Graphs: Depth-First Search (DFS)

L.EIC

Algorithms and Data Structures

2024/2025



P Ribeiro, AP Tomas

L.EIC (AED)

- One of the most important tasks it to **traverse** a graph, that is, **passing through all the nodes** using the **connections between them**
- This is known as a graph search (or graph traversal)
- There are two fundamental graph search methodologies that vary on order in which they traverse the nodes:
 - Depth-First Search DFS

Traverse the entire subgraph connected to a neighbor before entering the next neighbor node

Breadth-First Search - BFS

Traverse the nodes by increasing distance of number of links to reach them













- In their essence, DFS and BFS do the "same": traverse all the nodes
- When to use one or the other depends on the **order that betters** suits the problem that you are solving

• We will show how to **implement** both and give examples of applications

Representing Graphs in C++

- To implement graph search we first need to represent a graph
- There is no graph data structure in the C++ standard
 - This is mainly due to the flexibility and different variations of a graph (adj. list/matrix, node labeling, undirected/directed, weighted/unweighted, nodes/edge colors, temporal graphs,etc)
 - A "one size fits them all" approach would have too much overhead (a custom graph class can be much more suited for a specific situation)
 - ▶ In case you are curious there is Boost Graph Library (BGL)

Representing Graphs in C++

- For the purposes of this class we will (mainly) be using a **simple** graph class to offer some reusability:
 - The class should be very lightweight, with only the essentials (add what's needed for a problem, focus more on the algorithms)
 - Support for simple graphs only (no self-loops or parallel edges)
 - Support directed/undirected and weighted/unweighted graphs
 - We will use adjacency lists as the edges representation
 - We will assume nodes are labeled from 1 to |V|

(note: as explained before, we could have a .h with declarations and a .cpp with implementation; here we will use a single .h file to simplify code submissions)

A simple lightweight Graph class

• A simple lightweight graph class: graph.h

```
class Graph {
  struct Edge {
    int dest: // Destination node
    int weight; // An integer weight
  };
  struct Node {
    std::list<Edge> adj; // The list of outgoing edges (to adjacent nodes)
  }:
  int n:
                         // Graph size (vertices are numbered from 1 to n)
  bool hasDir; // false: undirected; true: directed
  std::vector<Node> nodes; // The list of nodes being represented
public:
  // Constructor: nr nodes and direction (default: undirected)
  Graph(int nodes, bool dir = false) { ... }
  // Add edge from source to destination with a certain weight
  void addEdge(int src, int dest, int weight = 1) { ... }
};
```

A simple lightweight Graph class

• The two included methods:

```
Graph(int num, bool dir = false) : n(num), hasDir(dir), nodes(num+1) {}
void addEdge(int src, int dest, int weight = 1) {
    if (src<1 || src>n || dest<1 || dest>n) return;
    nodes[src].adj.push_back({dest, weight});
    if (!hasDir) nodes[dest].adj.push_back({src, weight});
}
```

• Example usage:

<pre>Graph g(9, false); // 9 nodes, undirected graph g.addEdge(1, 2); // assuming weight=1 (unweighted)</pre>	1
g.addEdge(1, 3);	
g.addEdge(1, 4);	\square
g.addEdge(2, 4);	
g.addEdge(3, 4);	(4)
g.addEdge(4, 5);	CT T
g.addEdge(4, 6);	
g.addEdge(4, 7);	
g.addEdge(5, 8);	
g.addEdge(7, 9);	(8) (9)

Graphs: Depth-First Search (DFS)

2024/2025

10/40

Depth-First Search - DFS

The "backbone" of a DFS:

DFS (recursive version)

Complexity:

- Temporal:
 - ► Adjacency List: O(|V| + |E|)
 - Adjacency Matrix: O(|V|²)

Spatial: O(|V|)

L.EIC (AED)

DFS: Implementation

• Let's see a possible implementation with some livecoding

```
// we also need to add the attribute
// 'visited' to a node
void dfs(int v) {
  std::cout << v << " "; // show nodes
  nodes[v].visited = true;
  for (auto e : nodes[v].adj) {
    int w = e.dest;
    if (!nodes[w].visited)
        dfs(w);
  }
}</pre>
```

DFS (recursive version)

```
dfs(node v):
  mark v as visited
For all neighbors w of v do
  If w has not yet been visited then
     dfs(w)
```

12/40

• Example execution (assuming graph g was already created)



Example Application: Connected Components

- Find the number of connected components of a graph G
- Example: the following graph has 3 connected components



Example Application: Connected Components

The "backbone" of a program to solve it:

```
Finding connected components

counter \leftarrow 0

set all nodes as unvisited

For all nodes v of the graph do

If v has not yet been visited then

counter++

dfs(v)

return counter
```

Temporal complexity:

- Adjacency List: O(|V| + |E|)
- Adjacency Matrix: $O(|V|^2)$

Example Application: Connected Components

• In C++ code (and some *livecoding*):

```
int connectedComponents() {
    int counter = 0;
    for (int v=1; v<=n; v++)
        nodes[v].visited = false;
    for (int v=1; v<=n; v++)
        if (!nodes[v].visited) {
            counter++;
            std::cout << "connected component: ";
            dfs(v);
            cout << endl;
        }
    return counter;
}</pre>
```





L.EIC (AED)

Graphs: Depth-First Search (DFS)

2024/2025 15/40

Implicit Graphs

- We do not always need to explicitly store the graph
- Example: find the number of "blobs" (connected spots) in a matrix. Two cells are adjacent if they are connected vertically or horizontally.

