MaratonaCIn

Seletiva 2020

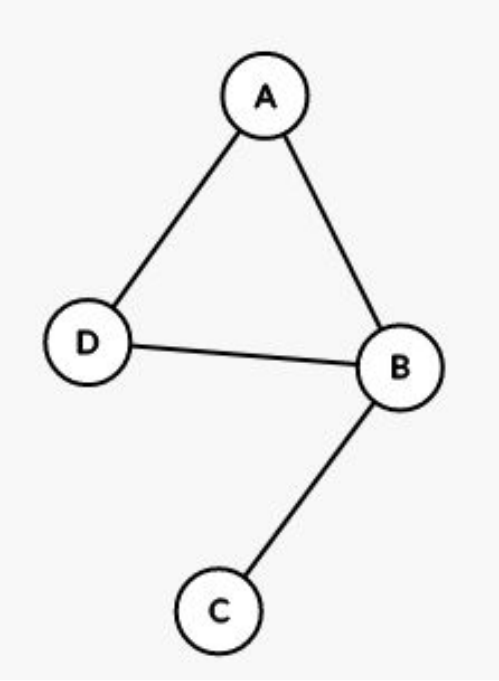
Graph representation
BFS / DFS
Toposort
Bicolorability

Aula 3

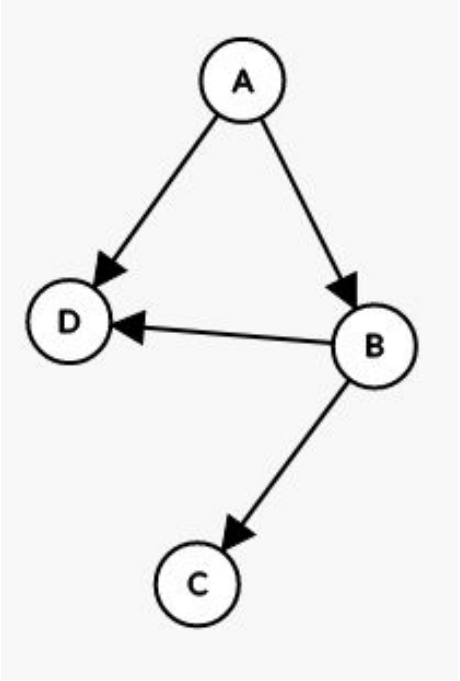
Graph representation

Graph representation

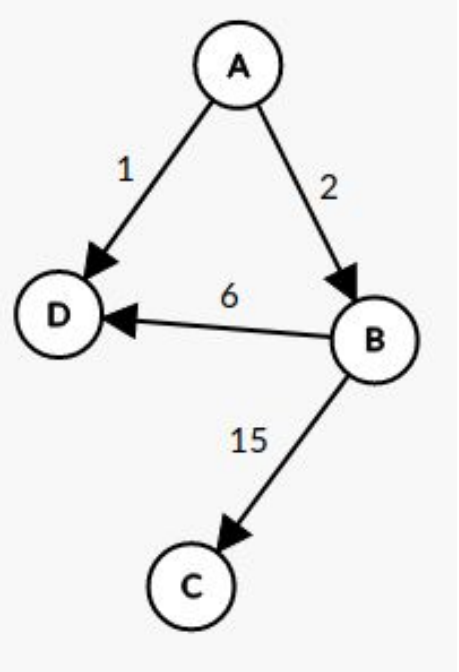
MaratonaCIn



NÃO-DIRECIONADO, SEM PESO



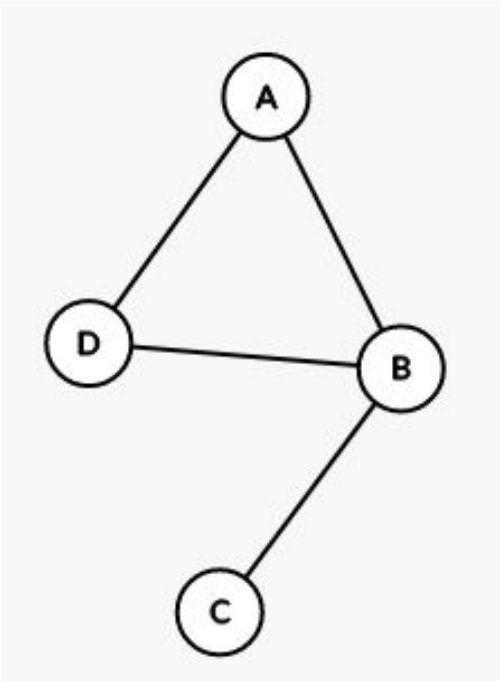
DIRECIONADO, SEM PESO



DIRECIONADO, COM PESO

Graph representation

MaratonaCIn

