Graphs : DFS + Properties[1]

1 Depth First Search (DFS)

We start graph algorithms with the pretty intuitive, but surprisingly powerful, *depth first search* (DFS). This algorithm solves the reachabilty problem, but then in one swoop solves much more. It also runs in $O(n+m)$ time. Let's get to it.

1.1 DFS from a vertex

Our journey starts from a given vertex v in the graph. As explorers in the past have done, we begin by first planting a "flag" at v. In algorithmic terms, we have a **global** Boolean variable $\mathsf{visited}[x]$ for every vertex x initialized to 0 (false). When starting at v, our first action is to set $\mathsf{visited}[v] = 1$. Subsequently, we investigate v's out-neighbors, and if we encounter any "unflagged" neighbor x, we proceed to x remembering (via a thread, say, to give a physical mnemonic) that we came from v, and then just repeat the procedure from x. If at some point we see that all of x's neighbors are flagged, then we ravel the metaphorical thread back to the place where x was called from (in this case v), and then continue the investigation of v's other neighbors. All this thread business is sweetly handled by recursion. Here is the pseudocode for DFS from a vertex.

```
 1: global visited[$1 : n$] initialized to $0$.
 2: global $F$ initialized to $\perp$; parent[$v$] $\leftarrow \perp$ for all $v \in V$.
 3: procedure DFS($G, v$): ▷ We assume $V = \{1, 2, \ldots, n\}$
 4:     ▷ $F$ will be an out-tree rooted at $v$
 5:     visited[$v$] $\leftarrow 1$. ▷ Mark $v$ visited.
 6:     for $u$ neighbor of $v$ do: ▷ In an arbitrary, but fixed order
 7:         if visited[$u$] $= 0$ then:
 8:             Add edge $(v, u)$ to $F$; parent[$u$] $\leftarrow v$.
 9:             DFS($G, u$)
10:     return $F$
```

As you can see from the pseudocode, there is some extra stuff that the algorithm is building. Namely, the ***global*** object F. It begins with being $\perp$, and then in Line 8 edges are added to F. These edges are indeed the "threads" that were alluded to above. At the end of the call of $\mathrm{DFS}(G, v)$, the algorithm returns this collection of edges. The following theorem shows that this collection indeed has a lot of structure, and contains information about the *connectivity* properties of G.

Theorem 1. Suppose we call $\mathrm{DFS}(G, v)$. A vertex $x \in F$ if and only if x is reachable from v in G. F is a *rooted out-tree* rooted at v.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 5th Feb, 2024*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

Proof. First we prove if visited$[x] = 1$, then x is reachable from v. In fact we show that x is reachable from v in F. The proof is by induction on the *time* at which visited$[x]$ was set to 1. Imagine every time the algorithm runs Line 5, we increment time by 1. At time 1, the algorithm set set visited$[v] \leftarrow 1$ and v is indeed reachable from v in F. This is the base case. Now pick a vertex x whose visited$[x] \leftarrow 1$ is set at time t, and assume the statement is true for all vertices whose flags were set at times $< t$. The algorithm sets visited$[x] \leftarrow 1$ because of some $y \in V$ such that (a) $(y, x) \in E$, and (b) the run of DFS(G, y) calls DFS(G, x). In that case, (a) visited$[y] \leftarrow 1$ has been set strictly before time t implying, by induction, y is reachable from v in F, and (b) we add edge (y, x) to F, which then implies x is reachable from v in F.

Now for the other direction. Suppose there exists a vertex x which is reachable from v in G but visited$[x] = 0$. Since x is reachable from v, there is a path $(v = v_0, v_1, \ldots, v_k = x)$ in G. Let us pick the *last* vertex v_i in this path which has visited$[v_i] = 1$; clearly $0 \leq i < k$. Since visited$[v_i] = 1$, we have run DFS(G, v_i). But the for-loop in the algorithm would then call DFS(G, v_{i+1}) since visited$[v_{i+1}] = 0$. But that would set visited$[v_{i+1}] = 1$, and once visited a vertex is never "un-visited". This is a contradiction, and thus all vertices x reachable from v have visited$[x] = 1$.