#.####			1.2233
###			133
##	>	4 blobs>	
##			44

- To solve we simply need to do dfs(x, y) to visit the cell (x, y) where the neighbors are (x + 1, y), (x - 1, y), (x, y + 1) and (x, y - 1)
- Using DFS to "color" the connected components is known as doing a **Flood Fill**.

Topological Sorting

- Given a DAG G (directed acyclic graph), find an order of nodes such that u comes before v if and only if there is no edge (v, u)
- Example: For the graph below a possible topological sorting would be: 1,2,3,4,5,6 (or 1,4,2,5,3,6 there are other possible valid orders)



A classic example of application is to decide in which order to execute a set of tasks with precedences.

Topological Sorting

• How to solve this problem with DFS? What is the relationship between topological sorting and the DFS node order?

```
Topological Sorting - time: O(|V| + |E|) (list)
order \leftarrow empty
set all nodes as unvisited
For all nodes v of the graph do
  If v has not yet been visited then
     dfs(v)
return order
dfs(node v):
  mark v as visited
  For all neighbors w of v do
     If w has not yet been visited then
       dfs(w)
```

```
add v to the beginning of order
```

Topological Sorting



Example of execution:

•	order	$ = \emptyset$	
•	start	dfs(1	$ order = \emptyset$
•	start	dfs(4	$ order = \emptyset$
•	start	dfs(5) $ order = \emptyset $
•	start	dfs(6	$ order = \emptyset$
•	end	dfs(6	order = 6
•	end	dfs(5	order $= 5, 6$
•	end	dfs(4) $ order = 4, 5, 6$
•	end	dfs(1) $ order = 1, 4, 5, 6$
•	start	dfs(2) $ order = 1, 4, 5, 6 $
•	end	dfs(2	order = 2, 1, 4, 5, 6
•	start	dfs(3) $ order = 2, 1, 4, 5, 6 $
۹	end	dfs(3) $ order = 3, 2, 1, 4, 5, 6$
•	order	r = 3,	2, 1, 4, 5, 6

- Find if a (directed) graph G is acyclic
- Example: the left graph has a cycle; the right graph doesn't





Let's use 3 "colors":

- White unvisted node
- Gray node being visited (we are exploring its descendants)
- Black node already visited (we visited all its descendants)

```
Cycle Detection - \mathcal{O}(|V| + |E|) (list)
color[v \in V] \leftarrow white
For all nodes v of the graph do
  If color[v] = white then
     dfs(v)
dfs(node v):
  color[v] \leftarrow gray
  For all neighbors w of v do
     If color[w] = gray then
       write("Cycle found!")
     Else if color[w] = white then
       dfs(w)
  color[v] \leftarrow black
```

Example (starting on node 1) - graph with two cycles

















L.EIC (AED)

Example (starting on node 1) - acyclic graph



Classifying DFS Edges

Another "angle" of DFS

- A DFS visit separates the edges into 4 categories
 - Tree Edges Edges from the DFS tree
 - Back Edges Edge from a node to one of its tree ancestors
 - Forward Edges Edge from a node to one of its tree descendants
 - Cross Edges All other edges (from one branch to another)



Classifying DFS Edges

Another "angle" of DFS

- Example application: finding cycles is finding... Back Edges!
- Knowing the edge types may help to solve problem!
- Note: an undirected graph has only Tree Edges and Back Edges.

A more complex DFS application

• Decompose a graph into its strongly connected components

A **strongly connected component** (SCC) its a maximal subgraph where there is a (directed) path between each of its nodes.

An example graph with 3 SCCs:



A more complex DFS application

- How to compute SCCs?
- Let's try to use our knowledge about DFS edge types:





A more complex DFS application

• Let's look at the generated tree:



- What is the "lowest" ancestor reachable by a node?
 - ▶ 1: it's 1
 - ▶ 2: it's 1
 - ▶ 5: it's 1
 - ► 3: it's 3
 - ► 4: it's 3
 - ▶ 8: it's 3
 - ▶ 7: it's 7
 - ▶ 6: it's 7
- *Et voilà!* Here are our SCCs!