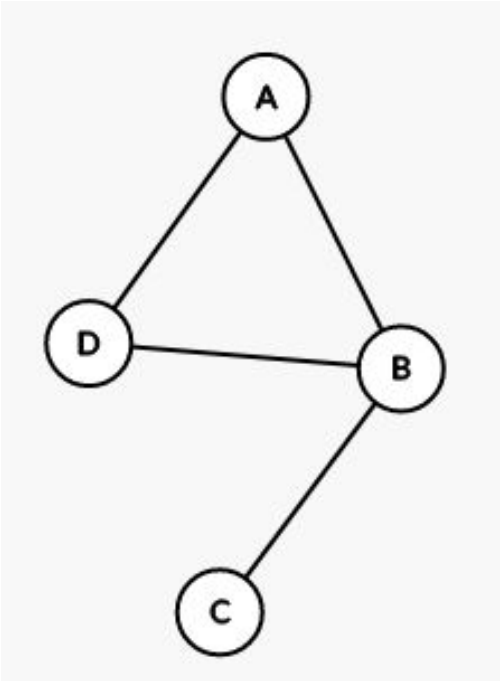


	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	0
D	1	1	0	0

MATRIZ DE ADJACÊNCIA

Graph representation

MaratonaCIn

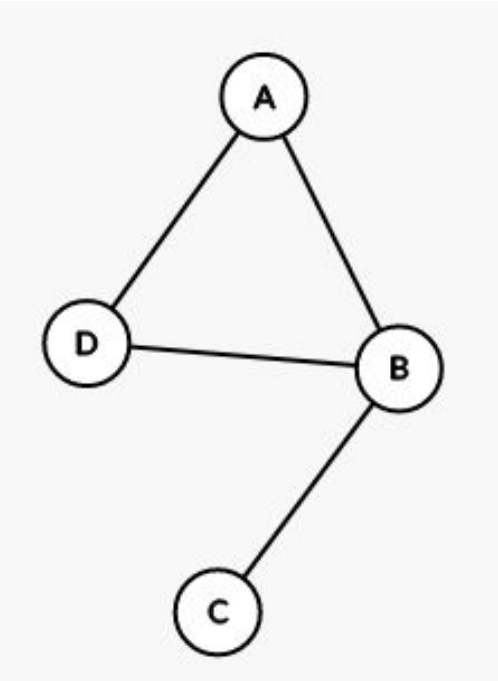



```
const int MAXN = 1000;

int adj[MAXN][MAXN];

int main() {
  adj['A']['B'] = 1;
  // adj['B']['A'] = 1;
  adj['A']['D'] = 1;
  // adj['D']['A'] = 1;
  adj['B']['D'] = 1;
  // adj['D']['B'] = 1;
    adj['B']['C'] = 1;
  // adj['C']['B'] = 1;
```