To show F is an "out-tree rooted at v", we need to show that every vertex x in the forest which is not v has $\deg^-_F(x) = 1$, and $\deg^-_F(s) = 0$. Fix an $x \neq v$ with visited$[x] = 1$. The previous part showed that $\deg^-_F(x) \geq 1$; suppose for the sake of contradiction it was ≥ 2. Let (y, x) be the first edge (first in order of time) to be added and let (z, x) be the second. Consider the time (z, x) is being added; it must be in the call of DFS(G, z) which calls DFS(G, x). At that time, since (y, x) was already added, visited$[x] = 1$. Therefore, the if-condition would have prevented this call. □

We make a remark about a generalization of the "first part" of the above theorem (about a vertex being in F) which will be needed when we talk about DFS on the whole graph.

Theorem 2 (A stronger theorem)**.** Suppose the global variables visited$[x]$'s are not initialized to 0 but to some *arbitrary* bit-vector. Suppose we run DFS(G, v) which adds some edges to the graph F. Every vertex x reachable from v in F must have visited$[x] = 0$ during initialization but is $= 1$ after DFS(G, v) terminates. Furthermore, if there is an x such that visited$[x] = 0$ in the initialization but is $= 1$ after DFS(G, v) terminates, then x is reachable from v in F.

Proof. (Sketch) DFS(G, v) may recursively call many DFS(G, x)'s, but each such x has the property that its visited$[x]$ goes from 0 to 1. The "furthermore" part is the same proof by induction of the first part of the above theorem. □

Remark: *It is important to note something which is* **not** *true. If we initialized* visited$[x]$*'s arbitrarily (instead of all* 0*s), then it is no longer true that is we call* DFS(G, v)*, then* **every** *vertex x reachable from v will have* visited$[x] = 1$*. Can you see an example? Imagine the path graph (v, y, x) on three vertices and two edges. Suppose* visited$[y]$ *was initialized to* 1 *and* visited$[x]$ *was initialized to* 0*. Then,* DFS(G, v) *would terminate immediately without making* visited$[x] = 0$ *but x is reachable from v.*

Theorem 3. The REACHABLE? problem can be solved in $O(n + m)$ time.

Proof. Theorem 1 implies this as a corollary. Given vertex u and v, we can check if v is reachable from u by just running $\mathsf{DFS}(G,u)$, and checking if $\mathsf{visited}[v]=1$ or not. To get the path, we can use the tree F.

What is the running time of DFS? It is a recursive algorithm, so it is not completely trivial to see this. Let us fix an arbitrary vertex x, and let us figure out the time taken in the running of its pseudocode *other* than the DFS calls it makes. That is, the time for running Line 5 to Line 8. We see that the maximum time is taken in the for-loop, and this costs $O(1+\deg^+(x))$ time (the $+1$ is to take care of the corner case when $\deg^+(x)=0$). The second, and the key, observation is that in *all* the calls of $\mathsf{DFS}(G,v)$, a call $\mathsf{DFS}(G,x)$ for any vertex x is made at most once (exactly once for vertices reachable from v). This is because, once $\mathsf{DFS}(G,x)$ is called, $\mathsf{visited}[x]$ is set to 1 which prevents any further calls. Thus, the total time taken by $\mathsf{DFS}(G,v)$ is at most $\sum_{x\in V}$ the total time taken by "non-recursive" calls of $\mathsf{DFS}(G,x)$, which is $\sum_{x\in V} O(1+\deg^+(x))$. This evaluates to $O(n+m)$. □

1.2 DFS on the whole graph

The next algorithm is a *traversal* over all vertices of the graphs using the subroutine $\mathsf{DFS}(G,v)$ repeatedly. This is called the *depth first traversal* algorithm of the graph G, but is also called the depth first search (or simply DFS) of G. The input to this algorithm is the graph G and a permutation/ordering σ of the vertices. This permutation tells the algorithm the order in which to "explore" vertices, that is, to run $\mathsf{DFS}(G,v)$.