A more complex DFS application

- Let's add 2 attributes to the nodes in a DFS visit:
 - num(i): order in which i is visited
 - low(i): smallest num(i) reachable by the subtree that starts in i. It's the minimum between:
 - ★ num(i)
 - ***** smallest num(v) between all back edges (i, v)
 - ★ smallest low(v) between all tree edges (i, v)



i	num(i)	low(i)
1	1	1
2	2	1
3	3	3
4	4	3
5	8	1
6	7	6
7	6	6
8	5	4

A more complex DFS application

Main ideas of Tarjan Algorithm to find SCCs:

- Make a **DFS** and in each node *i*:
 - Keep pushing the nodes to a **stack** S
 - Compute and store the values of num(i) and low(i).
 - If when finishing the visit of a node i we have that num(i) = low(i), then i is the "root" of a SCC. In that case, remove all the elements in the stack until reaching i and report those elements as belonging to a SCC!

A more complex DFS application

Example of execution: in the moment we leave dfs(7), we find that num(7) = low(7) (7 is the "root" of a SCC)



Remove elements from stack until reaching 7; output SCC: {6, 7}

L.EIC (AED)

Graphs: Depth-First Search (DFS)

2024/2025 31/40

A more complex DFS application

Example of execution: in the moment we leave dfs(3), we find that num(3) = low(3) (3 is the "root" of a SCC)



Remove elements from stack until reaching 3; output SCC: $\{8, 4, 3\}$

L.EIC (AED)

A more complex DFS application

Example of execution: in the moment we leave dfs(1), we find that num(1) = low(1) (1 is the "root" of a SCC)



Remove elements from stack until reaching 1; output SCC:: {5, 2, 1}

L.EIC (AED)

Tarjan Algorithm for SCCs

```
index \leftarrow 1 : S \leftarrow \emptyset
For all nodes v of the graph do
  If num[v] is still undefined then
     dfs_scc(v)
dfs_scc(node v):
  num[v] \leftarrow low[v] \leftarrow index; index \leftarrow index + 1; S.push(v)
  /* Traverse edges of v */
  For all neighbors w of v do
     If num[w] is still undefined then /* Tree Edge */
       dfs_scc(w) ; low[v] \leftarrow min(low[v], low[w])
     Else if w is in S then /* Back Edge */
        low[v] \leftarrow min(low[v], num[w])
  /* We know that we are at the root of an SCC */
  If num[v] = low[v] then
     Start new SCC C
     Repeat
        w \leftarrow \text{S.pop}(); Add w to C
     Until w = v
     Write C
```

Articulation Points and Bridges

An **articulation point** is a **node** whose removal increases the number of connected components.

A **bridge** is an **edge** whose removal increases the number of connected components.

Example (in red the articulation points; in blue the bridges):



A graph without articulation points is said to be **biconnected**.

L.EIC (AED)

A more complex DFS application

- Finding articulation points is a very useful problem
 - For instance, a "robust" graph should not have articulation points that when "attacked" will disconnect them.
- How to compute? A possible (naive) algorithm:
 - Make a DFS and count the number of connected components
 - 2 Remove a node from the original graph and execute a new DFS, counting again the connnected components. If this number increased, them the node is an articulation point.
 - 8 Repeat step 2 for all nodes in the graph
- What would be the **complexity** of this method? $\mathcal{O}(|V|(|V| + |E|))$, because we will make |V| calls to DFS, each one taking |V| + |E|.
- It is possible to do much better... using a single DFS!

A more complex DFS application

An idea:

- Apply DFS to the graph and obtain the DFS tree
- If a node v has a child w without any path to an ancestor of v, then v is an articulation point! (since removing it would disconnect w from the rest of the graph)

▶ This corresponds to verify if *low*[*w*] ≥ *num*[*v*]

• The only exception is the **root** of the DFS tree. If it has more than one child in the tree... it is also an articulation point!

A more complex DFS application

• An example graph:



- num[i] numbers inside the node
- Iow[i] blue numbers
- articulation points: yellow nodes



(here we are assuming that we cannot lower the *low* value of a node going "back" on tree edges - this is why for instance low[5]=5 and not 3; the algorithm would still work even if you also assume these edges as "back edges", with low[5]=3)

A more complex DFS application



- 3 is an articulation point: $low[5] = 5 \ge num[3] = 3$
- 5 is an articulation point: $low[6] = 6 \ge num[5] = 5$ ou

$$low[7] = 5 \ge num[5] = 5$$

- 10 is an articulation point: $low[11] = 11 \ge num[10] = 10$
- 1 is not an articulation point: it only has one tree edge

Algorithm very similar to Tarjan, but with different DFS:

```
Algorithm to find articulation points

dfs_art(node v):

num[v] \leftarrow low[v] \leftarrow index ; index \leftarrow index + 1 ; S.push(v)

For all neighbors w of v do

If num[w] is not yet defined then /* Tree Edge */

dfs_art(w) ; low[v] \leftarrow min(low[v], low[w])

If low[w] \ge num[v] then

write(v + "is an articulation point")

Else if w is in S then /* Back Edge */

low[v] \leftarrow min(low[v], num[w])

S.pop()
```

Instead of a stack, we could have used colors (gray means it is in the stack)

Remember that the **root node** of the dfs must be treated differently and is an articulation point if and only if it has more than one child on the dfs tree

L.EIC (AED)