MATRIZ DE ADJACÊNCIA

Graph representation

MaratonaCIn

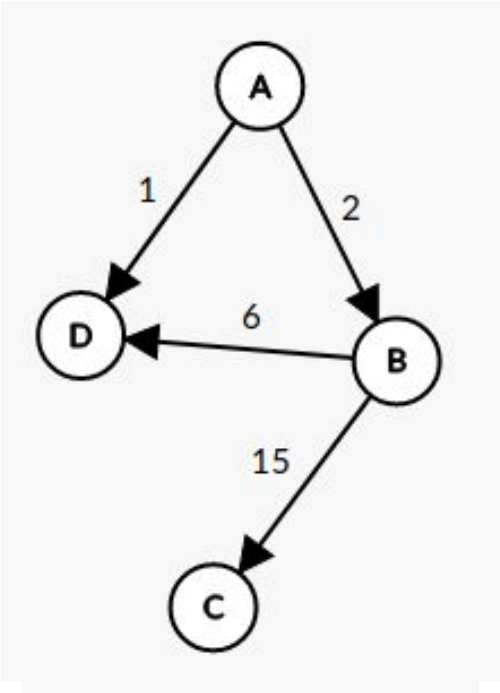


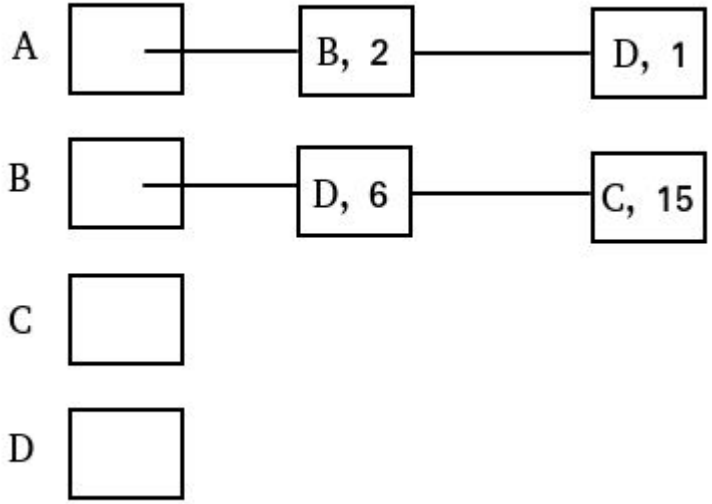

```
for (int u = 0; u < MAXN; ++u) {
  for (int v = 0; v < MAXN; ++v) {
    if (adj[v][u] != 0) {
      // aresta u -> v existente
      // aresta v -> u existente (bidirecional)
      // adj[v][u] = peso / adj[v][u] = 0 ou 1
    }
  }
}
```

MATRIZ DE ADJACÊNCIA

Graph representation

MaratonaCIn

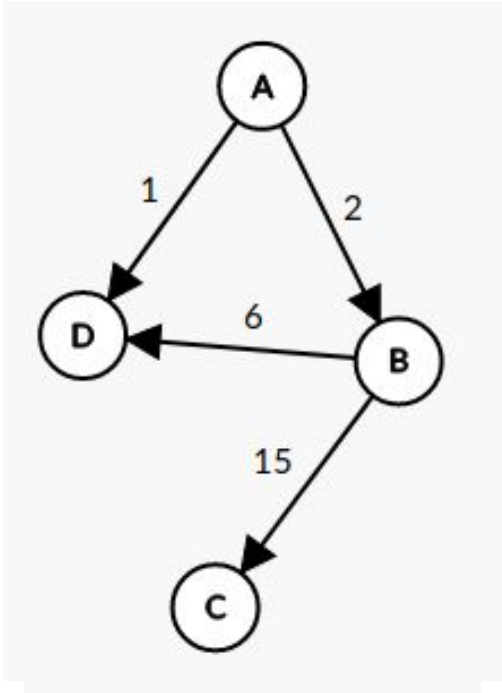




LISTA DE ADJACÊNCIA

Graph representation

MaratonaCIn

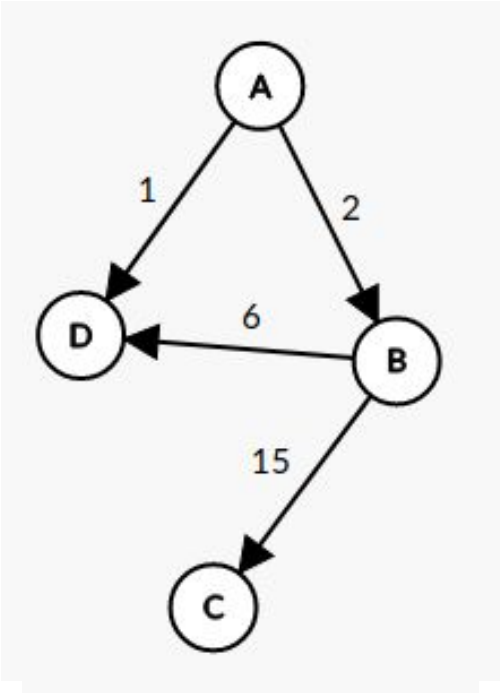



```
const int MAXN = 1000;

vector<pair<int, int>> adj[MAXN];

int main() {
  adj['A'].push_back(make_pair('B', 2)); // aresta A -> B, peso 2
  adj['A'].push_back(make_pair('D', 2)); // aresta A -> D, peso 1
  adj['B'].push_back(make_pair('D', 2)); // aresta B -> D, peso 6
  adj['B'].push_back(make_pair('C', 2)); // aresta B -> C, peso 15
```

