

Percursos em Grafos

prof. Fábio Luiz Usberti

MC521 - Desafios de Programação I

Instituto de Computação - UNICAMP - 2018

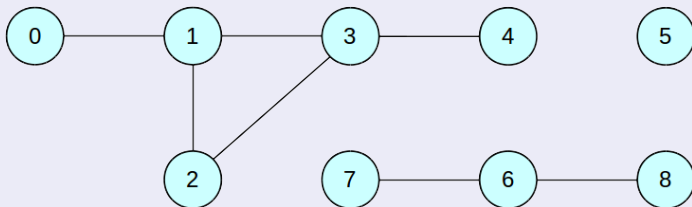
- 1 Busca em Profundidade
- 2 Busca em Largura
- 3 Componentes Conexos
- 4 Ordenação Topológica
- 5 Grafos Bipartidos
- 6 Referências

Busca em Profundidade

- Busca em profundidade (“depth first search” – DFS) é um algoritmo de percurso em grafo que se inicia em um nó raiz para, em seguida, realizar chamadas recursivas da busca em cada nó vizinho não-visitado.
- Ao atingir um nó que não possui vizinhos não-visitados, o algoritmo DFS recua para o nó anterior e busca recursivamente outro vizinho não-visitado, se houver.
- A complexidade do DFS depende da estrutura de dados utilizada:
 - 1 Lista de adjacência: $O(|V| + |E|)$.
 - 2 Matriz de adjacência: $O(|V|^2)$.

Busca em Profundidade

- Supondo que a chamada `dfs(0)` realiza a DFS com origem no vértice 0 e que a vizinhança de um vértice está armazenada de modo crescente pelo rótulo do vértice.
- Nesse caso, para o grafo abaixo a sequência de visitação será $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.



Busca em Profundidade

```
#define UNVISITED -1
#define VISITED 1

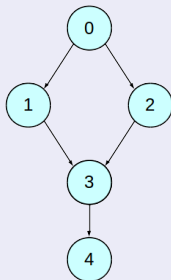
typedef pair<int, int> ii;    // In this chapter, we will frequently use these
typedef vector<ii> vii;      // three data type shortcuts. They may look cryptic
typedef vector<int> vi;      // but shortcuts are useful in competitive programming

vi dfs_num;    // this variable has to be global, we cannot put it in recursion
vector<vii> AdjList;

void dfs(int u) {           // DFS for normal usage: as graph traversal algorithm
    printf(" %d", u);      // this vertex is visited
    dfs_num[u] = VISITED;  // important step: we mark this vertex as visited
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];    // v is a (neighbor, weight) pair
        if (dfs_num[v.first] == UNVISITED) // important check to avoid cycle
            dfs(v.first);         // recursively visits unvisited neighbors v of vertex u
    } }
```

Busca em Profundidade

- **UVa 11902 - Dominator:** Um vértice X é um dominador de um vértice Y se todo o caminho que começa da raiz (nó 0) até o nó Y necessariamente passa pelo nó X . Se o nó Y não é alcançável a partir da raiz, então Y não possui um dominador. Além disso, todo nó alcançável a partir da raiz é seu próprio dominador. Dado um grafo com $|V| < 100$, determine os dominadores de todos os vértices.



Busca em Profundidade

- Esse problema pode ser resolvido com o seguinte algoritmo $O(|V|^2 + |V||E|)$:
 - 1 Verifique quais são os nós alcançáveis a partir da raiz 0 , executando `dfs(0)`.
 - 2 Para verificar quais são os nós dominados por um nó X , este nó é temporariamente removido do grafo e o método `dfs(0)` é executado novamente.
 - 3 Um nó Y é dominado por X quando: (i) a busca DFS no grafo original alcançou o nó Y e (ii) a busca DFS não alcançou Y após a remoção de X .
 - 4 Repetir para todo $X \in [0, \dots, |V| - 1]$.
- **Obs.:** Não é necessário remover de fato o nó X da estrutura em cada iteração. Basta simplesmente iniciar o nó X como visitado.

