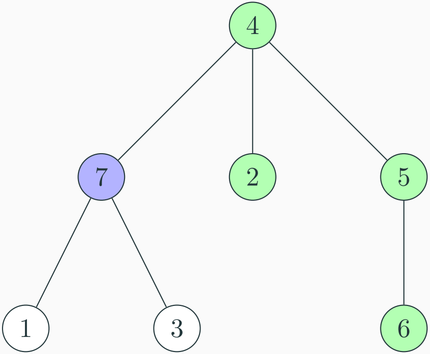
1	2	3	4	5	6	7
∞	∞	∞	2	1	0	∞

Visualização do algoritmo que computa o diâmetro com DFS



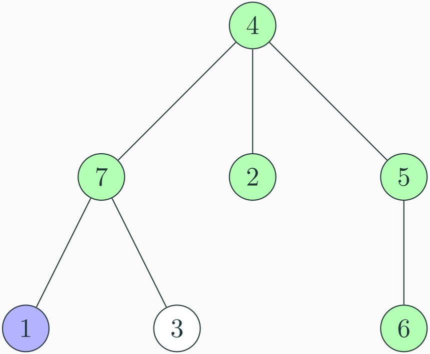
	1	2	3	4	5	6	7
`dist to 6 =`	∞	3	∞	2	1	0	∞

Visualização do algoritmo que computa o diâmetro com DFS



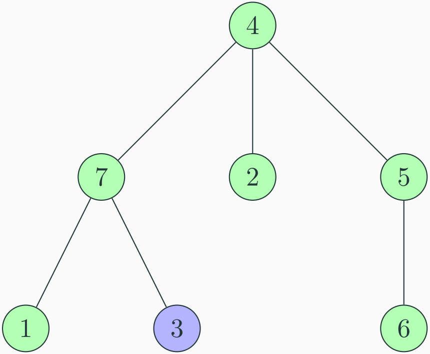
	1	2	3	4	5	6	7
`dist to 6 =`	∞	3	∞	2	1	0	3

Visualização do algoritmo que computa o diâmetro com DFS



	1	2	3	4	5	6	7
`dist to 6 =`	4	3	∞	2	1	0	3

Visualização do algoritmo que computa o diâmetro com DFS



dist to 6 =

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

Implementação da rotina que computa o diâmetro com DFS

```cpp
#include <bits/stdc++.h>

using namespace std;
using ii = pair<int, int>;

const int MAX { 1010 };
vector<int> adj[MAX];
int dist[MAX];

void dfs(int u, int p) {

    dist[u] = dist[p] + 1;

    for (const auto& v : adj[u]) {
        if (v == p)
            continue;

        dfs(v, u);
    }
}

```

Implementação da rotina que computa o diâmetro com DFS

```cpp
int diameter(int u, int N)
{
    dist[0] = -1;

    // dist = 1 2 1 1 2 3 0
    dfs(u, 0);

    auto it = max_element(dist + 1, dist + 1 + N);
    int w = it - dist;

    // dist = 4 3 4 2 1 0 3
    dfs(w, 0);

    it = max_element(dist + 1, dist + 1 + N);

    return *it;
}

```

Implementação da rotina que computa o diâmetro com DFS

```cpp
int main()
{
    vector<ii> edges { ii(1, 7), ii(3, 7), ii(7, 4), ii(4, 2),
        ii(4, 5), ii(5, 6) };

    for (const auto& [u, v] : edges) {
        adj[u].push_back(v);
        adj[v].push_back(u);
    }

    // 4
    cout << diameter(7, 7) << endl;

    return 0;
}
```

Referências

1. **HALIM**, Felix; **HALIM**, Steve. *Competitive Programming 3*, 2010.
2. **LAAKSONEN**, Antti. *Competitive Programmer's Handbook*, 2018.
3. **SKIENA**, Steven S.; **REVILLA**, Miguel A. *Programming Challenges*, 2003.