LISTA DE ADJACÊNCIA

Graph representation

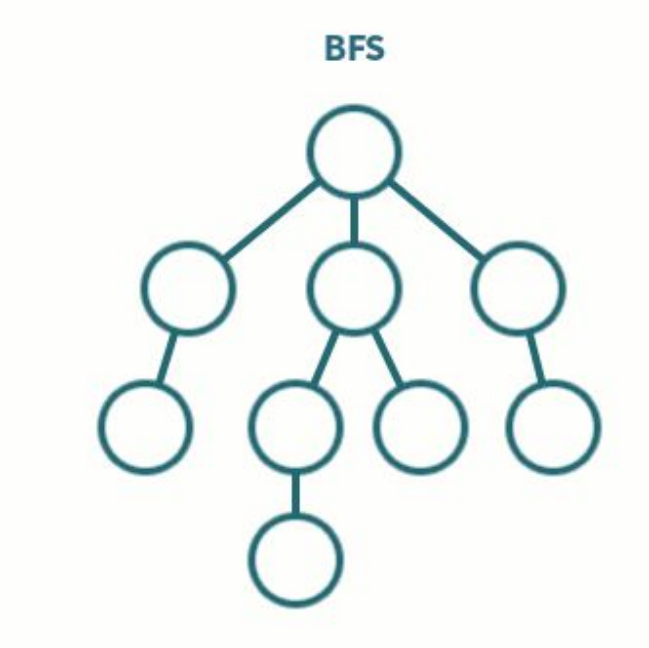
MaratonaCIn



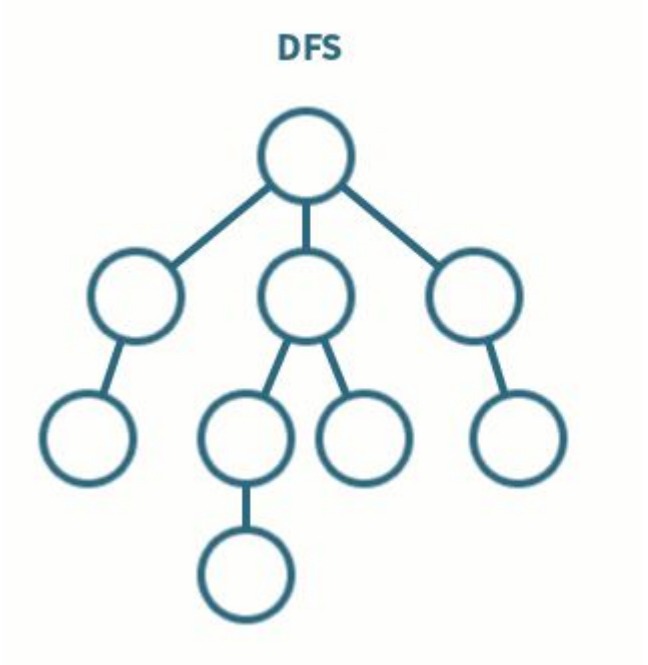

```
for (int i = 0; i < adj[v].size(); ++i) {
    int u = adj[v][i].first;
    int w = adj[v][i].second;
    // aresta v -> u, com peso = w
  }

  for (auto [u, w] : adj[v]) {
    // aresta v -> u, com peso = w
    // C++17
  }
```