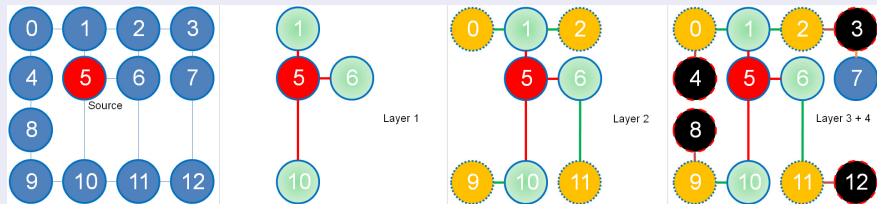
Busca em Largura

- Busca em largura (“breadth first search” – BFS) é um algoritmo de percurso em grafo que se inicia em um nó raiz 0 e visita todos os nós vizinhos a 0 (**primeira camada**) para depois visitar os vizinhos (não-visitados) dos vizinhos (**segunda camada**), e assim sucessivamente, camada por camada.
- A complexidade do BFS depende da estrutura de dados utilizada:
 - 1 **Lista de adjacência:** $O(|V| + |E|)$.
 - 2 **Matriz de adjacência:** $O(|V|^2)$.

Busca em Largura

- Supondo que o algoritmo BFS utiliza como raiz o nó 5 e que a vizinhança de um vértice está armazenada de modo crescente pelo número do vértice. Nesse caso, para o grafo abaixo a sequência de visitação será:

- 1 Camada 0: 5.
- 2 Camada 1: 1 → 6 → 10.
- 3 Camada 2: 0 → 2 → 11 → 9.
- 4 Camada 3: 4 → 3 → 12 → 8.
- 5 Camada 4: 7.



Busca em Largura

```
// inside int main()—no recursion
vi d(V, INF); d[s] = 0;    // distance from source s to s is 0
queue<int> q; q.push(s);   // start from source

while (!q.empty()) {
    int u = q.front(); q.pop();    // queue: layer by layer!
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];    // for each neighbor of u
        if (d[v.first] == INF) {  // if v.first is unvisited + reachable
            d[v.first] = d[u] + 1; // make d[v.first] != INF to flag it
            q.push(v.first);       // enqueue v.first for the next iteration
        }
    }
}
```

Busca em Largura

- Alguns problemas exigem como saída a **reconstrução do caminho mais curto**, e não apenas o comprimento do caminho;
- Exemplo: o caminho mais curto entre os vértices **5** e **7** é **$5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7$** ;
- Isso pode ser facilmente feito utilizando um vetor de inteiros **p** . Cada vértice **v** lembra de seu pai **u** (**$p[v] = u$**);
- Exemplo: para o caminho mais curto entre os vértice **5** e **7** onde **$5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7$** , o vértice **7** terá o vértice **3** como seu pai; o vértice **3** terá o vértice **2** como seu pai, e assim sucessivamente até que o vértice **1** terá o vértice **5** (origem) como seu pai;

Busca em Largura

```
// uses information from vector p
void printPath(int u) {
    // base case, at the source s
    if (u == s) {
        printf("%d", s); return;
    }
    // recursive: to make the output format: s -> ... -> t
    printPath(p[u]);
    printf(" %d", u);
}
```

Aplicação de Percursos em Grafos

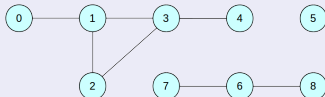
- Uma única chamada de `dfs(u)` (ou `bfs(u)`) visita somente vértices conectados a `u`. Esse fato pode ser utilizado para **encontrar e enumerar os componentes conexos** de um grafo não-orientado.
- Para isso é suficiente uma simples **adaptação do algoritmo DFS**: após o término do percurso, reiniciar a busca a partir de um nó ainda não-visitado. Se houver tal nó, um novo componente conexo foi encontrado.

Aplicação de Percursos em Grafos

```
// inside int main()—this is the DFS solution
numCC = 0;
dfs_num.assign(V, UNVISITED);    // this sets all vertices' state to UNVISITED
for (int i = 0; i < V; i++)      // for each vertex i in [0..V-1]
    if (dfs_num[i] == UNVISITED) // if that vertex is not visited yet
        printf("Component %d:", ++numCC), dfs(i), printf("\n"); // 3 lines here!
printf("There are %d connected components\n", numCC);
```

- Para o grafo abaixo, a saída será:

```
Component 1: 0 1 2 3 4
Component 2: 5
Component 3: 6 7 8
There are 3 connected components
```



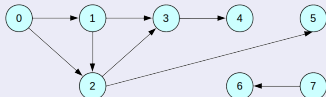
Aplicação de Percursos em Grafos

- Uma **ordenação topológica** tem por objetivo encontrar em um **grafo orientado acíclico** (“directed acyclic graph” – DAG) uma ordem para os vértices de modo que para cada arco (u, v) do DAG, tem-se que $u \prec v$ (u precede v) na ordenação topológica.
- Todo DAG possui pelo menos uma e possivelmente múltiplas ordenações topológicas.
- **Exemplo:** encontrar uma sequência de disciplinas tal que um aluno complete todos os créditos necessários para integralização de um curso. Uma disciplina pode conter pré-requisitos e estes nunca são cíclicos, logo as disciplinas podem ser modeladas por um DAG.