The output to this algorithm has a lot of things; these objects contain surprising amounts of information about G, as we will see below.

- One output is a couple of vectors $\mathsf{first}[1:n]$ and $\mathsf{last}[1:n]$ where for any vertex v, $\mathsf{first}[v]$ notes the "time" at which the algorithm starts exploring from v, that is, $\mathsf{DFS}(G,v)$ is called, and $\mathsf{last}[v]$ denotes the "time" the exploring ends, that is, the subroutine $\mathsf{DFS}(G,v)$ terminates.

- The other output is a *"directed forest"* F spanning all the vertices of G. Each weakly connected component in F is a rooted out-tree, directed (even when G is undirected) away from the root. Together with this we store the scalar fcomp which counts the number of these weakly connected components of F, the array $\mathsf{root}[1:\mathsf{fcomp}]$ where $\mathsf{root}(i)$ will store the root of the ith tree in F, and the array $\mathsf{Fcomp}[1:n]$ where $\mathsf{Fcomp}[v]$ contains a number between 1 and fcomp indicating the tree in which v exists.

The algorithm is simple: it has a for-loop going over all vertices in the order σ; if the vertex is unvisited, then we run $\mathsf{DFS}(G,v)$ on it starting a new tree rooted from v. We end when there are no more vertices left. Here is the full pseudocode, where we have enhanced $\mathsf{DFS}(G,v)$ to take care of what we need.

```
 1: global array visited$[1:n]$ initialized to all 0.
 2: global array first$[1:n]$, last$[1:n]$, root$[1:n]$, Fcomp$[1:n]$ initialized to all 0.
 3: global scalar fcomp, time initialized to 0.
 4: global $F$ initialized to $\emptyset$; parent$[1:n]$ initialized to all $\perp$'s

 5: procedure DFS$(G, \sigma[1:n])$: ▷ $\sigma$ is an ordering of the vertices
 6:     for $v$ in $\sigma$ do:
 7:         if visited$[v] = 0$ then:▷ $v$ hasn't been visited yet:
 8:             fcomp $\leftarrow$ fcomp $+ 1$ ▷ Increase the number of trees in the forest
 9:             root[fcomp] $\leftarrow v$ ▷ Set $v$ to be the root of the new tree
10:             DFS$(G, v)$

11: procedure DFS$(G, v)$:
12:     visited$[v] \leftarrow 1$
13:     Fcomp$[v] \leftarrow$ fcomp. ▷ Set $v$'s tree in the forest
14:     time $\leftarrow$ time $+ 1$.
15:     Set first$[v] \leftarrow$ time. ▷ Start exploring.
16:     for $u$ neighbor of $v$ do: ▷ In an arbitrary, but fixed order
17:         if visited$[u] = 0$ then:
18:             Add edge $(v, u)$ to the forest $F$; parent$[u] \leftarrow v$
19:             ▷ It will be added to the fcompth component.
20:             DFS$(G, u)$
21:     time $\leftarrow$ time $+ 1$.
22:     Set last$[v] \leftarrow$ time.
```

As we argued in the case of DFS(G, v), one can see that DFS(G, σ) takes $O(n+m)$ time as well.

Claim 1. The running time of DFS(G, σ) is $O(m+n)$ for any σ.

Remark: *Different permutations can lead to different outcomes: see Figure 1 and Figure 2. This will be **critically** used in one application of DFS.*

Edge Classification. Running DFS on a graph G with any ordering σ leads to four kinds of edges.