LISTA DE ADJACÊNCIA

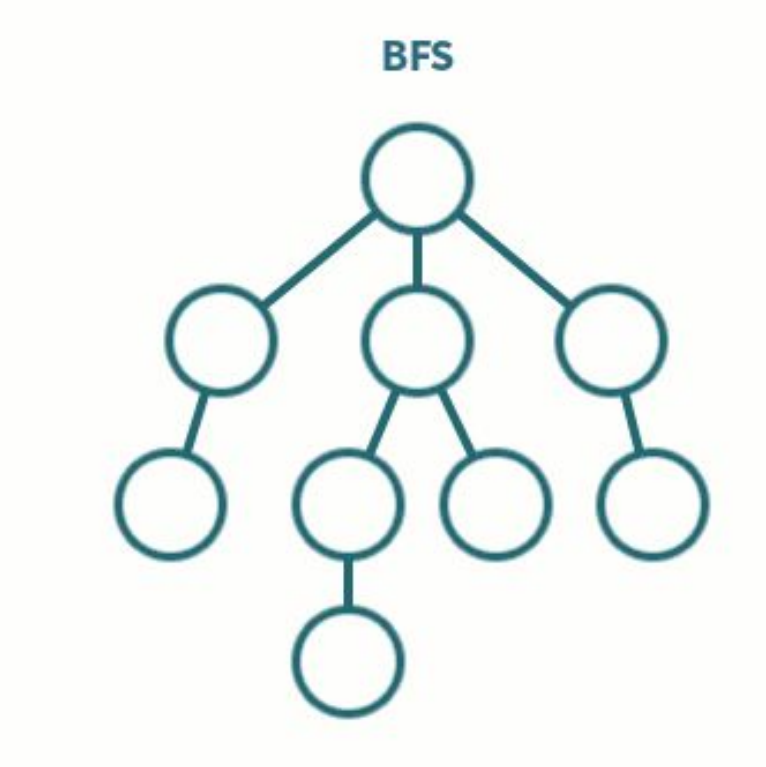
MaratonaCIn



BFS / DFS



Breadth-First Search (BFS)



- Busca nível a nível, a partir da origem
- Algoritmo iterativo
- Uso de uma estrutura FIFO (std::queue)

Breadth-First Search (BFS)

```
void bfs(int origem) {
  memset(dist, -1, sizeof dist);
  queue<int> fila;
  dist[origem] = 0; fila.push(origem);
  while (!fila.empty()) {
    int u = fila.front(); fila.pop();
    for (int v : adj[u]) {
      if (dist[v] == -1) {
        dist[v] = dist[u] + 1;
        fila.push(v);
      }
    }
  }
}
```

MaratonaCIn

COMPLEXIDADE

Memória -> O(V + E)

Tempo -> O(V + E)

Breadth-First Search (BFS)

```
void bfs(int origem) {
  memset(dist, -1, sizeof dist);
  queue<int> fila;
  dist[origem] = 0; fila.push(origem);
  while (!fila.empty()) {
    int u = fila.front(); fila.pop();
    for (int v = 0; v < MAXN; ++v) {
      if (adj[u][v] != 0) {
        if (dist[v] == -1) {
          dist[v] = dist[u] + 1;
          fila.push(v);
        }
      }
    }
  }
```

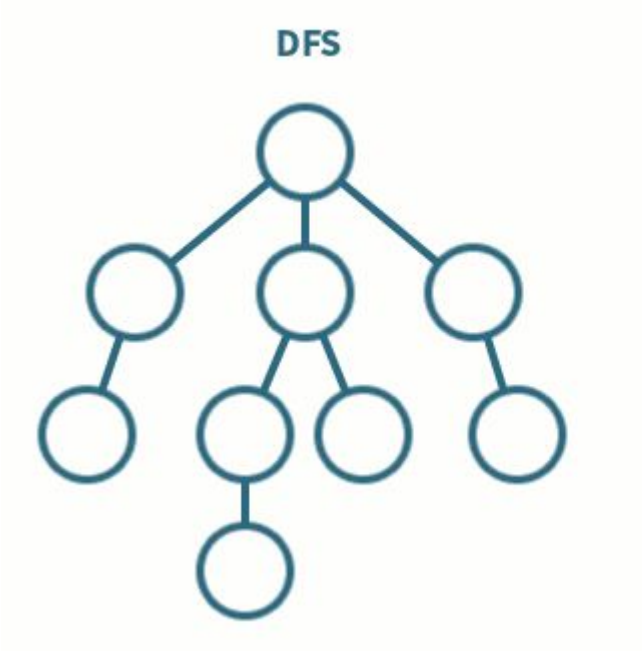
COMPLEXIDADE

Memória -> O(V * V)

Tempo -> O(V * V)

Depth-First Search (DFS)

MaratonaCIn



- Explora o mais longe/profundo que puder antes de retroceder
- Algoritmo recursivo (geralmente!)
- Cuidado com Stack Overflow (raro, mas pode acontecer)

Depth-First Search (DFS)