Aplicação de Percursos em Grafos

- Um algoritmo para a solução do problema de ordenação topológica pode ser obtido modificando o algoritmo DFS.
- O algoritmo, proposto por Robert E. Tarjan, inclui um nó u na lista de nós (ordenados topologicamente) quando todos as subárvores de u na árvore geradora DFS já foram visitados.
- Para o grafo abaixo, uma solução possível para o problema de ordenação topológica é:

7 6 0 1 2 5 3 4



Aplicação de Percursos em Grafos

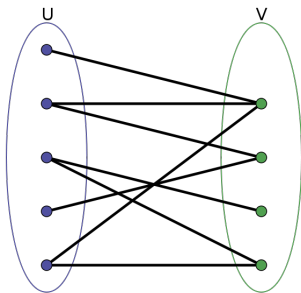
```
vi topoSort; // global vector to store the toposort in reverse order

void dfs2(int u) { // change function name to differentiate with original dfs
    dfs_num[u] = VISITED;
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        int v = AdjList[u][j];
        if (dfs_num[v.first] == UNVISITED)
            dfs2(v.first);
    }
    topoSort.push_back(u); // this is the only change from DFS
}

int main() {
    // make sure that the given graph is DAG
    printThis("Topological Sort (the input graph must be DAG)");
    topoSort.clear();
    dfs_num.assign(V, UNVISITED);
    for (int i = 0; i < V; i++) // this part is the same as finding CCs
        if (dfs_num[i] == UNVISITED)
            dfs2(i);
    reverse(topoSort.begin(), topoSort.end()); // reverse topoSort
    for (int i = 0; i < (int)topoSort.size(); i++) // or you can simply read
        printf(" %d", topoSort[i]); // the content of topoSort backwards
    printf("\n");
    return 0;
}
```

Aplicação de Percursos em Grafos

- Um **grafo bipartido** é um grafo cujos vértices podem ser divididos em dois conjuntos disjuntos U e V tais que toda aresta conecta um vértice em U a um vértice em V .
- Um grafo G é bipartido se e somente se G não possui **ciclos de tamanho ímpar**.
- Um grafo G é bipartido se e somente se G é **2-colorível**, o que significa que é possível colorir os vértices de G utilizando até 2 cores de modo que vértices adjacentes possuam cores distintas.



Aplicação de Percursos em Grafos

- Para **verificar se um grafo é bipartido**, é possível utilizar o algoritmo BFS da seguinte forma:
 - 1 Colorir o nó raiz de branco (camada 0).
 - 2 Colorir os nós vizinhos à raiz de preto (camada 1).
 - 3 Colorir os nós vizinhos dos vizinhos da raiz de branco (camada 2).
 - 4 Continue alternando as cores camada a camada da busca BFS. Se houver dois nós adjacentes com a mesma cor, então **o grafo não é bipartido**, pois foi encontrado um ciclo de tamanho ímpar.

Aplicação de Percursos em Grafos

```
int main() {  
    queue<int> q; q.push(s);  
    vi color(V, INF); color[s] = 0;  
    bool isBipartite = true; // addition of one boolean flag, initially true  
    while (!q.empty() & isBipartite) { // similar to the original BFS routine  
        int u = q.front(); q.pop();  
        for (int j = 0; j < (int)AdjList[u].size(); j++) {  
            int v = AdjList[u][j];  
            if (color[v.first] == INF) { // but, instead of recording distance,  
                color[v.first] = 1 - color[u]; // we just record two colors {0, 1}  
                q.push(v.first);  
            }  
            else if (color[v.first] == color[u]) { // u & v.first has same color  
                isBipartite = false; break; // we have a coloring conflict  
            }  
        }  
    }  
}
```

Referências

- ❶ S. Halim e F. Halim. Competitive Programming 2, Second Edition Lulu (www.lulu.com), 2011. (IMECC – 005.1 H139c)
- ❷ S. S. Skiena, M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
- ❸ T.H. Cormen, C.E. Leiserson, R.L.Rivest e C. Stein. Introduction to Algorithms. 2nd Edition, McGraw-Hill, 2001. (no. chamada IMECC – 005.133 Ar64j 3.ed.)
- ❹ U. Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley. 1989. (no. chamada IMECC – 005.133 Ec53t 2.ed.)