- (Forest Edges.) These are the edges present in F. These are marked black and solid in the Figures.
- (Back Edges.) These edges go from a descendant to an ancestor. These are marked blue and dotted.
- (Forward Edges.) These edges go from an ancestor to a descendant. These are marked red and dotted. For undirected graphs the forward edges are all back edges (there is no direction).
- (Cross Edges.) All the rest. They can be among pairs in the same component, or not. These are marked green and dotted.

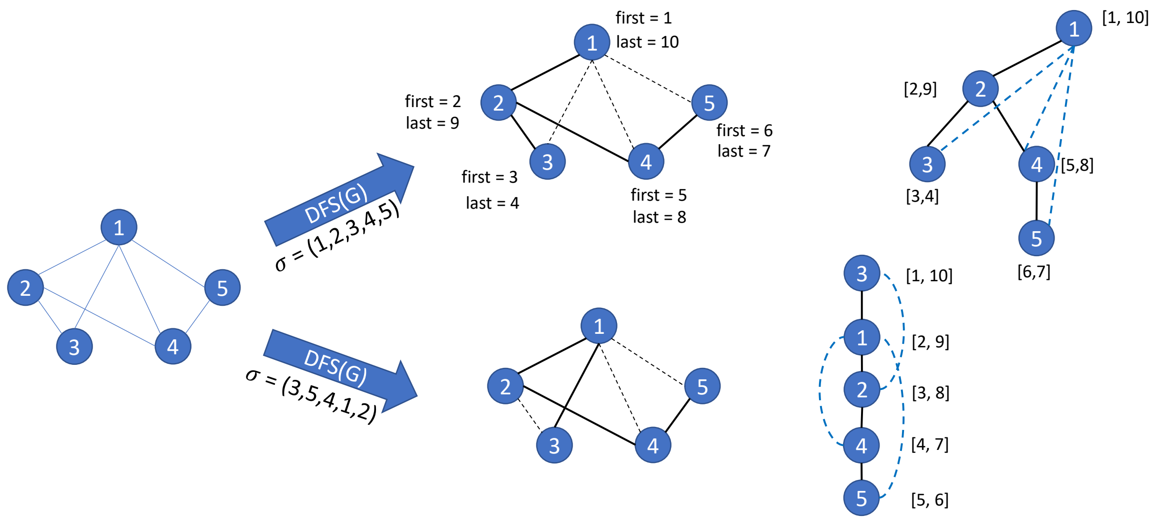

Figure 1: The edges that appear in the forest are marked in solid, while the remaining edges are dotted. The first and last are noted near the vertices. In the third figure on the right, the interval is the [first$[v]$, last$[v]$] interval.