MaratonaCIn

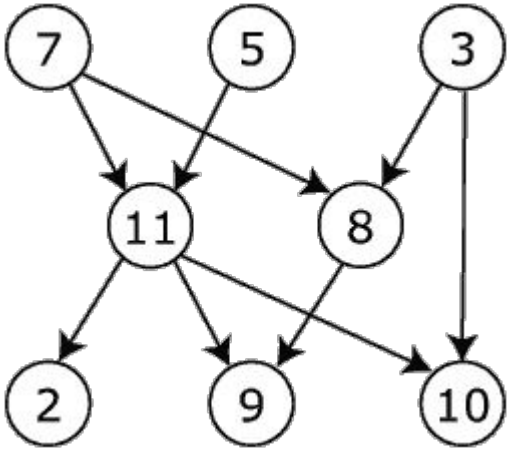
```
vector<int> adj[ms];
bool vis[ms];
void dfs(int u) {
  vis[u] = true;
  for (int v : adj[u]) {
    if (vis[v]) continue;
    dfs(v);
  }
}
```

COMPLEXIDADE

Memória -> O(V)

Tempo -> O(V + E)

Topological Sort



- **7, 5, 3, 11, 8, 2, 9, 10** (visual esquerda-para-direita, de-cima-para-baixo)
- **3, 5, 7, 8, 11, 2, 9, 10** (vértice de menor número disponível primeiro)
- **3, 7, 8, 5, 11, 10, 2, 9**
- **5, 7, 3, 8, 11, 10, 2, 9** (menor número de arestas primeiro)
- **7, 5, 11, 3, 10, 8, 9, 2** (vértice de maior número disponível primeiro)
- **7, 5, 11, 2, 3, 8, 9, 10**

- Uma ordem para os nós de um grafo dirigido acíclico (DAG).
- Um nó vem antes de todos nós que ele alcança
- Podem existir várias ordenações que obedecem essa restrição

Topological Sort with DFS

MaratonaCIn

```
vector<int> adj[ms];
bool vis[ms];
stack<int> S;
void dfs(int u) {
  vis[u] = true;
  for (int v : adj[u]) {
    if (vis[v]) { /* cycle => no ordering
*/ }
    dfs(v);
  }
  S.push(u);
}
```

- Inicializa a partir de todos os nós que têm grau de entrada igual a zero
- Encontra uma das possíveis ordenações, mas não tem como usar critérios de desempate

Topological Sort with BFS (Kahn Algorithm)

MaratonaCIn

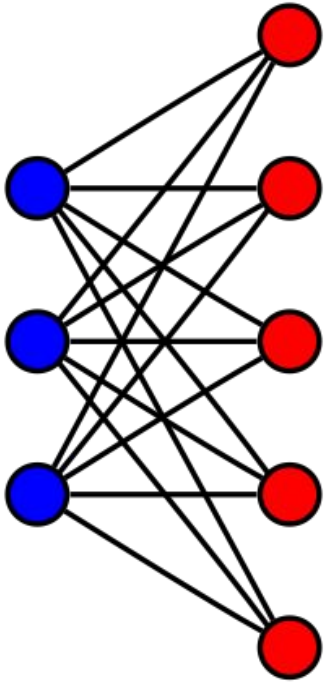
```
void bfs() {
  vector<int> topo_order;
  priority_queue<int, vector<int>, cmp> PQ;
  for (int u = 0; u < n; ++u) {
    if (indegree[u] == 0) PQ.push(u);
  }
  while (!PQ.empty()) {
    int u = PQ.top();
    PQ.pop();
    topo_order.push_back(u);
    for (int v : adj[u]) {
      if (--indegree[v] == 0) {
        PQ.push(v);
      }}}}
```

- Vantagem: se usar uma heap em vez de uma fila, pode especificar a ordem em que os elementos são escolhidos

- Se, ao terminar, topo_order.size() != n, não há ordenação possível

Bicolorability / Bipartite graph

MaratonaCIn



- Colorir os nós de um grafo com apenas duas cores
- Vértices adjacentes não podem ter cores iguais
- Um grafo é bicolorável sse ele é um grafo bipartido

Bicolorability with DFS


```cpp
vector<int> adj[ms];
int colour[ms]; // inicializa como -1
void dfs(int u) {
  for (int v : adj[u]) {
    if (colour[v] == -1) {
      colour[v] = 1 - colour[u];
      dfs(v);
    }
  }
}
```

```cpp
int main() {
  // ...
  for (int u = 0; u < n; ++u) {
    if (colour[u] == -1) {
      colour[u] = 0;
      dfs(u);
    }
  }
  // check if possible
  bool possible = true;
  for (int u = 0; u < n; ++u) {
    for (int v : adj[u]) {
      possible = possible &&  colour[u] != colour[v];
    }
  }
}
```