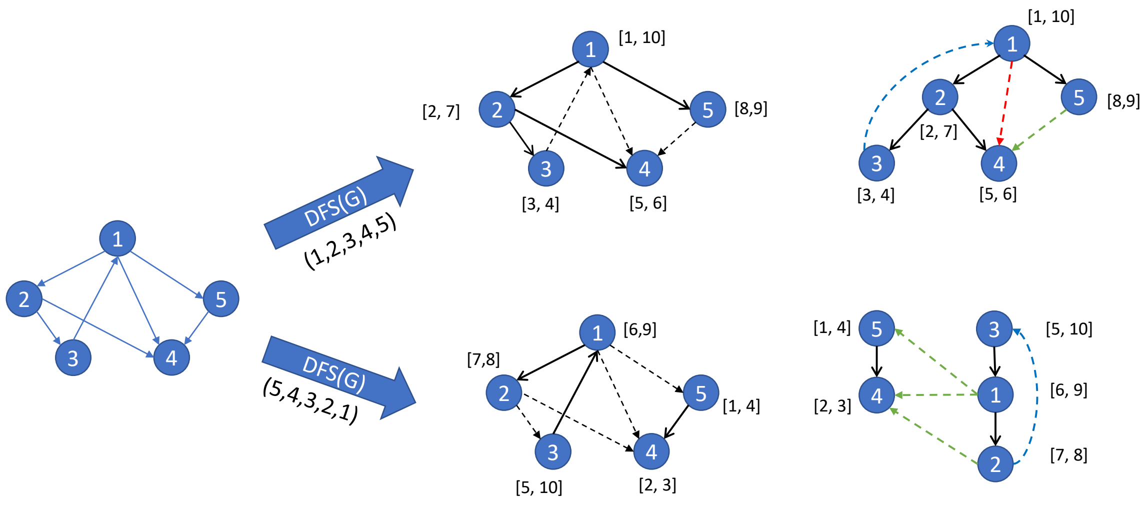

Figure 2: The edges that appear in the forest are marked in solid, while the remaining edges are dotted. The first and last are noted near the vertices. In the third figure on the right, the interval is the [first$[v]$, last$[v]$] interval.

Properties. Next, we state and prove **three** properties of the output of the DFS algorithm. Before reading the proofs, it will be useful to see their illustrations in the examples shown in Figure 1 and Figure 2 (or any other figures you have privately made). For all the properties below, we assume we have run DFS(G, σ) for some ***arbitrary*** ordering σ.

Lemma 1 (Nested Interval Property). For any two vertices u and v, with $\mathsf{first}[u] < \mathsf{first}[v]$, exactly one of the following two properties hold.

- $\mathsf{first}[u] < \mathsf{first}[v] < \mathsf{last}[v] < \mathsf{last}[u]$ *and* v is a descendant of u in F.
- $\mathsf{first}[u] < \mathsf{last}[u] < \mathsf{first}[v] < \mathsf{last}[v]$ *and* neither is a descendant of the other.

This shows that the n intervals of the form $[\mathsf{first}[v], \mathsf{last}[v]]$ don't "criss-cross" (although one may be contained in the other). This property is called the *nested* property (also called laminar property).

Proof. Since $\mathsf{first}[u] < \mathsf{first}[v]$, we call DFS$(G, u)$ before we call DFS(G, v). If v is ever discovered in the run of DFS(G, u), then (a) it will be a descendant of u in F (by Theorem 1 and Theorem 2), and (b) we must have $\mathsf{last}[v] < \mathsf{last}[u]$ since this recursive call must end before u's recursive call ends. This is case 1.

The only other case is v has not been discovered in the run of DFS(G, u). That is, DFS(G, u) ends before DFS(G, v) begins. Therefore, $\mathsf{last}[u] < \mathsf{first}[v]$ by definition of first and last. We also get v is not a descendant of u since all descendants of u are the vertices x on which we call DFS before DFS(G, u) ends. To end, u cannot be a descendant of v, because every descendant x of v must start DFS after v starts, that is, $\mathsf{first}[x] > \mathsf{first}[v]$. But, $\mathsf{first}[u] < \mathsf{first}[v]$. □

The above property is useful, and will be useful in proving some other properties below. But it also allows us to classify the edges (not in the forest F) just looking at the first and the last values.

- (Back Edges.) Edges $(u, v) \in E \setminus F$ with $\mathsf{first}[v] < \mathsf{first}[u] < \mathsf{last}[u] < \mathsf{last}[v]$.
- (Forward Edges.) Edges $(u, v) \in E \setminus F$ with $\mathsf{first}[u] < \mathsf{first}[v] < \mathsf{last}[v] < \mathsf{last}[u]$.
- (Cross Edges.) Edges $(u, v) \in E$ such that the intervals $[\mathsf{first}[u], \mathsf{last}[u]]$ and $[\mathsf{first}[v], \mathsf{last}[v]]$ are disjoint.

Lemma 2 (Edge Property). Let (u, v) be any edge in G with $\mathsf{first}[u] < \mathsf{first}[v]$. Then, we must have $\mathsf{last}[v] < \mathsf{last}[u]$.

Proof. Suppose not. By the Nested Interval Property, we must have $\mathsf{first}[u] < \mathsf{last}[u] < \mathsf{first}[v] < \mathsf{last}[v]$. That happens when DFS$(G, u)$ terminates before $\mathsf{visited}[v]$ is set to 1. But the Line 16 would discover v contradicting the above. □

We are now ready for the first application of DFS – we can solve CONNECTED COMPONENTS of an Undirected Graph using the following lemma.

Lemma 3. Let $G = (V, E)$ be any undirected graph and consider the forest F returned by DFS(G, σ) with any permutation σ. The components of F are precisely the connected components of G.

Proof. Let $V_1, \ldots, V_k$ be the vertices in the various trees of the forest F. Clearly $G[V_i]$ is connected since they are connected in the forest. We claim that there is no edge of the form (u, v) with $u \in V_i$ and $v \in V_j$. Suppose there is, and without loss of generality assume $\mathsf{first}[u] < \mathsf{first}[v]$ (this is where we are using the undirectedness of G). By the edge property, we have $\mathsf{first}[u] < \mathsf{first}[v] < \mathsf{last}[v] < \mathsf{last}[u]$. But this means v is a descendant of u in F contradicting the fact they exist in different connected components of F. □

Theorem 4. CONNECTED COMPONENTS of an undirected graph can be found in $O(n+m)$ time by running DFS(G, σ) for any ordering σ.

Moving on to more properties.

Lemma 4 (Path Property). If $(u = v_1, v_2, \ldots, v_k = v)$ is a path in G from u to v such that $\mathsf{first}[u] < \mathsf{first}[v_i]$ for all $2 \leq i \leq k$, then, $\mathsf{last}[v_i] < \mathsf{last}[u]$ for all $2 \leq i \leq k$.

In English, if there is a path from a vertex u to a vertex v such that u is the first vertex to be discovered among them, then all the vertices in the path are descendants of u in the DFS forest.

Proof. Suppose not. Choose the smallest $2 \leq i \leq k$ for which $\mathsf{last}[u] < \mathsf{last}[v_i]$. By the choice of i, we get $\mathsf{last}[v_{i-1}] < \mathsf{last}[u]$. Also note (v_{i-1}, v_i) is an edge.

Case 1: $\mathsf{first}[v_{i-1}] < \mathsf{first}[v_i]$. In this case, the Edge Property would imply $\mathsf{last}[v_i] < \mathsf{last}[v_{i-1}]$, and thus $\mathsf{last}[v_i] < \mathsf{last}[u]$. Which is what we supposed wasn't true.

Case 2: $\mathsf{first}[v_i] < \mathsf{first}[v_{i-1}]$. In that case, we see $\mathsf{first}[u] < \mathsf{first}[v_i] < \mathsf{last}[u] < \mathsf{last}[v_i]$ which violates the Nested Interval Property. □

The above property allow us immediately to solve the CYCLE? problem. The following theorem implies the algorithm: run DFS(G, σ) and check if any of the edges is a *back edge* (which is one linear time scan over all the edges and checking the first and the last).

Lemma 5. A graph G is **acyclic** if and only if there are no back edges.

Proof. One direction is trivial – if G has a back edge, then there is clearly a cycle. If the back-edge is (u, v), then by definition there is a path from v to u using F-edges, and then take the (u, v) edge back.

The other direction is more interesting. If G has a cycle C with k vertices $(v_1, \ldots, v_k, v_1)$, then without loss of generality let v_1 be the vertex with the smallest $\mathsf{first}[v_i]$ in this cycle. Since there is a path from v_1 to v_k, using the Path property and the fact that $\mathsf{first}[v_1]$ is the smallest, we get $\mathsf{first}[v_1] < \mathsf{first}[v_k] < \mathsf{last}[v_k] < \mathsf{last}[v_1]$. But this implies (v_k, v_1) is a back-edge. □

Theorem 5. CYCLE? can be solved in $O(n+m)$ time using DFS.